Strong Static Type Checking for Functional Common Lisp

Robert L. Akers

Technical Report 96 December 30, 1993

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

Abstract

This thesis presents a type system which supports the strong static type checking of programs developed in an applicative subset of the Common Lisp language. The Lisp subset is augmented with a *guard* construct for function definitions, which allows type restrictions to be placed on the arguments. Guards support the analysis and safe use of partial functions, like CAR, which are well-defined only for arguments of a certain type.

A language of type descriptors characterizes the type domain. Descriptors are composed into function signatures which characterize the guard and which map various combinations of actual parameter types to possible result types. From a base of signatures for a small collection of primitive functions, the system infers signatures for newly submitted functions.

The system includes a type inference algorithm which handles constructs beyond the constraints of ML-style systems. Most notable are the free use of CONS to construct objects of undeclared type and the use of IF forms whose two result components have unrelated types, resulting in *ad hoc* polymorphism. Accordingly, the type descriptor language accommodates disjunction, unrestricted CONS, recursive type forms, and *ad hoc* polymorphic function signatures. As with traditional type inference systems, unification is a central algorithm, but the richness of our type language complicates many component algorithms, including unification. Special treatment is given to *recognizer* functions, which are predicates determining whether an object is of a particular type. Type inference in this setting is undecidable, so the algorithm is heuristic and is not complete.

The semantics of the system are in terms of a function which determines whether values satisfy descriptors with respect to a binding of type variables. The soundness of each signature emitted by the system is validated by a signature checker, whose properties are formally specified with respect to the formal semantics and proven to hold. The checker algorithm is substantially simpler than the inference algorithm, as it need not perform operations such as discovering closed recursive forms. Thus, its proof is both more straightforward to construct and easier to validate than a direct proof of the inference algorithm would be.

Chapter 1
INTRODUCTION

Types in a programming language are a means of organizing the representation of program objects at a level of abstraction appropriate to the language. As such, they provide a means for subjecting data representation issues to distinct analysis. Inconsistent views of the representation can be isolated as type errors. A common belief is that a very large portion of programming errors are manifested as type errors. Early exposure of type errors reduces the cost of software development.

For the purposes of this discussion, I will adopt the following terminology:

- A *type error* occurs when an operation is applied to an operand of an incompatible type, or when the type of a construct does not match that expected in its context.
- *Static type checking* is type checking performed by a compiler or pre-processor.
- *Dynamic type checking* is type checking performed while the program is running.
- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A *type checker* implements a type system.
- A language is *strongly typed* if its compiler or another pre-processor can guarantee that the programs it accepts will execute without type errors.[1]
- *Polymorphic* languages are those in which some values and variables may have more than one type, and *polymorphic functions* are functions whose operands can have more than one type.
- *Polymorphic types* are types whose operations are applicable to operands of more than one type. [Cardelli 85]
- *Type inference* is the problem of determining the type of a language construct from the way it is used.

We assume the reader has at least a rudimentary familiarity with Lisp.

---

[1] These definitions, adopted by Aho, Sethi, and Ullman [Aho 86], differ from some others. For example, Horowitz [Horowitz 84] says that a language is strongly typed if the type of all expressions is known at compile time. Cardelli and Wegner [Cardelli 85] apply Horowitz's definition of strongly typed to the term *statically typed*, and accept Aho's definition of strong typing.

1.1 Static Type Checking and Common Lisp

Common Lisp [Steele 90] is often referred to as an untyped programming language.[2] To the contrary, Lisp has a number of well-documented data types, including characters, various kinds of numbers, arrays, and CONS-es. But rather than suggesting the absence of types in Lisp, the claim that Lisp is untyped generally refers to one of two notions:

- There is little static type checking at function definition time or at compile time.
- In the vast universe of structures which can be composed with CONS, the most explicit typing normally ascribed is simply that the structure is a CONS, i.e., it is an ordered pair with a CAR (head) and a CDR (tail).

Although some Lisp compilers will do perfunctory type checking when constant values are embedded in a program, there are generally no type restrictions on variables, and thus no type checking on variables is possible until run-time, when the variables are bound to specific values. The inability to type-check CONS structures is especially problematic, partly because of the ease with which arbitrary structures can be created. A major deterrent to static type checking in Lisp is the presence of destructive operations on CONS structures. If allowed in its full generality, this upsets the locality of type analysis and makes static type checking infeasible. It would be possible to allow destructive updates which preserve the type of the updated object, but even a determination that the type was being preserved would require a search of the program context, since a function higher in the call tree might have a more restrictive view of the type than is necessary in the local context.

The lack of a strong static type discipline hampers Common Lisp in several respects. Software engineers highly value the early detection of errors, recognizing the high cost of correction later in the software cycle. The lack of type checking and the lack of thorough typing of structured objects in Common Lisp is a serious deterrent to the employment of Lisp. In its lack of type checking and type annotation, the problem of maintaining type consistency in Lisp is much like the same problem in Backus' original FP language [Backus 78], and it has been noted that misunderstandings of data structures are the most common errors in FP programming [Pozefsky 78]. If anything, the open-ended flexibility of Lisp makes the problem even more serious than in FP.

Furthermore, the dynamic type checking complicates compilation and degrades run-time performance of compiled programs. If a compiler is to generate code that supports diagnosis and debugging of type errors, it must generate code that checks for type errors prior to any operation which may expose them. The normal execution of this code incurs substantial performance overhead, which contributes to the reputation of Lisp as being inefficient. To mitigate this inefficiency, Common Lisp [Steele 90] gives compilers the option of offering as many as four safety levels. The KCL compiler [Yuasa 85], for example, may be instructed to generate code without type checks that "runs" fast but behaves without accountability when errors occur. Clearly, a better solution, which might be provided by strong static type checking, is generation of fast code combined with the assurance by static analysis that type errors will not occur. The program would then have the same behavior with respect to input and output, whether or not type checking code were generated.

A third motivation lies in support of automated formal reasoning about Lisp programs. Typically, reasoning about total functions is much more straightforward than dealing with partial functions. The Boyer & Moore theorem prover Nqthm [Boyer & Moore 88], for example, implements a logic bearing a clear resemblance to a functional subset of Lisp, but with the important distinction that all functions, in particular functions like CAR, are total and default to standard values in cases where Lisp functions would

[2] Cardelli [Cardelli 89] refers to Lisp as a *type-free* language, meaning that there is run-time typing, but no static typing.

break. If the simplicity of having a domain of total functions were to carry over to the analysis of Common Lisp functions, some mechanism would be necessary to guarantee that function calls like (CAR 3) could never occur in functions admitted to the logic. Static type analysis is one obvious way to approach this problem.

Fortunately, a functional subset of Common Lisp may lend itself to this kind of analysis. Consider a subset which is purely applicative (with no destructive list operators) and is thereby referentially transparent.[3] With such a subset, Hindley-Milner style type inference may be possible, as it is in ML [Hindley 69, Milner 78]. Lisp, however, is much more flexible than ML-style languages in ways that will complicate the straightforward classical inference algorithms. The crux of this thesis is to deal with these complications in developing a strong static polymorphic type checking system for an applicative subset of Common Lisp.

A desirable side effect of the effort deals with the clarity of Lisp programs. Pre- and post-conditions [Hoare 69], in addition to being useful tools for formal reasoning about imperative programs, are valuable program documentation. They provide a specification of the domain and range of a function which a user may use both as an aid to writing the function and as a guide in determining when an existing function may be properly utilized. Type signatures are weak pre- and post-conditions. Since our strong static typing discipline results in type signatures for functions, this expressive benefit can be added to Lisp functions where it has previously been missing.

1.2 A Comparison of ML-style Languages and Functional Common Lisp

This effort targets a subset of Common Lisp which is applicative, and hence has relatively uncomplicated semantics, yet which presents some type checking problems which are not faced by the family of ML-style languages. What about this subset, then, lies outside the well traveled territory of ML and its descendents?

ML supports *parametric polymorphism* of functions (a term originated by Strachey [Strachey 67]) which means that functions may work normally on an infinite number of types, all of which have a given common structure. Lisp functions, however, may exhibit both parametric polymorphism and *ad hoc* polymorphism (also due to Strachey), whereby functions may work on objects of unrelated types, perhaps by executing different code. Furthermore, the various objects returned by a given Lisp function may be of completely unrelated types. One may view all Lisp conditional forms as variations of IF. The IF expression was invented by McCarthy [McCarthy 62] apparently to support the explicit definition of recursive functions, but it may be used in a more general manner to achieve *ad hoc* polymorphism. The two result arms of Lisp IF forms may have completely different types, and both must be considered options for the result type. By contrast, the result arms of IF forms in ML-style languages must have the same type, or at least unifiable types. This feature of Lisp would throw a significant monkey wrench in ML type inference by limiting the use of classical unification, which is ubiquitous in the inference algorithm.

Another equally critical distinction is that Lisp offers relative flexibility in the composition of structures. The ML family offers only restricted versions of structured types: lists, all of whose elements must be of the same type; tuples, which are distinct from lists and whose elements are of specified type; and

---

[3] Referential transparency is the property that any expression can be replaced by any other expression which has the same value without affecting the value of any larger expression of which it is a part. Conventionally stated, "Equals can be substituted for equals."

explicitly prescribed structures like the algebraic types of Miranda [Turner 85]. Declarations of algebraic types are required and specify unique constructor and selector functions. There is no commonality among these structured types. They are constructed differently, represented differently, and are not comparable. All are strictly typed. A representation of the type structure of Miranda, for example, in terms of a grammar for type descriptors, might be (where italicized identifiers are type variables in the type descriptor grammar and *, **, ... are generic type variables in the Miranda language):

```
<type> ::= NUM | CHAR | BOOL | (*LIST <type>) |
           (*TUPLE t1, .., tn), n >= 2 |
           t1 -> t2    {function types} |
           *, **, ... | UNDEF | <algebraic> | <type variable>

<algebraic> ::= (*ALGEBRAIC <struct list>)

<struct list> ::= <struct> | <struct> <struct list>

<struct> ::= <constructor name> t1..tn
```

Algebraic types, described with the <algebraic> descriptor, are like tuples in that they can be used to construct mixed structures, but only if the algebraic type is previously declared. Algebraic objects are tagged in the concrete representation.

Lisp CONS structures, on the other hand, are arbitrarily irregular. Lists, all manner of trees, and arbitrary tuples are all constructed with the binary CONS function. S-expressions may be viewed as having multiple types, in the sense that a singleton list may also be considered to be a singleton tuple or a brutally pruned tree. Even the predefined value NIL is overloaded, since it is routinely viewed as both a (negative) Boolean value and as an empty list. Dealing with this flexibility raises significant complications in many respects of the type inference problem, including canonicalization of type information, unification of type descriptors, and dealing with a multiplicity of potential recognizer functions.

A grammar for type descriptors for a functional Common Lisp might be:

```
<descriptor> ::= <simple descriptor>  |  <variable>  |  *EMPTY  |
                 *UNIVERSAL | (*CONS <descriptor> <descriptor> )  |
                 (*OR <descriptor>* )  | <rec descriptor>

<simple descriptor> ::=
    $CHARACTER | $INTEGER | $NIL | $NON-INTEGER-RATIONAL |
    $NON-T-NIL-SYMBOL | $STRING | $T

<rec descriptor> ::= (*REC <rec name> <recur descriptor> )

<rec name> ::= a symbol whose first character is not "&"

<variable> ::= a symbol whose first character is "&"

<recur descriptor> ::=
   <simple descriptor>  |  <variable>  |  *EMPTY  |
   *UNIVERSAL | (*CONS <recur descriptor> <recur descriptor> )  |
   (*OR <recur descriptor>* )  | <rec descriptor> |
   (*RECUR <rec name>)
```

This is, in fact, the grammar employed in the function signature checker described in this thesis. The *CONS descriptor constructs a binary object from arbitrary pieces. The *OR descriptor can be used to characterize *ad hoc* polymorphism of function types. These characterize two of the factors which will most distinguish Lisp type inference from the classical algorithms. *REC descriptors, which allow for arbitrary recursive structures using the *CONS and *OR constructs, extend the impact of these distinctions.

Yet another characteristic which will distinguish a Lisp type system from an ML-style system is the structure of function signatures. A function signature typically maps a list of parameter types to a result type. For instance, the type of a binary integer addition function can be expressed:

```
(INTEGER INTEGER) -> INTEGER
```

ML-style function types need only include a single expression of this form, even when they are polymorphic, since their polymorphism is parametric. For example, an ML function extracting the first element of a list would have the signature:

```
(* list) -> *
```

where the "*" is a type parameter which can be instantiated with any ML type. But the *ad hoc* polymorphism of Common Lisp functions means that often no single expression of this form can capture a function signature. Differently typed parameters can produce results of unrelated type. A schema for Lisp signatures can accommodate *ad hoc* polymorphism by including multiple components, or *segments*. For example, a modified CAR function which defaults to 0 if the parameter is something other than a CONS might have the signature:

```
(*CONS &1 *UNIVERSAL) -> &1

(*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
     NON-T-NIL-SYMBOL $STRING $T) -> $INTEGER
```

Multiple-segment signatures complicate the type inference problem for Common Lisp and further distinguish it from ML-style type inference.

1.3 Synopsis

This thesis presents a type system which supports the strong static type checking of programs developed in a simple subset of the Common Lisp language. A language of type descriptors characterizes the type domain. Descriptors are composed into function signatures which characterize the type requirements placed on actual parameters to the functions, and which map various combinations of actual parameter types to possible result types. From a base of signatures for a small collection of primitive functions, the system infers signatures for newly submitted functions. The signature inferred for each new function is shown to be sound with respect to a formal semantics for the type system by application of a signature checker whose properties are formally specified and proven to hold.

The language subset is limited to functions of fixed arity, where the only forms allowed in the body are references to the formal parameters, IF forms of arity three, calls of previously defined functions, literals (of type character, integer, rational, T, NIL, or string), and quoted forms where the quoted object is either a literal, a symbol, or a CONS of any such objects (recursively). All data objects are assumed to be of finite size. Although there is no formal requirement that functions always terminate, the inference algorithm was designed with terminating functions in mind, and submission of non-terminating functions may result in weak results.

We augment the Common Lisp subset with a *guard* construct, which may be included as part of a function definition. A guard is a predicate on the function arguments which essentially restricts the domain of the function. The function is well-defined on some arguments only if the guard evaluates to a non-NIL value. Typically, a guard can characterize type restrictions on the parameters which will guarantee the absence of type errors in the function body.

The type system includes a type inference algorithm in the tradition of ML. But it handles constructs which violate the constraints of ML and its successors, most notably the free use of CONS to construct objects of undeclared type and the use of IF forms whose two result arms have objects of totally unrelated type, resulting in *ad hoc* polymorphism. Accordingly, in addition to the primitive types of Lisp, the type descriptor language includes type variables and forms for disjunction, unrestricted use of CONS, and a similarly unrestricted recursive type constructor. As with ML type systems, unification is the central algorithm. But because of the richness of the type language, and since unification is performed as part of a process of discovering closed form descriptors for recursively constructed data structures, the unification algorithm is vastly more complicated.

The type descriptor language is used to express type signatures for functions. Among other things, a signature includes a *guard descriptor*,which is a list of descriptors whose length is equal to the arity of the function, and whose elements each characterize the type requirements on the corresponding function argument. The signature also includes a collection of *segments*, where a segment is a list of argument type descriptors paired with a result descriptor. For any values satisfying the guard of a function, there exists a segment for the function such that the values satisfy the argument descriptors in the segment, and the result of evaluating the function on those values satisfies the result descriptor. The system maintains a database of signatures for all existing functions. Given a new function, the inference system attempts to generate a signature. When it succeeds, it adds it to the database. When the function's guard can be completely captured within the type system, and when the same is true for all the functions in its call tree, we will show that the signature is correct, that whenever the function's guard is satisfied by some actual parameters (i.e., the guard evaluates to a non-NIL value), evaluating the function call will not result in a guard violation, and furthermore that to establish that the function's guard is satisfied, it is sufficient to show that its guard descriptors are satisfied by the actual parameters.

Typically, guards, as well as predicates used as IF tests, include applications of type *recognizer* functions to formal parameters. A recognizer is a Boolean-valued, single-argument function of a certain form which determines whether its argument has a particular type within the type system. Some recognizer functions, such as INTEGERP, CHARACTERP, and CONSP, are in the primitive subset. But others may be submitted as new functions to recognize objects of arbitraily complex type. Recognizer functions are spotted by the inference tool and given special treatment, as they can play an important role in inferring precise function signatures. In particular, a guard can be totally captured in the type system if it is expressed as a conjunction of recognizer calls on distinct formal parameters. In this case, we say the guard descriptor is *complete*.

The semantics of the system are in terms of a function which interprets descriptors and values with respect to a binding of type variables. This function determines whether a value satisfies a descriptor under the binding. The soundness of the system is guaranteed by a result checker, which has been proved correct. The checker validates each signature emitted by the inference system. The checker algorithm is substantially simpler than the inference algorithm, as it need not perform operations such as finding closed recursive forms and negating descriptors. Thus, its proof is both simpler to construct and easier to validate than a direct proof of the inference algorithm would be.

The inference system is not shown to be complete, as the general problem of type inference in this domain is undecidable. Neither is the algorithm proven to terminate, though a collection of safeguards have enabled it to withstand rigorous testing without an instance of non-termination.

The rest of this document is structured as follows. The next chapter is a survey of previous work related to the topic. Chapter 3 is an overview of the complete system, including an introductory explanation of function signatures and their derivation. Chapter 4 is a detailed guide to the type inference algorithm.

Chapter 5 presents the formal semantics of the type system. Chapter 6 is a detailed presentation of the algorithm used to check the validity of function signatures generated by the inference algorithm. Chapter 7 is a proof of soundness for the checker described in Chapter 6. Since the proof is voluminous, many significant components are incorporated by reference from a proof appendix. Chapter 8 describes the performance of the system on an extensive test suite. Chapter 9 discusses future work on the topic and makes concluding remarks. Other appendices include a description of the initial state of the system, discussion of some alternate semantic models which were explored, and the Lisp code implementing some selected functions in the system.

Several other supplementary materials are available via anonymous ftp at the time of writing. These include the source code implementing the entire system, the postscript for this document, and a log of tests conducted on the system. Appendix H describes everything that is available and gives instructions on how to retrieve it.

Chapter 2
BACKGROUND AND RELATED WORK

In this section, we will examine the various fundamental characterizations of *type* relevant to the body of work on automatic type inference. Then we will survey the various related threads of research into recursive types, functional programming languages, polymorphism, and type inference, with particular emphasis on the lineage of ML. Next we will look at previous efforts to bring static type checking to Lisp systems in particular, and finish with references to the relationship between formal type checking systems and optimizing compilers.

2.1 What Is a Type?

The term *type* has roots in mathematics which significantly predate computers. Bertrand Russell [Russell 08] developed a theory of types to deal with paradoxes involving self-referencing sets. His notion of type was "the range of significance of a propositional function, i.e, the collection of arguments for which the said function has values." Church's "A Formulation of the Simple Theory of Types" [Church 40] partially incorporated the calculus of lambda conversion into a theory of types adapted from Russell's original theory. Hoare and Allison [Hoare 72] give a very readable presentation of the resolution of Russell's paradox with a typed system.

Though types have been manifest in programming languages since the inception of "high level" languages like FORTRAN in the late 1950's, for years their status was more intuitive than formal. The primary motivation was to distinguish different classes of data, partly because of their different storage requirements and partly because of their different semantic properties. The underlying intuition is that a type characterizes a set of values, presumably with some semantic or structural commonality, which can be assumed by a variable or expression. Hoare [Hoare 73] characterized a type with a grammar for the language of constant expressions in the set, involving only basic constructor functions for the type. His work was in the spirit of types as sets of values, but his formal, abstract approach filtered concrete representation issues away and placed types in a more algebraic setting.

The Simula 67 language [Dahl 66] was the first appearance of the notion of classes, which bundle a set of values with the primitive operations over the value set. The goal of the Simula mechanism was not the enrichment of type systems, however; the first reference in the literature with this mindset may be due to John Laski [Laski 68]. J. H. Morris also recognized the significance of classes in providing type abstraction, and expanded on it in his work. Morris [Morris 73] maintained that the specification of a type should include not only a set of values, but also a collection of basic operations for manipulating the objects. This packaging of operations with types has become a central feature in the conventional style of data abstraction supported in many programming languages. Examples of this approach include

SML [MacQueen 84a], Ada [DoD 83], Miranda [Turner 85], and Quest [Cardelli 89].[4] Object-oriented programming languages also take Morris' approach, though in a significantly different style.

Straying even further from the simple notion of types as sets of values, in the Russell language [Demers 78, Demers 80a] "a data type is a set of named operations that provide an interpretation of the values of a single universal value space common to all types." Demers and Donahue were following the work of Dana Scott, who defined types as *retracts* of a universal domain [Scott 76]. This treatment allows data types to be treated as first class values. Russell accomplishes static type checking and has a general parameterization mechanism supporting parameterized types, types as parameters, and a disciplined form of self-application. Any construction in the language can be parameterized with respect to any of its free identifiers using call-by-value semantics. The type system includes a calculus of *signatures*, which are type descriptors associated with every identifier and expression. Demers and Donahue use this uniformity to argue for "type completeness" as a language design principle [Demers 80b]. One impact of the universal value space on the type system is that the notion of "error" becomes one of "misinterpretation" of data caused by applying an operation inappropriate for that type of object. An excellent general discussion of the Russell approach to types and type checking is presented in [Donahue 85].

MacQueen and Sethi [MacQueen 82] developed a model of types as *ideals*, or sets with an object domain which satisfies certain closure conditions. Essentially, these conditions are that the structure characterized by types is preserved when going "downward" to approximations and "upward" to least upper bounds of consistent sets of values. Their formulation follows Shamir and Wadge [Shamir 77][5] and Milner [Milner 78]. Within their theory, polymorphic functions are expressed as conjunctions of types. They introduce higher order types, or *kinds*, to help formulate operations for constructing new types from existing ones. MacQueen, Sethi, and Plotkin [MacQueen 84b] further developed this work by developing a metric structure on their sets to establish the existence and uniqueness of solutions of most recursive type equations. The set of admissible type expressions are those which are *contractive* in their algebra.

In contrast, Leivant contends that a type is a structural condition on data objects [Leivant 83a]. Such a condition can be conveyed fully by syntactic expressions in a type discipline which simply encode the ways in which data objects are allowed to interact.

Type theorists are also exploring other approaches to the notion of type as it applies to programming. Focusing on abstract data types, Goguen et al. [Goguen 78] define a type as an isomorphism class of an initial (many-sorted) algebra[6] in some category. Thereby they have a schema for discussing the algebraic properties of a type at a high level of abstraction, while maintaining the ability to map this level of abstraction to an isomorphic implementation. Kamin [Kamin 80] identifies a type with the isomorphism class of a final algebra, whereby every element of a type is the sole member of its equivalence class with respect to all the operations on the type, and thus the type is maximally compact. In Martin-Lof's constructive type theory, types correspond to propositions in a formal logic [Martin-Lof 79]. This approach reduces logic to type theory, and it defines a constructive logic as opposed to the conventional classical logic. The rules of logic are derived from the definitions and rules for types, and the result is a natural deduction system which is strong enough to formalize constructive mathematics and which allows types to be specified with extremely fine granularity. This view is also described by Constable [Constable

---

[4] None of these languages apply this packaging uniformly over their type systems, however. Rather, they provide it as a special feature for defining abstract data types; their primitive types are in the straightforward ALGOL/Pascal tradition.

[5] Shamir and Wadge were among the first to formulate a semantics of data types wherein types were themselves first class objects in the domain of data objects.

[6] An algebra here is a family of sets with a collection of operations among them.

80] and used in his PL/CV project and in the Nuprl environment [Constable 86]. Steensgaard-Madsen [Steensgaard-Madsen 89] proposes that a type is a set of functions defined in the lambda calculus. All these efforts are motivated by factors not particularly related to the concerns of type checking Lisp programs. They are nevertheless interesting in that they carry the potential benefit of type systems much further into the semantic realm than the more syntactic purposes of guaranteeing representation compatibility and supporting modularity.

2.2 Recursive Data Types

Definition of recursive data types has been long supported in programming languages. Pointer types, popularized in ALGOL68 [VanWijngaarden 69], PL/I [Beech 70], and Pascal [Wirth 71] support user-defined recursive structures using pointers. A pointer mechanism is also implicit in the CONSes of LISP [McCarthy 62].

Several formalizations of algebrae for expressing recursive types are in the literature. McCarthy [McCarthy 63] and Hoare [Hoare 71, Hoare 73] each gave general algebraic specifications of generalized recursive type definition schema and used their schema to help define sets of Lisp primitive functions. Lewis and Rosen [Lewis 73] also formulated a view of recursive data types, one which allowed them to resolve, for example, whether two types expressed in different terms within a type system in fact represent the same type.

2.3 Functional Languages, Polymorphism, and Type Inference

The most significant advances in type inference and polymorphism have come from a lineage of work which culminated with ML, but which continues to this day. With foundations in the lambda calculus, the type system and type inference mechanism supporting the ML language are milestones in the history of programming language development. Because of its success, the ML type system has been employed with minor variations in a number of more recent languages. The next section surveys the ML lineage, including its roots in the study of lambda calculus and the continuing development of its ideas in succeeding efforts. It is followed by sections covering other significant work on polymorphic type systems and type inference algorithms. For more general background, Hudak [Hudak 90] provides an excellent review of the functional programming paradigm, its history, and its underpinnings in the lambda calculus.

2.3.1 ML, Its Predecessors, and Its Successors

J. H. Morris laid important theoretical groundwork for dealing with parametric polymorphism in his thesis [Morris 68] on the lambda calculus. Although he did not specify a polymorphic type system, he recognized the type inference problem and showed how a valid type assignment could be found for a lambda calculus term by solving a set of simultaneous linear equations. His axiomatic description of formal rules allowed type inference to be studied separately from the underlying intuition that types are sets of values. His algorithm was the basis for the work of Milner and established a relationship between type inference systems and functional programming which has been developed in most succeeding functional language systems. In later work [Morris 73], Morris proposed a type system including modules, which were an improvement over Simula's classes [Dahl 66]. Though he was working primarily to develop and strengthen notions of type abstraction, he observed that his type system provided a limited form of polymorphism.

Reynolds [Reynolds 74] formalized a notion of type structure similar to Morris', introducing an extension of the typed lambda calculus which supported user defined types and polymorphic functions. The key idea was to introduce types as values and to allow them to be passed as arguments to functions and be bound to type variables. This provided an explicit form of polymorphism in terms of type parameters, and became known as the *polymorphic* or *second order typed lambda calculus*.[7] The use of type parameters was incorporated into the programming languages Alphard [Wulf 78], Russell [Demers 79], and CLU [Liskov 76, Liskov 78, Liskov 81], but without the full generality of Reynolds' system, since types in those languages are identified with sets of operations, and thus the polymorphism is limited to the operations used.

Hindley [Hindley 69] developed a method for deriving the *principal type scheme* of an object in combinatory logic. A type scheme discovered by an algorithm is *principal* if it is the most general type scheme which can be inferred for the expression from the rules and axioms of the type system. Principal type schemes support the notion of polymorphic type, though Hindley did not use that terminology. Hindley also noticed that the unification algorithm developed by Robinson [Robinson 65] is applicable to the type inference problem.

ML [Gordon 79, Milner 84] is the landmark programming language for utilizing type inference in the presence of parametric polymorphism. ML types can contain type variables that are instantiated to different types in different contexts. Hence, it is possible to write functions which can operate uniformly over objects of different type, so long as there is sufficient commonality among the objects to support the operation. Milner [Milner 78] described the theory of polymorphism and type inference around which ML was built. He demonstrated a type inference algorithm which accepts any well-typed ML program and determines its most general type signature. Furthermore, he used the formal semantics of the ML language to show that any program which is accepted by the type checking algorithm is guaranteed to be free of type errors. Consequently, the implementation need not tag program objects for the purpose of dynamic type checking.[8] Since all types can be deduced by the system, the user does not need to provide type declarations or type signatures on program objects.

The Hindley-Milner type system is based on an extension of the pure typed lambda calculus. It may seem strange to say that a language which requires no type signatures has the typed lambda calculus as its abstract model. But in fact, the only ML expressions which pass the type checker are those for which types can be supplied to obtain valid typed lambda calculus expressions. Thus, the type inference mechanism allows type structure to be automatically recovered from a program in which it may be missing. Good, pragmatic discussions on Hindley-Milner style type inference are provided by Damas and Milner [Damas 82], Cardelli [Cardelli 84a], and Peter Hancock [Hancock 87], the latter in terms of Miranda. A clear discussion of the Hindley-Milner system contrasted with Reynolds' polymorphic lambda-calculus is given by Giannini [Giannini 85].

Hindley's and Milner's use of an extension to the pure calculus is motivated by the premise that the pure typed lambda calculus is not sufficiently expressive. Fortune et al. [Fortune 83] showed that every term in the calculus can be shown to be *strongly normalizable*, meaning that there exists a normal form and an algorithm for reducing expressions to their normal form. Consequently, the family of functions which can be computed (the primitive recursive functions) is too restricted, especially compared to those definable in

[7] This calculus was invented independently by Jean-Yves Girard [Girard 72].

[8] Moreover, Appel [Appel 89] asserts that run-time tags are not necessary for statically-typed polymorphic languages at all. Aside from type checking, he asserts that tags are needed only for garbage collection, and he proposes a garbage collection algorithm which makes use of type information which the compiler can provide. Though no such garbage collector has been implemented, he used some reasonable assumptions to estimate that the performance of such a garbage collector would be comparable to that of a conventional system and better in some scenarios.

the second order typed lambda calculus. The pure calculus lacks the power of lambda-definability, thus self-applicative forms, such as Church's Y-combinator, which was used to prove his fixpoint theorem, cannot be type checked. The fact that the pure typed lambda calculus has no general fixpoint operator can be treated by adding a domain of constants to the typed lambda calculus, by including in the domain a constant fixpoint operator for every type, and by adding a conversion rule for each of the fixpoint operators. Polymorphism can then be achieved in the calculus by adding a domain of type variables and extending the domain of types accordingly. But if the calculus includes type variables, there can be problems with type checking. Type checking in this calculus is a variant of the problem of *partial polymorphic type inference*, which Boehm showed to be undecidable [Boehm 85]. By placing some judicious restrictions on expressions, Milner and Hindley were able to support polymorphism while achieving decidable type inference. By not including explicit type variables in the programming language (only in the algebra of types), they sidestepped the problems later examined by Boehm. One restriction they impose is that the use of polymorphism is limited to the scope in which it is defined. Another is that self-application is not allowed. Though these seem to be fundamental limitations, the class of polymorphic functions which are allowed is quite substantial and sufficient to make the language quite useful.

Kanellakis and Mitchell [Kanellakis 89] have recently made the disquieting observations that:

- The length of the principal type expression for an ML expression of length *n* is doubly exponential in *n*.
- The unification of two type expressions for the ML core language is PSPACE hard.
- It is PSPACE hard to determine whether a given core ML expression is typable.

These observations contrast with the common belief, based on practical experience with ML type systems, that ML typing is efficient and does not slow down the process of compilation. We can only speculate that typical experience with ML type systems is with types of relatively small size or types not containing problematic constructs, so the combinatorial explosion is of limited scale.

A number of languages have been developed in the tradition of ML, among them HOPE [Burstall 80], Miranda [Turner 85], and Haskell [Hudak 90]. While ML contained some non-applicative constructs, all of these descendants are functional languages. A goal of HOPE was to consolidate a number of well-understood features of functional languages. In style and theory, it owes considerable debt to Landin and his ISWIM language [Landin 66]. HOPE requires type signature declarations on all function definitions, but allows polymorphism and does complete compile-time type checking. Its most significant new contribution was the capability for a user to declare his own concrete and abstract data types and to destructure them by pattern matching. This feature, not included in the original ML, was later incorporated into SML, or Standard ML [Milner 84]. Miranda is the most recent of a series of languages developed by David Turner. Preceded by SASL [Turner 76] and KRC [Turner 81] and inheriting many of their concepts, Miranda provides nice syntactic sugar for functional programming structures and emphasizes the use of higher order functions and lazy evaluation. It is the first commercially available functional programming system. Haskell is a general purpose functional programming language which attempts to consolidate the functional programming research of the 1980's. Additionally, it provides tools such as a module facility, a well defined I/O system, and a rich set of primitives to support larger scale programming projects. Haskell's approach to data abstraction treats algebraic specification and information hiding as orthogonal concepts.

ML-style type inference has also been adapted to existing programming languages. Mycroft and O'Keefe, for instance, developed a polymorphic type system for Prolog [Mycroft 84], and Dietrich and Hagl modified and extended that type system to handle explicit subtype relationships [Dietrich 88].

Meertens [Meertens 83] gives an interesting account of a type inference system employed to support a non-applicative language, B. The inference algorithm is essentially Milner's, but the organization of the algorithm is such that it supports type checking by increments corresponding to minor changes in the program. Meertens' paper provides an excellent detailed description of how Robinson's unification algorithm is employed[9], how mutually recursive definitions are treated, and how non-termination is detected. Meertens claims a stronger sense of completeness than that demonstrated by Damas and Milner, in that the B algorithm assigns types when the ML algorithm cannot, but this, he admits, can be attributed to the absence of infinite types and procedure parameters in B.

Although ML is not itself an applicative language, neither is it the first non-applicative language to be supported by automatic type inference. Henderson [Henderson 77], presented a method for type checking several imperative programming language constructs which bore a resemblence to the ML style of type checking. He specified a language which was free of type signatures and which supported procedure parameters as well as a small but interesting set of conventional program structuring operations. Then he gave type equations for the supported operations which restrict their operands and which could then be composed to determine the type consistency of a program in the language.

2.3.2 Variants on Classical Unification

The classical unification algorithm, which is sufficient to support ML type inference, does not support several data forms which are critical to the work on Common Lisp. Specifically, these are disjunction, negation, and recursive structures. These limitations have also been encountered by the computational linguistics community in pursuit of stronger grammar processors. Kay employed unification in his Unification Grammar [Kay 79] for processing feature structures, a common linguistic representation. Karttunen [Karttunen 84] observed that most grammar formalisms for features at that time had the same inability to deal with negation, disjunction, and cyclic structures in unification and generalization. In particular, he was concerned about negation and disjunction as being quite important to the task, and he proposed a method for supporting them using negative and positive constraints (without mentioning a solution for recursive forms). In doing so, he pointed out that performance problems persisted.

The complexity of unification with disjunction is indeed formidable. Kasper and Rounds [Kasper 86] [Rounds 86] presented a model for describing feature structures and used it to prove that the unification of disjunctive feature structures was an NP-complete problem. Kasper [Kasper 87] went on to provide an algorithm which performed reasonably well in the average case. He did this by canonicalizing the structures prior to unification, then tailoring the algorithm to consider the most inherently expensive cases only after other methods had been exhausted. Unfortunately, his canonicalization would not be possible on recursive forms containing disjunction.

Aiken and Wimmers [Aiken 92] place the problem of merging forms containing variables, conjunction, disjunction, and negation in a different setting, that of solving system of set constraints, and exhibit an algorithm for doing so.

---

[9]The unification algorithm cited is credited to Boyer and Moore [Boyer 72], who showed how unification could be performed more efficiently by using a stack of binding environments rather than by using explicit term substitution as formulated by Robinson.

2.3.3 Type Inference Treating Coercion and Inheritance, Object-Oriented Systems

When type coercions are added to the ML type system, the ML style type schemes are no longer adequate to characterize all the possible typings of a given term. Mitchell [Mitchell 84] addresses this special problem in a theoretical paper by adding to the system of type schemas a model of coercion based on set containment. His model is strong enough to support type inference in an ML-style language with automatic type coercion, but the algorithms for doing so are not given in the paper.

The approach to types in the object-oriented programming paradigm has evolved from notions of inheritance and abstract data types. Type inference has not received a great deal of attention in the object-oriented programming world, but in a recent paper, Wand [Wand 87] modeled hierarchical objects as extensible records[10] in an adaptation of an approach by Cardelli [Cardelli 84b, Cardelli 89]. Then, in the tradition of Milner [Milner 78] and Cardelli [Cardelli 84a], he reduced the type inference problem for his language to the unification problem.

Mishra and Reddy [Mishra 85] have a system whose capabilities are as similar to those of the work described in this report as any we have found. Their type language is quite similar to ours, using a "fix" form analogous to our *REC descriptor to represent recursive structures, and including type variables, conjunction, and disjunction. Their system also discovers undeclared types, unlike ML-derivative systems and existing Lisp type inference systems. Their type variables allow them to capture parametric polymorphism in the signatures for functions. But they do not capture the notion of *ad hoc* polymorphism, inherent in the nature of Lisp.

Fuh and Mishra [Fuh 88] present another type inference system which supports the notion of subtypes. They prove a principal type schema property within their type system and demonstrate a complete inference algorithm. One of the things that distinguishes their algorithm is that their MATCH algorithm, which is the counterpart to the unification algorithm in ML-style systems, derives a unifying substitution from a set of (subtyping) coercion relationships rather than from a set of equations expressing type constraints.

2.3.4 Ad Hoc Polymorphism and Overloading

Although overloading is not directly relevant to the Common Lisp work, *ad hoc* polymorphism and overloading are related concepts and are sometimes considered to be equivalent. Methods for dealing with overloading are an active area of research. There seems to be no single conventional way for treating overloading in type checking systems. Some efforts for bringing a more uniform discipline to overloading have focused on the commonality in the objects handled by the overloaded operator. Kaes [Kaes 88] proposed that all overloaded symbols be declared in advance, and that all symbols of the same name have a common polymorphic type. Wadler and Blott take a similar approach in the Haskell language [Wadler 89]. They propose the notion of type classes, which capture and characterize collections of overloaded operators in a consistent way, and require that any given symbol may belong to only one class.

Others have dealt with overloading and/or type parameterization by placing various restrictions on the language being supported. Cormack and Wright [Cormack 90] incorporate type inference, implicit parameter binding, and overload resolution to do type-dependent parameter inference. Although they handle higher-order polymorphic functions, they require all function abstractions to be explicitly typed, and type application is restricted to simple types. Boehm [Boehm 85] showed Reynolds' formalism of polymorphic functions to be problematic when the supplying of type parameters was optional, proving

[10]For a discussion of record type extensions, see Wirth [Wirth 88].

that the type inference problem is undecidable for a simple extension of the Girard-Reynolds polymorphically typed lambda calculus. In Boehm's extension, type bindings not specified are to be inferred by the type system. His characterization disallows the inference of type abstractions and requires that the positions of omitted type parameters be explicitly marked, and his proof of undecidability relies on these restrictions. In later work [Boehm 89], Boehm provided a set of rules which require that type parameters are inferred only when functions are applied (and not when functions are passed unapplied to higher order functions) and that functions are uncurried so that type parameters are applied in conjunction with other parameters which depend on them. O'Toole [O'Toole 89] outlines a language supporting explicit conversions between the explicit type abstractions of second-order polymorphic lambda calculus and ML-style polytypes. Where an explicit conversion is performed, type inference is done; elsewhere, type applications are required to be explicit. Ada [DoD 83] supports implicit operation parameters to generic functions and packages, but only after type parameters have been explicitly instantiated.

2.3.5 Other Efforts

After McCarthy's introduction of Lisp, which will be discussed later, the next major development in the functional programming paradigm was Peter Landin's work in the mid-1960's. He discussed how to mechanically evaluate lambda calculus expressions through an abstract machine called the SEDC machine [Landin 64] and formally defined a subset of ALGOL 60 in terms of the lambda calculus [Landin 65]. His ISWIM language [Landin 66] embodied a number of syntactic ideas (infix notation, LET and WHERE constructs, simultaneous and mutually recursive definitions, and the offside rule) and semantic principles, including referential transparency, which are significant in functional languages to this day.

One of the first block-structured programming languages with a significantly extended notion of type was EL1 [Wegbreit 74]. Types (or *modes* in EL1 terminology) were first class objects in the data space. The language supported mode-generating routines and generic routines. One of the kinds of mode was a union of modes. Functions whose parameters were of union mode were compiled so that each invocation "froze" the mode of its parameters for the duration of its activation. Another unusual facility was the ability for users to specify conversions among all modes. The type system of EL1 was quite adventurous for its day, so much so that it was viewed to be overly general and to provide the user with a bewildering excess of capability. Moreover, the flexibility of the type system has been alleged to defeat the goals of strong static type checking [Giannini 85].

John Backus' landmark 1978 Turing Award Lecture [Backus 78] provided much of the impetus for the development of functional programming. Backus' language, FP, was built upon a fixed set of combining forms called functional forms rather than upon the lambda calculus, which Backus claimed provided too much freedom for programmers' good. $\perp$ (bottom) is a valid FP object, so all FP functions can be said to be total, and FP did not originally support any type checking. But programmers usually wish to avoid $\perp$, and as a result commonly had to code explicit checks to validate the form of arguments. To alleviate the need for this tedium, Guttag, Horning, and Williams added a type system to FP [Guttag 81]. For each FP function, the type system ascribes a domain predicate and a target, or result, type. Target types are treated as "type transformers" in order to accommodate combination of the types of polymorphic functions. A function is said to be *fully typed* with respect to the type system if the domain predicate and the target type depend only on the type of the argument rather than on particular values within the type. Thus, while the plus function is fully typed, division is not, since the domain predicate depends not only on the type of the argument, but also on whether the second component of the argument is zero. A program is *type safe* if the domain predicate is T and *type erroneous* if the domain predicate is F. Division is neither type safe nor type erroneous; its type is data dependent and is safe for a set of data objects. The type system also supports user definition of abstract types. The definition includes a name for the type, a representation of a class of objects, an invariant over objects in the representation which captures the notion of a valid value, and a list of functions which implement the operations available on objects of the type. The representation is visible only to the implementation functions. Recognizer functions are defined implicitly, as well as functions for mapping abstract values to values in the representation and vice versa. Geoffrey Frank proposed an automatic type checking system for FP [Frank 81] which was to be built around a kernel algebra specified in his paper. The type checker would essentially do type inference by composing the domain and range specifications of functions called in the definition of a functional form.

Takuya Katayama [Katayama 84] proposed an approach to type inference for FP programs based on the notion that there exists a type domain which is a projection of the data domain in which we normally think of programs computing. Any computation in the data domain has an image, or *reduced computation* in the type domain. Katayama produces relations in the type domain as the images of functions and procedures in the data domain and then performs type inference with an algebra of relations. His approach was exploratory when published, particularly with respect to recursive functions. It appears to amount simply to a rephrasing of the same computations which occur in Milner-style type inference, though in very different terms.

Fairbairn's Ponder system [Fairbairn 82, Fairbairn 86] provides full type checking for a richer type system than that offered by Hindley-Milner. In particular, it supports polymorphic functional arguments[11] and a notion of domain strong enough to recognize within the type system, for example, that reciprocal works only on a non-zero argument. Although no primitive types are defined, type generators may be defined for the purpose of declaring types. Generators may be parameterized, providing a powerful tool for developing polymorphic types. A special generator *rectype* is employed for generating recursive types. Restrictions built into rectype ensure that all types declared with it are finite cycles rather than infinite trees.

Leivant applied the notions of type structure he formulated in [Leivant 83a] to the type inference problem, developing a general algebraic formulation of Milner's inference algorithm [Leivant 83b]. In this paper, he also formulated a different type inference algorithm which accommodated coercion and overloading and yet another which handled abstraction and general type quantification. The author knows of no instance where Leivant's work has been incorporated into any programming language or environment.

Abadi, Cardelli, Pierce and Plotkin [Abadi 89] developed a system which supports a type called *dynamic*, whose purpose is to deal with persistent objects existing outside the program being type checked. Their goal was to support maximal static type checking of programs, but to have a means within the same discipline for doing dynamic type checking of objects which are out of reach of the static type checker. Any such object is stored with a tag which is some representation of the type of the object. Objects thus tagged have the type "dynamic". After being read, they may be examined by a running program with a "typecase" form which case splits on the indicated type and controls flow accordingly. Their paper also presents some preliminary thoughts on dealing with polymorphism within the dynamic construct.

Another style of polymorphism is Coppo's conjunctive discipline [Coppo 80a, Coppo 80b]. It is powerful enough to allow the characterization of various classes of lambda-calculus terms, but is impractical for programming languages because type checking is unsolvable.

---

[11] For example, a Ponder function could have a formal parameter whose type is ($\forall$ T. T -> T), so that the actual parameter supplied could be a function whose type is T_1 -> T_1.

2.4 Type Reasoning for Lisp

Lisp has remained by far the most popular language with a largely applicative basis, particularly in the western hemisphere. Emerging from its roots as an exploratory language for non-numeric computation [McCarthy 62] used primarily in artificial intelligence and theorem proving, it has found significant application in such disparate areas as systems programming [Symbolics 88], numerical applications [Mathlab 82, Mathlab 83], text editing [Stallman 77], and image processing, and has recently been enlarged to support object-oriented programming [Cointe 88, Steele 90]. Plagued for years by a lack of standardization which discouraged software portability, the Lisp community has consolidated itself in support of Common Lisp [Steele 84, Steele 90]. The success of this effort and the continued development and support of Lisp-based products and research seems to guarantee the language's continued popularity, despite claims that newer languages rest on more sophisticated and attractive foundations and impose greater discipline.

Cartwright's thesis [Cartwright 76] was an early attempt to implement a strong static type checker for Lisp. Not surprisingly, Cartwright restricted his attention to a small subset of Lisp with functional semantics. Moreover, some primitives within his subset had more restrictive definitions than their Lisp counterparts. His base subset contained no primitive constructors (not even CONS), but his type system provided a disciplined facility for declaring constructions, thereby allowing users to build up various abstractions. His type system was essentially the one proposed by McCarthy [McCarthy 63]. Cartwright required all identifiers and functions to have user-furnished type signatures, much like any ALGOL-like language. In its original formulation, his type system would have been statically strong, but he found this too restrictive and relaxed his static checks, thus requiring run-time checks when static checks are not satisfied. He attempted no type inference and incorporated no notion of polymorphism.

Boyer and Moore have developed automated support tools for formal reasoning about Lisp-like logics. Their Computational Logic [Boyer 75, Boyer & Moore 79, Boyer & Moore 88] bears a strong resemblance to an applicative Common Lisp. In its original form, it allowed definition of only total functions, but with the addition of V&C$, the logic allows the definition of non-terminating functions. Within the logic, there is a mechanism, the shell principle, for declaring new classes of recursively structured objects. Primitive types such as natural numbers, lists, and litatoms (words) are defined as shells in the base logic. The theorem prover which manipulates the logic, Nqthm, does some limited type inference, but only in terms of known shell "types". The logic allows for arbitrary CONS structures, but does not provide specific type support for them.

Boyer and Moore are carrying many of their methods into a new logic and theorem prover, Acl2 [Boyer 90], which is under development. In many ways Acl2 resembles Nqthm, but the logic (and the implementation language) is an applicative subset of Common Lisp. Guards on function parameters, reminiscent of those on Verdi functions in the EVES environment [Pase 89] address the partial nature of many Lisp functions. If a program passes the Acl2 admissibility tests and if all its guard predicates are verified on all function calls, the program may be safely compiled by a standard Lisp compiler with a low safety setting, thus producing fast code without fear of breakage due to type errors. But type reasoning in Acl2 is only slightly more powerful than in Nqthm, and the need to verify that all function calls satisfy the guards of the called functions puts much greater pressure on the type system.

McPhee [McPhee 89] did a preliminary investigation of type inference for Lisp, which led to the implementation of a simple tool in Miranda. But the small set of Lisp functions which were supported, including CONS and IF, were assumed to obey the same restrictions as their Miranda counterparts, and thus his work did not break any new ground.

Kaplan and Ullman [Kaplan 80] used a lattice-theoretic model of types and developed a flowgraph model of programs to support type inference for dynamically type-checked languages. A flowgraph is a directed graph whose nodes are associated with one or more assignment statements. The types of all expressions in a program are analyzed by repeated application of "forward" and "backward" analysis with respect to pre-defined lattices of types until a fixed point is reached for all type assignments. Forward analysis uses the types of subexpressions to help determine the type of an expression. Backward analysis uses context information to help determine the type of an expression, i.e., knowledge of the type of an expression is used to refine the types of its subexpressions.

This approach is largely the basis of the TICL type inference system for Common Lisp [Ma 90]. The system uses repeated application of forward analysis with respect to a collection of partial type lattices for Common Lisp types to derive type annotations for use by the compiler. Since the type lattices must be fixed, user types such as those declared with DEFSTRUCT may not be included. The system, however, does include an *ad hoc* type LIST-OF to provide some support for a particular kind of CONS structure. Type inconsistency is resolved by assignment of a most general type, enabling the inference process to continue. Since the goal of the system is to improve run-time performance, the type inference system is tailored to process only types for which declarations offer significant speed-up for most compilers. Thus, list-processing operations and general CONS structures are largely ignored in favor of accurate assignments of various types, like the numerics, likely to be specifically supported at the machine level.

Kaplan and Ullman’s work is also the basis of the Nimble type inference system for Common Lisp [Baker 90]. As with TICL, the type lattices it employs are primarily focused on numeric types, and the system does not appear to deal with structured types in any direct sense. Both the Nimble and the TICL systems achieve their pragmatic goals of improving run-time performance of compiled Lisp code.

Baker has also developed a decision procedure for the Common Lisp SUBTYPEP predicate [Baker 92], utilizing a specific lattice structure suitable for performing type inference. A complete implementation of SUBTYPEP can assist a compiler in making optimal storage allocation decisions and in removing some type-checking code.

A problem limits the effectiveness of the lattice-theoretic models when confronting *ad hoc* polymorphic functions. When a form can return disparate types, the type inferred for the form is the least upper bound of all the possibilities. Commonly, this forces the assignment to the universal type, which is of no use except to enable the inference process to continue. Disjunctive types would provide greater specificity, but these are not compatible with the lattice approach in systems styled after Kaplan and Ullman.

Johnson [Johnson] uses a technique he calls "type flow analysis" to represent the type-level structure of Lisp programs. Though his report does not present much technical detail, he does claim to deal with the problems of *ad hoc* polymorphism and heterogeneous structures.

2.5 Static Type Checking and Optimizing Compilers

The fundamental relationship between static type checking and the generation of fast executable code for programs is simple: checks that can be performed statically need not be performed at runtime. A program running without dynamic type checking code will do its job "faster" than an otherwise identical program executing those checks. This relationship is a significant motivation for most of the static checks which compilers perform.

Efforts employing mechanically assisted deduction systems and verification to enable compilers to eliminate dynamic type checking code date back to the Euclid project [Lampson 76]. As designed, Euclid's approach is that a legal program is not allowed to have run-time exceptions. The compiler has the responsibility of proving that exceptions cannot occur. If it fails, it is obliged to create a *legality assertion*, which can be submitted to proof by other static means or be checked at run-time. If an unvalidated assertion is false at run-time, the program is by definition not a legal program.[12]

Implied, but not directly expressed in Cartwright's thesis on Typed Lisp [Cartwright 76] is the notion that where his static type system is able to determine type correctness, the compiler does not need to generate type checking code. Cartwright alluded to this notion, though, by noting that when he relaxed the rules in his static type checker, he created the requirement that "in an actual TYPED LISP implementation, run-time type checking should be done in all those cases where standard parse-time rules are violated. If the user formally proves that a certain run-time error can never occur, then that particular check can be safely eliminated."

In his Phd thesis, McHugh [McHugh 83] explored the application of formal reasoning to compiled code optimization in Gypsy, particularly with respect to the runtime detection of type errors. [Good 86a]. He modified the verification condition generation process in the Gypsy Verification Environment [Good 86b, Young 90] to generate theorems which, if proved, guaranteed the absense of certain run-time type errors. The fact that these theorems were proved was then communicated to the compiler [SmithL 80] which then suppressed the generation of run-time error checking code. Unfortunately, the Gypsy Verification Environment no longer supports a compiler, although the generation of verification conditions for proving the absence of type errors has been re-implemented and preserved [Akers 86].

Boyer and Moore's work with Acl2, previously mentioned, falls squarely into this realm as well. They use the mechanical assistance of their theorem prover to validate conformance to domain restrictions expressed as guards. Accepted functions may then be compiled at a low safety setting, i.e., without runtime type checks. Unfortunately, the assistance provided in their early developmental system was not sufficient to provide this validation automatically in many routine situations.

A different approach to efficient run-time type checking is used in the Symbolics Lisp machines. [Moon 85] The system employs a hardware co-processor to perform run-time type checking at the same time it is doing data operations, so the main processor incurs minimal extra run-time overhead.

---

[12]This notion of illegal programs is somewhat distasteful, as the compiler abdicates on assigning a meaning to a program which it compiles without complaint. A similar notion pervades the definitions of Ada [DoD 83] and Common Lisp [Steele 90].

Chapter 3

OVERVIEW

The purpose of this chapter is to provide an overview of the entire type inference system. We will first provide some orientation to function signatures and the type inference process. Then, in a bit more detail, we will examine each of the principal constituents of the system and see how they fit together. This should provide the reader enough background and perspective to approach the chapters that follow, which cover the formal theory and each of the system's components in great detail.

3.1 The Lisp Dialect

The subset of Lisp which is supported by the inference system is quite simple. It is limited to functions of fixed arity, where the only forms allowed in the body are:

- references to formal parameters,
- literals of type character, integer, rational, T, NIL, or string,
- quoted forms, where the quoted object is either a literal, a symbol, or a CONS of any such objects (recursively),
- IF forms of arity three, and
- recursive function calls or calls to previously accepted functions.

All functions are assumed to terminate for all inputs, and all data objects are of finite size, i.e., there can be no circularly linked lists. Mutually recursive functions are not allowed. Function names are not allowed to begin with the characters "$", "*", "%", or "!".

In the initial system state, only a small set of native functions are defined. These functions are CONS, CAR, CDR, BINARY-+, UNARY--, BINARY-*, UNARY-/, <, EQUAL, CONSP, INTEGERP, RATIONALP, STRINGP, CHARACTERP, SYMBOLP, NULL, DENOMINATOR, NUMERATOR, SYMBOL-NAME, and SYMBOL-PACKAGE-NAME. Most of these are native Lisp functions. BINARY-+, UNARY--, BINARY-*, and UNARY-/ are just the binary and unary versions of the n-ary Lisp functions +, -, *, and /. We call our native functions *subrs*.

All functions are purely applicative. There is no global data. Since there are no "destructive" functions, i.e., functions like RPLACA which destructively alter data, no data structure can be created where a pointer points to a CONS cell higher in the structure. Thus, no CONS structure contains an infinite chain of pointers.

Type restrictions are present in the domain of many Lisp functions. CAR, for instance, requires its argument to be a CONS or NIL, and application of CAR to any non-NIL atom will cause a runtime error.

Type restrictions, if analyzed statically, tend to percolate through an application. To characterize them and to allow for restricting the domain of a function, we augment the Common Lisp subset by allowing a guard to be included as part of a function definition.

Definition: A *guard* is a component of a function definition which is a Lisp predicate on the function arguments. The guard must be *satisfied*, i.e., it must evaluate to a non-NIL value, in order for the function to be well-defined on those arguments. Syntactically, it follows the formal argument list, and is of the form of a Lisp compiler directive:

(DECLARE (XARGS :GUARD <form>))[13]

A guard may not call the function within which it is defined. If no guard is declared with a function, the default guard T is used.

Guards can characterize type restrictions on the parameters which will guarantee the absence of type errors in the function body. Properly employed, they can be of great utility in conjunction with the inference system. When the type inference algorithm analyzes the body of a function, it assumes the guard is satisfied and makes judgements about the types of the arguments accordingly.

3.2 Type Checking and Function Signatures

The primary task of the inference system is to infer a type signature for a function from the function text and from a database of signatures for previously defined functions. Previously defined function signatures allow the type checking process to scale up as new functions are introduced. The signatures are in terms of type descriptors, and manipulation of type descriptors is the primary occupation of the inference system. The grammar defining the language of type descriptors used in our signatures is as follows:

```
<descriptor> ::= <simple descriptor>  |  <variable>  |  *EMPTY  |
                 *UNIVERSAL | (*CONS <descriptor> <descriptor> )  |
                 (*OR <descriptor>* )  | <rec descriptor>

<simple descriptor> ::=
    $CHARACTER | $INTEGER | $NIL | $NON-INTEGER-RATIONAL |
    $NON-T-NIL-SYMBOL | $STRING | $T

<rec descriptor> ::= (*REC <rec name> <recur descriptor> )

<rec name> ::= a symbol whose first character is not "&"

<variable> ::= a symbol whose first character is "&"

<recur descriptor> ::=
   <simple descriptor>  |  <variable>  |  *EMPTY  |
   *UNIVERSAL | (*CONS <recur descriptor> <recur descriptor> )  |
   (*OR <recur descriptor>* )  | <rec descriptor> |
   (*RECUR <rec name>)

<dlist descriptor> ::= (*DLIST <descriptor>*)

Note: *DLIST is simply a notation for packaging a list of simple
      descriptors into a single form.
```

---

[13] This syntax is identical to that used in Boyer and Moore's Acl2 system [Boyer 90].

As stated in the introduction, a signature has two primary components,[14] a *guard descriptor*, which is a vector of variable-free descriptors characterizing the guard, and a collection of *segments* mapping parameter types to result types. The guard descriptor has the same length as the arity of the function, and each of its descriptors characterize the type requirements placed on the corresponding argument. Each segment in the signature is of the form

(td$_1$ .. td$_n$) -> td

where n is the arity of the function. We interpret a segment to mean that if the descriptors (td$_1$ .. td$_n$) characterize the function's arguments, then the result may be (but is not necessarily) characterized by the descriptor td. The meaning of the signature as a whole is that if the function's guard, evaluated on some arguments, yields a non-NIL value, then there is some segment in the signature which characterizes both the function's arguments and the value produced by evaluating the function on those arguments.

An important notion pertaining to a signature is that of completeness of its guard descriptors. A guard descriptor exists that is *complete* with respect to a guard if the guard is expressed as a conjunction of recognizer calls on distinct formal parameters. When a guard descriptor is complete, we will show it is satisfied by the actual parameters if and only if the guard expression evaluates to a non-NIL value on the parameters. When a function's guard is complete, and the guards of all the functions in its hereditary call tree are also complete, then we will show that if the actual parameters satisfy the guard descriptors, they will also satisfy the real guard (i.e., cause it to evaluate to a non-NIL value), and the evaluation of the function call will not cause a guard violation.

Here is the signature for the CAR function.

```
Function: CAR
Guard computed by the tool:
   ((*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*CONS (*FREE-TYPE-VAR 1.) *UNIVERSAL)) -> (*FREE-TYPE-VAR 1.))
 (($NIL) -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T
```

CAR takes a single argument, so its guard is a singleton vector. The guard descriptor requires the argument to be either the value NIL or a CONS of any two objects. (*UNIVERSAL is a descriptor satisfied by any value whatsoever.) There are two segments. The first says that, given an argument which is a CONS, the function may return the same value which was the first element of the CONS. The second says that, given NIL, the function may return NIL. When we say the guard is complete, we mean the guard descriptors completely capture the actual guard predicate, which in this case is (where X is the formal parameter name):

```
(CAR-ABLE X)
```

```
where (DEFUN CAR-ABLE (Y) (IF (NULL Y) T (CONSP Y)))
```

---

[14] It has other components which we will discuss later.

An example of a guard which is not complete is (EQUAL X 3), since the corresponding descriptor vector would be ($INTEGER), but some values satisfying this vector (4, for instance) would not satisfy the guard. When we say "All called functions complete", we mean that the guards of every function in the call tree of the definition are complete.[15] When we say the recognizer descriptor is NIL, we mean this function is not a recognizer. If it were, this entry would contain the descriptor which characterizes the objects for which the function returns the value T. The other elements of this signature we can disregard for now. We will explore all these notions much more fully in later chapters.

The initial state of the system includes the signatures for all the native functions. The signatures for these functions are given in Appendix A.

In the process of inferring a signature for a new function, the inference tool attempts to validate that the guards of all called functions are satisfied, at least within the scope of the type system. This is both a means and an end for the system. It is a means because the tool makes a conservative judgement, when it cannot demonstrate that the guard descriptors are satisfied, that the real guard expression will not be satisfied either. If the real guard is not satisfied, there is no guarantee that the use of the signature is sound. Thus, when the tool fails to show the guard descriptors are satisfied, it declares failure and aborts the analysis. But guard verification is also a goal for the system, because in the cases where the guard descriptors are known to be complete for a function, then the tool can guarantee that if it does not detect a guard violation on a function call, the real guard will be satisfied by any possible parameters. Furthermore, if the guard descriptors are also complete for all the functions in the call tree of the called function, then the tool's failure to detect a guard violation signifies that the guards for all functions called in the course of evaluating the original function call will also be satisfied.

3.3 Recognizer Functions

A *recognizer* function is a Boolean function of one argument and of a certain form which determines whether its argument conforms to a variable-free type descriptor. Conversely, for any variable-free type descriptor, one could define a corresponding recognizer function. A recognizer must have no guard (or a vacuous guard, T), so it is defined for all actual parameter values.

One reason recognizers are important to the type system is that if a call to a recognizer is used as a test in an IF expression, optimal modifications to the type context of the IF may be made to support the analysis of the THEN and the ELSE arms. Another reason is that, if a function guard is expressed as a conjunction of recognizer calls on distinct parameters, we know that the guard descriptors are complete. The entire inference system is monotonically conservative, so the descriptors computed to characterize the value of an expression may be too inclusive, but they can never be too small. In the case of a complete guard (or any complete type predicate), the fit is perfect. The significance of a complete guard is that a list of actual parameter values satisfies a complete guard expression if and only if they satisfy the guard descriptors computed by the inference system. If the system determines that the (possibly too inclusive) descriptors characterizing the actual parameters represent a subset of the values represented by the guard descriptors, then the real function guard will evaluate to a non-NIL value.

Given a new function, if the inference algorithm determines the function is a recognizer, it will employ special techniques to infer its signature, techniques which will lead to the perfect signature we seek.

---

[15]There are two pairs of completeness tags, the latter preceded with the notation "TC". We shall see later that the first set of tags signifies a mere conjecture, but the TC tags, more stringently defined, carry a guarantee with respect to the evaluation of guard expressions.

3.4 The Inference Algorithm

The inference system operates in two phases when a new function is submitted. First, the inference algorithm makes a good heuristic guess at the signature. Then, the checker algorithm, which is formally specified and proven sound, validates the signature.

The system makes certain assumptions about functions being submitted to it, mainly that they lie within the language subset supported by the system, that the only non-recursive function calls within the function definition are to functions already in the system database, and that, given any arguments satisfying the function guard, the evaluation of the function body will terminate.

There are several significant algorithms within the main inference algorithm. TYPE-PREDICATE-P takes a form, determines if it is a type predicate, and if so, returns (among other things) a vector of descriptors characterizing what can be deduced about the types of the variables in the environment when the predicate evaluates to a non-NIL value, and whether any information is available when the predicate evaluates to NIL. DESCRIPTOR-FROM-FNDEF takes a function, determines if it is a recognizer function, and if so, returns the descriptor which characterizes the objects for which the function returns T. The composition of DERIVE-EQUATIONS and SOLVE-EQUATIONS, given a form, returns a guard and the segments which become the main elements of its signature.

When a new function is submitted to the inference algorithm, the following events transpire. If there is no guard, or if the guard is T, DESCRIPTOR-FROM-FNDEF examines the function to determine if it is a recognizer. If so, it generates what we hope is a perfect signature for the function, meaning that there are two segments, one whose result descriptor is $T and the other $NIL, where the respective argument descriptors characterize no values in common, but together characterize the entire universe of values.

If there is a guard, it is first checked to ensure that it does not call the function recursively. Then, the inference algorithm invokes DERIVE-EQUATIONS and SOLVE-EQUATIONS on the guard form to ensure there are no detectable guard violations. If there are, the entire analysis fails, and the function is rejected. If none are found, then TYPE-PREDICATE-P is invoked to construct the guard vector for the function.

Then, using this guard vector as an assumption on the types of the function arguments (or using a guard vector composed of *UNIVERSAL descriptors if there was no guard form), we invoke DERIVE-EQUATIONS and SOLVE-EQUATIONS on the function body to generate the function signature and, while in the course of doing so, attempt to validate all the guards of functions called within the body. If a guard violation is detected, the function is rejected.

If the function is not rejected, SOLVE-EQUATIONS returns a signature for it. But the algorithms which generated the signature are not trusted algorithms. They have not been formally modelled at all, much less shown to be consistent with the semantics of the type system. So we can think of them as heuristic algorithms which have suggested a signature for the function which must now be validated in order to be trusted. This validation is performed by the signature checker, which has been rigorously formalized and proven to be sound with respect to the formal semantics of the type system.

The inference algorithm just outlined is discussed in much greater detail in Chapter 4.

3.5 The Formal Semantics

We have spoken frequently of values "satisfying" descriptors. This is the fundamental notion in the formal semantics. Roughly speaking, a value satisfies a descriptor if it is a member of the set of values represented by the descriptor. But with type variables in the descriptor language, a descriptor in isolation cannot necessarily prescribe a set.

The notion of type variables employed in the formal semantics is that a variable represents a single arbitrary Lisp value. While this is not the conventional notion of type variable appearing in most type systems, it is perfectly suited to deriving maximal information from the signatures of certain very important functions, like CAR, CDR, and CONS, whose argument values (or specific components thereof) appear intact in the result. Thus, the following segment, from the signature of CAR,

```
(((*CONS &1 *UNIVERSAL)) -> &1)
```

signifies that the very value which is the first element of a CONS actual parameter appears as the result value.

When we speak of a value satisfying a descriptor containing type variables, we mean the value is a member of the set prescribed by the descriptor under a binding of each of its type variables to particular Lisp values. The set prescribed by the descriptor (*CONS &1 $INTEGER) under the binding ((&1. 10)) is the set of all CONS values whose CAR is the value 10 and whose CDR is any integer.

Variables are only interesting when the same variable appears more than once in a descriptor or descriptor list, as in the descriptor list formed by the above-mentioned segment for CAR. In order for a list of values to satisfy such a list of descriptors, the very same value must appear in every position occupied by a given type variable. So the value list (("foo" . 0) "foo") satisfies the descriptor list

```
((*CONS &1 *UNIVERSAL) &1)
```

whereas the value list (("foo" . 0) "bar") does not.

A consequence of this interpretation of variables is that they are of very limited use inside *REC descriptors. The situation suggested by a descriptor like

```
(*REC FOO (*OR $NIL (*CONS &1 (*RECUR FOO))))
```

is that we have a list, all of whose values are identical. We believe such structures occur infrequently, and the formal semantics does not support placing type variables in positions where the variable would appear replicated within a structure if the descriptor were opened up several times. A different kind of variable would be appropriate here, one which could be instantiated with an arbitrary descriptor, but work on the system has not yet progressed to the point of implementing such a thing. This could, in fact, be a very useful extension. On the other hand, under our semantic model, there is no reason not to allow type variables to occur in a non-replicating position of a *REC descriptor, for example

```
(*REC BAR (*OR &2 (*CONS $INTEGER (*RECUR BAR))))
```

The formal semantics are captured by a recursive function INTERP-SIMPLE, which takes as parameters a list of descriptors, a list of values of the same length as the descriptor list, and a binding that maps all the type variables in the descriptor to Lisp values. It returns T or NIL, depending on whether the values satisfy the descriptors under the binding. This function is presented and discussed in detail in Chapter 5, along with a discussion motivating this approach.

The semantics of a function signature is captured in the following definitions, which formulate the key soundness criterion for every signature in the system database. This is what the signature checker attempts to verify for every signature emitted by the inference algorithm. Here, "I" is an abbreviation for INTERP-SIMPLE. "E" is a Lisp evaluator which is defined in Section 5.4.

**Definition: A *world* is a collection of Lisp function definitions
including all the functions called within any of the definitions.**

Definition: foo^{world,clock}

This is a notational convention. Where arg_1 .. arg_n
are Lisp values and a_1 .. a_n are the formal
parameters of foo, by
foo^{world,clock}(arg_1 .. arg_n)
we denote
(E (foo a_1 .. a_n)
((a_1 . arg_1) .. (a_n . arg_n))
world
clock)

Definition: GOOD-SIGNATUREP (FNDEF GUARD SEGMENTS WORLD CLOCK)

For any function foo of arity n in our Lisp subset, whose
definition is denoted

(defun foo (a_1 .. a_n)
(declare (xargs :guard guard-form))
body)

and whose SEGMENTS are denoted
(($td_{1,1}$ .. $td_{1,n}$) -> td_1) .. (($td_{m,1}$ .. $td_{m,n}$) -> td_m),
for any non-negative integer clock and world containing the
above definition of foo,

(good-signaturep foo guard segments world clock)
=
(and
(well-formed-signature-1 guard segments n)
for any Lisp values arg_1 .. arg_n,
(and
H1 (not (break-out-of-timep (foo^{world,clock}(arg_1 .. arg_n))))
H2 (not (null (E guard-form ((a_1 . arg_1) .. (a_n . arg_n)) world clock))))
=>
(and
C1 (not (break-guard-violationp (foo^{world,clock}(arg_1 .. arg_n))))
C2 for some k in [1..m]
for some binding b of type variables to Lisp values
covering $td_{k,1}$ .. $td_{k,n}$ and td_k
(I ($td_{k,1}$ .. $td_{k,n}$ td_k)
(arg_1 .. arg_n (foo^{world,clock}(arg_1 .. arg_n)))
b)))

WELL-FORMED-SIGNATURE-1 just checks that arities are consistent, checks the well-formedness of all the descriptors, and checks that the guard descriptors are variable-free. It is defined in Appendix G.2. BREAK-OUT-OF-TIMEP and BREAK-GUARD-VIOLATIONP are defined along with E in Section 5.4.

Definition: VALID-FS (FS WORLD CLOCK)

```
(valid-fs fs world clock)
 =
For every signature (guard segments) in fs corresponding to a
function foo of arity n in world, whose definition is denoted
 (defun foo (a1 .. an)
   (declare (xargs :guard guard-form))
   body),

 (and (tc-all-called-functions-complete guard-form fs)
      (tc-all-called-functions-complete body fs))
 =>
 (good-signaturep foo guard segments world clock)
```

TC-ALL-CALLED-FUNCTIONS-COMPLETE checks the guard completeness property for all the functions in the form and in the hereditary call tree of each such function.

Paraphrased, VALID-FS says that a signature in fs is valid when, if the guards are complete for all the functions in the call tree of foo, if the guard evaluates to a non-NIL value when applied to some actual parameter values, and if clock is large enough to allow complete evaluation of the function call, then evaluating the function on those parameters will not result in a guard violation, and under some binding of type variables, the parameter values and the function result satisfy some segment in the signature. The predicate VALID-FS, which is a precondition of many of the most significant lemmas we will consider, is the iteration of this validity determination over all the functions in the database.

This seems a good place for a remark about the convention we use for the character case of identifiers in this document. In general, one may view the entire document as being case insensitive (with the obvious exception of characters within Lisp strings). For reasons having solely to do with the aesthetics of the type faces used here, we often use lower-case in text set apart as examples, as in the definitions just above. Upper case, however, is sometimes used in examples when depicting output from the system. Within normal prose paragraphs, we will use upper case for functions defined in the system and for forms generated by it, to set them apart from normal text.

Another convention used throughout the document is the omission of the Lisp quote character "’" from forms which are obviously quoted forms. In the few instances where this could lead to any misunderstanding, we take care to explain exactly what forms are being handled.

3.6 The Signature Checker

The job of the signature checker is to validate each signature emitted by the inference algorithm. For a recursive function, one may think of each iteration of the inference algorithm as producing a signature which is a better approximation of the function graph than that produced by the previous iteration. In this sense, the main inference algorithm produces a signature by iterating n times its process of passing over the function body and producing segments, and the checker produces the next approximation by making the n+1st traversal over the body. If the validation succeeds, the checker’s signature is a better approximation of the function graph, but we prefer to store and use the inference algorithm signature because it is almost always in a much more compact and efficient form, and any improvement in accuracy embodied in the checker signature is usually marginal.

Since the checker is to be validated with a proof of soundness, it is important that it be simple enough for the proof to be tractable, and in fact it is much simpler in concept than the inference algorithm. This simplicity is possible largely because the checker assumes that the signature it is given is correct, in that it is sound with respect to the semantics of the type system. Descriptor negation, represented explicitly with a *NOT form in the descriptor language supported in the inference algorithm, has been completely factored out in the final signature, so the checker need not deal with *NOT forms at all. Closed *REC forms have already been derived and constructed from analysis of the recursion in the function text, relieving a substantial burden on the checker. Moreover, the checker makes only minimal efforts to produce concise results. This allows a combinatoric explosion in the number of segments produced and impairs the runtime performance of the checker. But the tricky simplifications necessary to maintain a minimal representation would have required tricky formal justifications and would have made the checker signature less amenable to formal validation. The benefit of a clean formal algorithm clearly outweighs the desire for performance. Though the checker is by no means a simple algorithm, the semantics of its operations are clean enough to be tractable in proof. Furthermore, the performance penalty occurs only once for any given function, because the checker is not iterative and because the segments it generates are discarded after the validation process is complete.

It may seem paradoxical that the checker can derive a trusted signature, when one of its chief assumptions is the validity of the signature produced by the inference system, which we do not trust in any formal sense. But a formal validation of this approach is presented in Section 7.2. The key to this approach is a notion of descriptor *containment*. Descriptor containment is basically a subset notion, augmented to accommodate the presence of type variables. If the checker accepts a heuristic guess at a signature and then uses a trusted algorithm to produce a new signature, and if it can use a trusted algorithm to demonstrate that each segment of the new signature is contained in some segment of the original, then we can show that the original, heuristically derived signature is valid.

In the course of its analysis, the checker attempts to perform guard verification on function calls.

Definition: *Guard verification* **on a function call is the determination that the function’s guard will be satisfied for any possible actual parameter values.**

The checker’s strategy is to determine whether the actual parameter descriptors are contained in the guard descriptors for the called function. If it cannot do so, it signals an error and aborts the computation, since to continue would be to proceed on an unvalidated assumption. Containment by itself does not guarantee guard satisfaction in the general case, since the guard descriptor may be more inclusive than the guard itself. But a finding of containment is sufficient to proceed with the analysis. But if a given guard descriptor is complete, i.e., if the guard is a composition of recognizer function calls on distinct parameters, the checker’s guard validation does guarantee that the real guard will be satisfied, and then we say the guard is verified. Furthermore, if all the guard descriptors in the hereditary call tree of the called function are also complete, then the checker’s verification of the called function’s guard is sufficient to demonstrate that the guards will be satisfied for all the functions called in the course of evaluation of the call being checked.

Though the inference system will attempt to generate a signature even when this completeness property does not hold, no claims are made that the resulting signature is valid. This is because the use of the signatures of subsidiary functions is sound only when the satisfaction of their guards is known, and in the absence of complete guards, the inference system by itself cannot make this determination.

A detailed discussion of the signature checker appears in Chapter 6.

The inference system, including both the type inference algorithm and the checker, is implemented in Common Lisp and runs on the Symbolics [Symbolics 88] and on Sun workstations under akcl [Yuasa 85]. Aside from a very top-level function, which stores accumulated results in the database and accepts new definitions, the implementation is in the form of a composition of purely applicative functions.[16]

3.6.1 The Proof of Soundness

There are two top level goals which must be proven to demonstrate the soundness of the inference system. TC-SIGNATURE is the top level function of the checker, and the first and most significant is Lemma TC-SIGNATURE-OK, the conjecture that a signature validated by the checker is indeed correct with respect to the formal semantics of the type system.

```
Lemma TC-SIGNATURE-OK

 For any n-ary function foo, whose definition is of the form
  (defun foo (a1 .. an)
    (declare (xargs :guard guard-form))
    body)
  where guard-form is a conjunction of recognizer calls on distinct
  formal parameters,
 for any world of Lisp functions world, including at least the above
  definition of foo and the definitions of all the functions in the
  call tree of foo,
 for any list of function signatures fs, including signatures for
  at least all the functions in the call tree of foo, except foo
  itself,
 for any non-negative integer clock,
 when (tc-signature foo fs) successfully validates a signature
  for foo,

H1  (and (valid-fs fs world clock)
H2       (and (not (equal (guard (tc-signature foo fs))
                          *guard-violation))
              (not (equal (segments (tc-signature foo fs))
                          *guard-violation)))
H3       (tc-all-called-functions-complete guard-form fs)
H4       (tc-all-called-functions-complete+ body fs foo t) )
    =>
    (valid-fs (cons (tc-signature foo fs) fs) world clock)
```

Paraphrased, it states that if the database of function signatures is sound, if the computation of a signature did not abort with a *GUARD-VIOLATION result, and if the guards for all the functions in the call tree of foo are complete, then the checker adds a valid signature for foo to the database, in the sense of validity previously explained. This lemma is the top level goal of the large proof which is described in most of Chapter 7 and Appendix B.

The other principal lemma justifies our claim regarding guard verification. If a guard descriptor is complete, i.e., if the guard is a conjunction of recognizer calls on distinct formal parameters, and if the actual parameters of a function call satisfy descriptors which are contained in the guard descriptors, then we know the guard expression evaluated on those parameters will yield a value of T.

```
Lemma GUARD-COMPLETE

 Given a function of arity n with argument list (a1 .. an),
 and guard expression of the form
  (and (Ra1 a1) .. (Ran an))
 denoting a conjunction of calls to recognizer functions on distinct
 formal parameters, and where the recognizer function Rak
```

[16]Actually, this is a not perfectly accurate. There are three special variables which are used as counters for generating unique gensyms. The counters are updated as side effects of calling the gensym routines. There is no technical reason why any of these counters need to be implemented non-applicatively. They were implemented as specials only for expediency.

```
 has the segment (rtdk) -> $t,
 for any values arg1 .. argn, descriptors argtd1 .. argtdn,
 type variable binding b, non-negative integer clock, and a world of
 Lisp functions including all those in the call tree of the guard
 expression,

  (and
H1  (valid-fs fs world clock)
H2  (I (argtd1 .. argtdn) (arg1 .. argn) b)
H3  (contained-in-interface (*dlist argtd1 .. argtdn)
                            (*dlist rtd1 .. rtdn))
H4  (not (break-out-of-timep
           (E (and (Ra1 a1) .. (Ran an))
              ((a1 . arg1) .. (an . argn))
              world
              clock))) )
  =>
  (equal (E (and (Ra1 a1) .. (Ran an))
            ((a1 . arg1) .. (an . argn))
            world
            clock)
         t)

 Note: For the sake of uniformity in notation, let us say that
 there is one recognizer call for each parameter, where for
 parameters which are unrestricted in the guard expression we
 use a recognizer (DEFUN UNIVERSALP (X) T) whose segments
 are ((*universal) -> $t) and ((*empty) -> $nil).  In
 any real guard, any such recognizer call may be omitted from
 the guard without the loss of generality of this lemma.
```

CONTAINED-IN-INTERFACE is the top level function of the containment algorithm, which is presented in Section 6.8. We present the proof of Lemma GUARD-COMPLETE in Section 7.3.1.

3.7 Two Simple Examples

Consider the Lisp function:

```
(DEFUN SYMBOL-LISTP (X)
  (IF (CONSP X)
      (IF (SYMBOLP (CAR X))
          (SYMBOL-LISTP (CDR X))
          NIL)
      (NULL X)))
```

Recall that SYMBOLP is a recognizer function which returns T when given a symbol, NIL otherwise. The descriptor characterizing a symbol is (*OR $NIL $NON-T-NIL-SYMBOL $T).

Since there is one parameter and no guard form, the guard descriptor is (*UNIVERSAL), which is, of course, complete. No guard violations are detected, since CONSP, SYMBOLP, SYMBOL-ALISTP, and NULL all have (*UNIVERSAL) guards, and the calls to CAR and CDR are protected by the IF test which ensures their argument is a CONS. DESCRIPTOR-FROM-FNDEF determines that the body of the function satisfies the requirements of a recognizer function, and so it returns the descriptor that characterizes the values for which the function returns a T result:

```
(*REC SYMBOL-LISTP
      (*OR $NIL
           (*CONS (*OR $NIL $NON-T-NIL-SYMBOL $T)
                  (*RECUR SYMBOL-LISTP))))
```

The algorithm constructs a segment which maps this argument to $T. Then, it negates and canonicalizes this descriptor into a *REC descriptor characterizing its complement (the !REC1 descriptor below), and this descriptor becomes the argument in a segment which maps it to a $NIL result. The segments for the function, then, are:

```
(((*REC SYMBOL-LISTP
        (*OR $NIL
             (*CONS (*OR $NIL $NON-T-NIL-SYMBOL $T)
                    (*RECUR SYMBOL-LISTP)))))
 -> $T)
(((*REC !REC1
        (*OR $CHARACTER
             $INTEGER
             $NON-INTEGER-RATIONAL
             $NON-T-NIL-SYMBOL
             $STRING
             $T
             (*CONS *UNIVERSAL (*RECUR !REC1))
             (*CONS (*OR $CHARACTER
                         $INTEGER
                         $NON-INTEGER-RATIONAL
                         $STRING
                         (*CONS *UNIVERSAL *UNIVERSAL))
                    *UNIVERSAL))))
 -> $NIL))
```

An aside on !REC notation: When the inference system generates a new *REC descriptor, it supplies a unique label for the descriptor. In the exposition in this thesis, we use the notation !RECn, where n is some positive integer. The name generator maintains a counter, which is appended to the characters "!REC" in forming a new symbol, thus ensuring the uniqueness of each label. In fact, though, the counter is reset as each new function is submitted. In order to guarantee uniqueness, then, the label generator also appends the name of the function being analyzed to the *REC name. Since this additional verbage would clutter the exposition, we do not display the function name component of *REC names.

Next, this signature is tentatively added to the database of function signatures, and the function is submitted to the checker for validation. The checker produces the following segments. We will clearly see the combinatoric explosion to which we referred above.

```
(((*CONS $NIL
         (*REC SYMBOL-LISTP
               (*OR $NIL (*CONS (*OR $NIL $NON-T-NIL-SYMBOL $T)
                                (*RECUR SYMBOL-LISTP))))))
 -> $T)
(((*CONS $NIL
         (*REC !REC1
               (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                    $NON-T-NIL-SYMBOL $STRING $T
                    (*CONS *UNIVERSAL (*RECUR !REC1))
                    (*CONS (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                                $STRING (*CONS *UNIVERSAL *UNIVERSAL))
                           *UNIVERSAL)))))
 -> $NIL)
(((*CONS $NON-T-NIL-SYMBOL
         (*REC SYMBOL-LISTP
               (*OR $NIL (*CONS (*OR $NIL $NON-T-NIL-SYMBOL $T)
                                (*RECUR SYMBOL-LISTP))))))
 -> $T)
(((*CONS $NON-T-NIL-SYMBOL
         (*REC !REC1
               (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                    $NON-T-NIL-SYMBOL $STRING $T
                    (*CONS *UNIVERSAL (*RECUR !REC1))
                    (*CONS (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
```

```
                                $STRING (*CONS *UNIVERSAL *UNIVERSAL))
                           *UNIVERSAL)))))
 -> $NIL)
(((*CONS $T
         (*REC SYMBOL-LISTP
               (*OR $NIL (*CONS (*OR $NIL $NON-T-NIL-SYMBOL $T)
                                (*RECUR SYMBOL-LISTP))))))
 -> $T)
(((*CONS $T
         (*REC !REC1
               (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                    $NON-T-NIL-SYMBOL $STRING $T
                    (*CONS *UNIVERSAL (*RECUR !REC1))
                    (*CONS (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                                $STRING (*CONS *UNIVERSAL *UNIVERSAL))
                           *UNIVERSAL)))))
 -> $NIL)
(((*CONS $CHARACTER *UNIVERSAL)) -> $NIL)
(((*CONS $INTEGER *UNIVERSAL)) -> $NIL)
(((*CONS $NON-INTEGER-RATIONAL *UNIVERSAL)) -> $NIL)
(((*CONS $STRING *UNIVERSAL)) -> $NIL)
(((*CONS (*CONS *UNIVERSAL *UNIVERSAL) *UNIVERSAL)) -> $NIL)
(($CHARACTER) -> $NIL)
(($INTEGER) -> $NIL)
(($NIL) -> $T)
(($NON-INTEGER-RATIONAL) -> $NIL)
(($NON-T-NIL-SYMBOL) -> $NIL)
(($STRING) -> $NIL)
(($T) -> $NIL))
```

Loosely speaking, each of these segments represents one of the disjuncts produced from a single unfolding of the *REC forms in the original segments. The containment test ensures that each of them is contained within one of the original descriptors. In this case, it is easy to see which one contains each, since the signature contains only one segment with a $T result and one with a $NIL result.

The checker also validates that the guard is (*UNIVERSAL), that this guard descriptor is complete, that the original guard is contained in the guard computed by the checker, that the guards of all functions called in the body are complete, and that the function is a recognizer.

For an example of a function whose signature involves type variables, consider:

```
(DEFUN CADR (X)
  (DECLARE
    (XARGS :GUARD
           (IF (NULL X)
               (NULL X)
               (IF (CONSP X)
                   (IF (NULL (CDR X)) (NULL (CDR X)) (CONSP (CDR X)))
                   NIL))))
  (CAR (CDR X))))
```

The guard is a cumbersome expression which ensures that the argument is such that CDR is defined on it and CAR is defined on its CDR. (It is not a simple conjunction of CONSP tests because CAR and CDR each return NIL when given NIL as an argument.) The guard is itself free of guard violations, and the tool produces the guard descriptor:

```
((*OR $NIL
      (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))))
```

This guard descriptor is complete. Since the function has a guard, it fails one of the first tests for being a recognizer, so the tools employs DERIVE-EQUATIONS and SOLVE-EQUATIONS to compute its segments. No guard violations are found in the body, thanks to the assumption provided by the guard. The segments produced are:

```
(((*OR $NIL (*CONS *UNIVERSAL $NIL))) -> $NIL)
(((*CONS *UNIVERSAL (*CONS &1 *UNIVERSAL))) -> &1))
```

Thus, if the argument is a CONS whose CDR is a CONS, the result is the same value as the CAR of the CDR. Otherwise, the result is NIL.

The checker generates an identical guard descriptor, but the checker has a more stringent test for guard completeness than the inference tool. The checker requires that guard be a conjunction of recognizer calls on distinct parameters. Had the guard expression been made the body of another unguarded function, that function would have qualified as a recognizer. Then, the guard for CADR could have been simply a call to this new function on the parameter X, and the checker would have judged the guard complete.

The segments generated by the checker are:

```
(($NIL) $NIL)))
(((*CONS *UNIVERSAL $NIL)) -> $NIL)
(((*CONS *UNIVERSAL (*CONS &125 *UNIVERSAL))) -> &125)
```

The first two segments are contained in the first original segment, and the third segment is contained in the second original. Thus, the signature is validated.

Chapter 4

THE IMPLEMENTATION OF THE INFERENCE ALGORITHM

This chapter is an informal synopsis of the type inference algorithm, intended to provide insight into all the operations undertaken by the implementation.

This discussion will bootstrap itself, in a sense. First, we will mention some key functions and describe them just well enough to then follow with a high-level overview of the algorithm. Then we devote the bulk of the chapter to a very detailed discussion of the most critical component algorithms.

Keep in mind that this algorithm is purely heuristic, in that its purpose is to *suggest* a function signature. We make no formal claims to its validity or to the soundness of the operations performed within the algorithm. Formal assurance of the correctness of the signature is left to the checker algorithm.

Recall, the tasks of the algorithm are to:

1. Determine that the guard expression, if there is one, is well-formed. The guard must not contain any recursive calls, it can call only functions which are in the system state, and its evaluation must not cause any guard violations detectable within the granularity of the type system.
2. If the guard is well-formed, extract a type descriptor representation of the guard.
3. Determine whether the guard type descriptor is a complete characterization of the guard.
4. Determine whether there are any guard violations in the function body which are detectable within the granularity of the type system.
5. If there are no guard violations, formulate the signature segments for the function.

Fortunately, we define our subset of Common Lisp so that we can make some handy assumptions about the function we are given.

1. A new function may call only itself and functions which exist in the database.
2. The only variables are those in the parameter list.
3. Quoted constants must be objects whose types are known to the type system. For instance, complex numbers are not handled.

Furthermore, it was not our intention to support type inference for non-terminating functions. Though there is no specific prohibition of such functions, the algorithm may be prone to generating weak results or to diverge, thus yielding no result at all.

The algorithm as it stands is the result of an evolving prototype effort. Its development was arbitrarily halted at a point where it was judged to be generating sufficiently interesting results. As such, there are many improvements which could be made to the algorithm, and some are suggested within this chapter.

4.1 A Few Important Functions

Here is a brief description of several important functions which will be mentioned in the overview of the algorithm below. Details and some parameters are omitted to keep the big picture clearer. Several of these functions will be explained in greater detail later.

`UNIFY-DESCRIPTORS (DESCRIPTOR1 DESCRIPTOR2 ...)`

If one thinks of descriptors as terms which, under some arbitrary substitution of ground type descriptors for type variables, represent sets of data objects, then one may think of UNIFY-DESCRIPTORS as returning a descriptor characterizing the intersection of those two sets under any common substitution. UNIFY-DESCRIPTORS returns a descriptor which represents the common ground between DESCRIPTOR1 and DESCRIPTOR2, annotated with substitutions mapping type variables to descriptors.

`GUARD-VIOLATION-ON-FNCALL (FORMALS ACTUALS)`

FORMALS and ACTUALS are lists of descriptors. FORMALS is the descriptor list corresponding to the guard of the function being called. ACTUALS are the descriptors of the actual parameters. GUARD-VIOLATION-ON-FNCALL returns T if the ACTUALS descriptors fail to satisfy the FORMALS descriptors. This judgement is made essentially by unifying the FORMALS and ACTUALS and seeing that the result is isomorphic to ACTUALS, thus ensuring FORMALS placed no additional constraint on ACTUALS.

`CANONICALIZE-DESCRIPTOR (DESCRIPTOR)`

This function employs a number of heuristics to massage a descriptor into a canonical form. There is not always a single unique canonicalization computed by this function for different descriptors characterizing the same set of values.

`TYPE-PREDICATE-P (FORM ARGLIST RECOGNIZERS TYPE-ALIST)`

Given a Lisp expression FORM, a list of names of formal parameters the function ARGLIST, a list of known RECOGNIZERS from the system database, and a TYPE-ALIST characterizing the types of the variables in the environment, TYPE-PREDICATE-P determines what information can be determined about the parameters by evaluating the form.

`PREPASS (FORM FNNAME FUNCTION-SIGNATURES RECOGNIZERS)`

PREPASS takes a Lisp expression FORM and the system database and returns an IF-normalized form, which will always evaluate to the same value as FORM in any environment. IF-normalization consists mainly of transforming the test form in the IF so that it always evaluates to either T or NIL. It also reduces an IF form to its THEN or ELSE arms if it can deduce trivially that the test always evaluates to T or NIL, respectively.

4.2 Overview of the Algorithm

At the highest level, the algorithm does the following. When a function is submitted, if it has no guard (or the guard is T), we check its PREPASS-ed body to see if it is a recognizer function (See Section 4.4.5 below). If it is, we compute the variable-free type descriptor which characterizes what it recognizes. Then we construct the signature which maps that descriptor to $T and its negation to $NIL. We store this signature in the database, noting that the function's guard descriptor is (*UNIVERSAL), that the guard is complete, and that it recognizes forms conforming to its descriptor.

If the function is not a recognizer, we check the PREPASS-ed guard to ensure it has no guard violations itself, then use TYPE-PREDICATE-P to construct a type alist which captures what is implied about the actual parameters when the guard expression evaluates to a non-NIL value. If the real guard is stronger than the type alist (for example, if the guard were (EQUAL X 3), the type alist would just say that X is an INTEGER), we note that the guard is not complete. Then, using the type alist as a starting point, we construct a representation of the function body which is in the form of a table with one entry for each subexpression in the body, ordered by Lisp evaluation order. This table, described later, will be the basis for the computation of the function signature.

Next we traverse this table, computing the types of each subexpression. For each subexpression, we produce a list of segments, each mapping a list of descriptors characterizing the types of the variables in the environment to a descriptor characterizing the result. There can be more than one segment for a given subexpression because different variable types may produce different result types. Ultimately, the set of segments for the outermost subexpression, i.e., the function body, will become the segments of the signature for the function.

How are these segments produced, then? Suppose we are analyzing a function call. We have already computed the segments for each of its arguments. First we take the cross product of all the segments for all the arguments. This gives us a pattern of argument descriptors for every possible combination we have uncovered. Now, we fetch the signature for the called function from the system database. The signature is composed of a guard vector of the same arity as the parameter list, and a collection of segments. For each actual parameter argument pattern, we first "unify", using UNIFY-DESCRIPTORS, that pattern against the pattern representing the guard of the called function. If the result of this unification indicates that the guard descriptor further restricts the argument types, we report a failure to verify guards, terminate the computation, and make no addition to the system database. Otherwise, we unify each argument pattern against the argument pattern in each segment of the signature for the called function. For each segment for which this unification is successful, i.e., for which a non-empty result is produced, we are rewarded with a unifying substitution, and this substitution is applied to the result descriptor for the matching segment. This, when canonicalized, gives us a result type which characterizes a possible result corresponding to our argument pattern. We map our argument pattern to this result descriptor to produce a segment. Since there may be multiple formal argument patterns for which the unification is successful, we may get multiple segments for any given actual argument pattern. For each actual parameter descriptor pattern from the cross product, we perform this unification with each segment for the called function, collecting all the results to form the set of segments for this function call.

A problem, of course, is what to do about recursive function calls? The type inference process just described depends on the existence of a signature for every called function, and we are only in the process of computing the signature for the function at hand. Initially, we have nothing whatsoever with which to work. We address this problem by doing an iterative approximation of the signature for the function. An iteration amounts to one pass through the entire table representing the function. During the first pass, we have no segments to match against recursive function calls, so we know nothing about the results of those calls. However, this does not prevent us from producing a first approximation of our segments on this pass. On the second pass, we can use this approximation to produce a refined approximation. Eventually, we hope that the approximation we produce will be equivalent to the one we produced on the previous pass. When this happens, the algorithm has stabilized. We canonicalize our segments into a suitable form, and we are finished. We store the segments in the system database, along with the representation of the function guard, a tag which says whether the guard is complete, and a tag which says the function is not a recognizer (since we would not have resorted to this algorithm if it were).

Ensuring that we have reached stability is non-trivial, however. The function is recursively composing structures which contribute to the result. If we proceed naively, these structures will just grow with each iteration, our segments will reflect that growth, and nothing will ever stabilize. Therefore, we employ a technique for determining when these structures are taking on a form which can be described with a closed recursive descriptor. With such a closed recursive descriptor, the expansion and folding which occurs during an iteration of the algorithm will produce the same result we had in the previous pass, thus leading to stabilization.

In some situations, our heuristics may not be strong enough to close our recursive forms in such a way that stability can be reached. To prevent the algorithm from running away, we simply place a limit on the number of iterations it will attempt before giving up.

A characteristic collection of results generated by the algorithm is presented in Chapter 8, and an extensive collection of results is available via anonymous ftp. (See Appendix H.)

4.3 The Type Descriptor Language

The following is the grammar for type descriptors which are handled by the inference algorithm.

```
  <descriptor> ::= <simple descriptor>  |  <variable>  |  *EMPTY  |
                   (*CONS <descriptor> <descriptor> )  |
                   (*OR <descriptor>* )  |  (*NOT <descriptor> )  |
                   <rec descriptor>  |  <and descriptor>  |
                   <fix descriptor> | *MERGE-FIX-POINT  |
                   **RECUR-MARKER**
```

*MERGE-FIX-POINT is a transitory descriptor used to represent a fixed point in the UNIFY-DESCRIPTORS algorithm.

RECUR-MARKER is a transitory descriptor used to represent a fixed point in the RECOGNIZERP algorithm.

```
<simple descriptor> ::= $T | $NIL | $INTEGER | $NON-INTEGER-RATIONAL |
                        $CHARACTER |$STRING | $NON-T-NIL-SYMBOL

<rec descriptor> ::= (*REC <rec name> <recur descriptor> )

<rec name> ::= a symbol whose first character is not "&"

<variable> ::= a symbol whose first character is "&"

<recur descriptor> ::=
    <simple descriptor>  |  <variable>  |  *EMPTY  |
    (*CONS <recur descriptor> <recur descriptor> )  |
    (*OR <recur descriptor>* )  |  (*NOT <recur descriptor> )  |
    <rec descriptor>  |  <and descriptor>  |  <fix descriptor>  |
    (*RECUR <rec name>) | *ISO-RECUR

<dlist descriptor> ::= (*DLIST <descriptor>*)
```

A <recur descriptor> is further constrained so that when it takes the form (*RECUR <rec name>), the <rec name> must be identical to that associated with the immediately containing *REC form. Note that the grammar for <recur descriptor> is just the grammar for <descriptor> extended with the *RECUR form and modified to be internally recursive.

*ISO-RECUR is a transitory form which can replace (*RECUR <rec name>) in a small algorithm which determines if two *REC descriptors are isomorphic.

*DLIST is simply a notation for packaging a list of simple descriptors into a single form for submission to functions like UNIFY-DESCRIPTORS.

In an *AND form, the <descriptor> is a constraint on the descriptor of the <rec arm>. The <rec arm> is some kind of recursive form, either a <recursive arm> or a <descriptor> which contains a *REC form.

```
<and descriptor> ::= (*AND <descriptor> <recursive arm> )
```

Annotated descriptors are useful intermediate forms for descriptor unification.

```
<annotated descriptor> ::=
    <descriptor> | <subst descriptor> |
    (*CONS <annotated descriptor> <annotated descriptor> )  |
    (*OR <annotated descriptor>* )  |
    (*NOT <annotated descriptor> )  |
    <annotated rec descriptor>  |  <annotated and descriptor> |
```

A <subst descriptor> is a marked descriptor characterized by the embedded <annotated descriptor> but indicating that its place is held in common by the indicated variable.

```
<subst descriptor> ::= (*SUBST <variable> <annotated descriptor>)

<annotated rec descriptor> ::=
    (*REC <recname> (*OR <annotated rec arm descriptor>* ) )

<annotated rec arm descriptor> ::= <annotated descriptor>  |
                                   <annotated recursive arm>

<annotated recursive arm> ::=
    A <recursive arm> with any embedded descriptor being potentially
    annotated

<annotated and descriptor> ::=
    (*AND <annotated descriptor> <annotated recursive arm> )
```

A <fix descriptor> is constructed by SOLVE-EQUATIONS as a representation for the value returned by a recursive call of the function being analyzed. The <fix body form> grammar is the grammar for descriptors modified to be internally recursive and to include a *FIX-RECUR form.

```
<fix descriptor> ::=
    (*FIX (*DLIST <descriptor>*) <fix body form> )

<fix body form> ::=
    <simple descriptor>  |  <variable>  |  *EMPTY  |
    (*CONS <fix body form> <fix body form> )  |
    (*OR <fix body form>* )  |  (*NOT <fix body form> )  |
    <rec descriptor>  |  <and descriptor>  |  <fix descriptor>  |
    (*FIX-RECUR (*DLIST <descriptor>*) )
```

4.4 Details of Significant Sub-algorithms

Before we delve into the higher-level algorithms which produce signatures for recognizers and for general functions, it is useful to understand several very important utility functions which help implement the algebra of type descriptors. These are UNIFY-DESCRIPTORS, which finds the common ground between two descriptors, CANONICALIZE-DESCRIPTOR, which performs canonicalization, PREPASS, which performs IF-normalization, and TYPE-PREDICATE-P, which extracts type information from a certain important class of descriptors. Then we can move on to RECOGNIZERP and DESCRIPTOR-FROM-FNDEF, which determine if a function is a recognizer and derive its associated descriptor. Finally we will look at DERIVE-EQUATIONS, SOLVE-EQUATIONS, and INFER-SIGNATURE, which compute type signatures for general functions.

4.4.1 UNIFY-DESCRIPTORS

Think for the moment of descriptors as terms which, under some arbitrary substitution mapping type variables to ground type descriptors, represent sets of data objects. With this view, one can think of UNIFY-DESCRIPTORS as returning the intersection of those two sets under any common substitution. Given two descriptors, DESCRIPTOR1 and DESCRIPTOR2, UNIFY-DESCRIPTORS returns a descriptor which represents the common ground between DESCRIPTOR1 and DESCRIPTOR2, plus a substitution list mapping type variables to descriptors which embodies one part of the unification of the descriptors. We say "one part of" because in the presence of disjunction in the descriptor language, descriptor unification also performs an intersection operation not typically associated with classical unification. For example:

```
(UNIFY-DESCRIPTORS '(*OR $CHARACTER $INTEGER) '(*OR $INTEGER $NIL))
 =
($INTEGER . NIL)
```

Here, though no variables appear, the unifier still must find the common ground between the two disjunctions, and this is essentially an intersection problem. An example which combines the tasks of finding an appropriate substitution and taking an intersection is:

```
(UNIFY-DESCRIPTORS '(*CONS &1 $NIL) '(*OR $INTEGER (*CONS $INTEGER &2)))
 =
((*CONS $INTEGER $NIL) . ((&1 . $INTEGER) (&2 . $NIL)))
```

Here the result must be a *CONS, since that is the only possibility allowed by DESCRIPTOR1. DESCRIPTOR2 requires that, if the object it represents is a CONS, the CAR be an $INTEGER, and DESCRIPTOR1 requires that the CDR be $NIL. Hence, the pair in the result indicates that the resulting descriptor must be (*CONS $INTEGER $NIL), and we obtained this result in part by instantiating the variables according to the substitution ((&1 . $INTEGER) (&2 . $NIL)).

UNIFY-DESCRIPTORS also returns a second mapping list called a restriction list, which is of no particular interest to the external caller (unless it is exploited -- see Section 4.4.7), but which is important internally in dealing with variables mapped to disjunctive forms. If a form within one of the input descriptors is marked with an annotation which associates it with a type variable, and if that subform is later restricted in the course of unification, then an entry is made in the restrictions list which maps the variable to its restricted form.

Henceforth in this section, we will use the angle-bracketed notation:

```
<< descriptor substs restrictions >>
```

to indicate the form of a result returned by UNIFY-DESCRIPTORS, and the extraction functions TERM, SUBSTS, and RESTRICTIONS to access the pieces.

UNIFY-DESCRIPTORS takes some parameters which we have not yet mentioned. Each of them safeguard against particular kinds of infinite recursion.

1. OPEN-NOT-REC is a flag used only to regulate whether CANONICALIZE-NOT-DESCRIPTOR opens up *REC descriptors which occur within a *NOT form.
2. STACK-RECS is a stack of pairs of *REC descriptors. Every time UNIFY-DESCRIPTORS is called where both DESCRIPTOR1 and DESCRIPTOR2 are *REC descriptors, the pair is pushed on the stack. If on a deeper recursive call, the same pair is encountered, we have

found a fixed point in the recursive descent, and we can construct a closed *REC form to represent the result.

3. TERM-RECS is a stack of pairs of descriptors. In each pair, one is a *REC and the other is not. Whenever one descriptor argument to UNIFY-DESCRIPTORS is a *REC and the other not, the pair is put on this stack, and if it is encountered again on recursion, the algorithm fails, returning the *EMPTY descriptor. This is explained below.

To illustrate the action of UNIFY-DESCRIPTORS, we state rules providing generic examples of results it returns. A rule is of the form

```
(UNIFY-DESCRIPTORS D1 D2) => <<TD SUBSTS RESTRICTIONS>>
```

where D1 and D2 are various forms representing the kinds of descriptors to which the rule applies, and the triple represents, in terms of D1 and D2, the result produced by the unification. In some cases, we supply additional remarks to help clarify the adequacy of the result. UNIFY-DESCRIPTORS is commutative, so we will not bother to present a result with the arguments reversed. To imagine a soundness argument for the result, one might imagine a containment relation for descriptors and then question whether the result descriptor is contained in both the arguments.

By convention in our rules, let the names D, D1, D2, ... represent type descriptors. The SUBSTS and RESTRICTIONS lists have elements of the form (&i . <descriptor>).

```
Rule:
(UNIFY-DESCRIPTORS D D) => <<D NIL NIL>>
```

```
Rule:
(UNIFY-DESCRIPTORS *EMPTY D) => <<*EMPTY NIL NIL>>
```

```
Rule:
(UNIFY-DESCRIPTORS D *UNIVERSAL) => <<D NIL NIL>>
```

In the following rule, the *SUBST form is the annotation mentioned above which associates a variable with a component of the result descriptor. The form of a *SUBST is (*SUBST &i <descriptor>), where &i is the variable. The set of values represented by the *SUBST is the set represented by its <descriptor>. Thus, unification consists of unifying against the <descriptor>, and then, if this unification results in a narrowing of <descriptor>, noting in the restrictions list that the variable is restricted accordingly. REBUILD-SUBST-FORM is the function which does this, modifying the result of the unification by adding to the restriction list a restriction on the *SUBST variable whenever the *SUBST form is restricted by the unification.

```
Rule:
(UNIFY-DESCRIPTORS (*SUBST &i D1) D2) =>
   where unified-form = (UNIFY-DESCRIPTORS D1 D2)
     if TERM (unified-form) = *EMPTY
        then <<*EMPTY NIL NIL>>
        else (REBUILD-SUBST-FORM (*SUBST &i D1) unified-form)
```

```
Rule:
(UNIFY-DESCRIPTORS &i (*OR .. &i ..)) => <<&i NIL NIL>>
```

```
Rule:
Under the condition that D contains an occurrence of the variable &i
nested within both an *OR and a *CONS,
(UNIFY-DESCRIPTORS &i D) =>
```

```
  <<(*SUBST &i (*REC <rec-name> D/((&i . (*RECUR <rec-name>))))
    ((&i . (*SUBST &i (*REC <rec-name> D/((&i . (*RECUR <rec-name>))))))
    NIL>>

"/" signifies application of a substitution.  So
"D/((&i . (*RECUR <rec-name>)))" signifies the descriptor D with
each occurrence of &i replaced by (*RECUR <rec-name>).
```

The preceding rule illustrates a different treatment of variables than we will see in the formal semantics and in the checker algorithm. In the inference algorithm, variables serve a dual purpose, indicating both the sharing of data values and the need for common type instantiation. Instances of the latter are cleaned up at the end of the inference process. In hindsight, we believe these purposes should have been separated, treated differently, and preserved in the formal semantics.

```
Rule:
Under the condition that D contains an occurrence of the variable &i,
but where D is neither an *OR with &i as one of its disjuncts nor a
form where the variable &i is nested within both an *OR and a *CONS,
(UNIFY-DESCRIPTORS &i D) => <<*EMPTY NIL NIL>>
```

This covers only the case where D contains &i within a *CONS but not within an *OR. If we created a *REC in the spirit of the previous case, it would be a non-terminating structure.

```
Rule:
Unifying a descriptor D with an *OR descriptor which has D as one
of its disjuncts yields D intact with no substitutions necessary.
Otherwise, the result is the disjunction of the unification of D with
each of the disjuncts, with each unification yielding its own
substitution.

(UNIFY-DESCRIPTORS D (*OR D1 .. DN)) =>
  (if (MEMBER D (D1 .. DN))
      then <<D NIL NIL>>
      else (*OR (UNIFY-DESCRIPTORS D D1) .. (UNIFY-DESCRIPTORS D DN)))


Rule:
Where &i does not occur in D
(UNIFY-DESCRIPTOR &i D) =>
  <<(*SUBST &i D) ((&i . (*SUBST &i D))) NIL>>


Rule:
(UNIFY-DESCRIPTORS (*REC <rec-name> (.. (*RECUR <rec-name>) ..)) D)
  =>
 (UNIFY-DESCRIPTORS
     (.. (*REC <rec-name> (.. (*RECUR <rec-name>) ..)) ..) D)

This is to say that when we unify a *REC form with another form, we
just open the *REC and unify.
```

When unifying two non-identical *REC descriptors, there is a danger of getting lost in a non-terminating recursion. UNIFY-DESCRIPTORS does a nice trick to prevent this and to reach closure. When it is asked to unify two *REC descriptors, it places the pair on a stack before proceeding. This stack is the STACK-RECS parameter mentioned above. Then, as usual, it opens one up and unifies against the other, which results in the other being opened up, so that their internals get unified. This constructs a new, unified descriptor as it descends, and sooner or later, we will reach a point where the two *REC forms, which reappear nested with every unfolding, once again are to be unified with each other. If we just continued unfolding, the algorithm would never terminate, but before unifying two *RECS, UNIFY-DESCRIPTORS checks the stack to see if they have already been encountered. If so, we have reached a fixed point in the recursion. We simply create a new *REC label <recname>, declare the result of the nested unification to be (*RECUR <recname>), and when we finally unwind our way to the top, wrap a (*REC <recname> ..) around the accumulated result. For example, unifying the descriptors

```
  (*REC R1 (*OR $NIL (*CONS (*OR $INTEGER $CHARACTER) (*RECUR R1))))
  (*REC R2 (*OR $NIL $T (*CONS *UNIVERSAL (*CONS $INTEGER (*RECUR R2)))))
```

returns the descriptor:

```
(*REC !REC1 (*OR $NIL (*CONS (*OR $CHARACTER $INTEGER)
                             (*CONS $INTEGER (*RECUR !REC1)))))
```

Another pitfall is the possibility of infinite recursion when each descriptor argument contains a *REC form, but the forms unwind out of sync with one another. For example, consider the descriptors:

```
  (*REC FOO (*OR $NIL (*CONS *UNIVERSAL
                             (*CONS *UNIVERSAL (*RECUR FOO))))
  (*CONS *UNIVERSAL
         (*REC BAR (*OR $NIL (*CONS *UNIVERSAL
                                    (*CONS *UNIVERSAL (*RECUR BAR))))
```

The *REC forms are opened up alternately, leapfrogging one another in an endless unwinding. This unwinding does not get spotted under the STACK-RECS regimen because at no point do we attempt to unify two *REC forms head to head. So we employ the TERM-RECS mechanism, mentioned above, to prevent this case from running away. TERM-RECS is a stack of pairs of descriptors, where in each pair one is a *REC and the other is not. If on recursion we encounter arguments which match pairs already on the TERM-RECS stack, we return *EMPTY as the result, since any object satisfying both descriptors would be impossible to construct. In the example above, it would have to be a list whose length is both even and odd.

```
Rule:
(UNIFY-DESCRIPTORS D (*NOT D)) => <<*EMPTY NIL NIL>>
```

Canonicalization complements a simple descriptor and forces the *NOT inside a structure, as we shall see. This reduces unification of *NOT descriptors to unification on other forms. The rare exception is where the canonicalization of the *NOT form produces no change. This occurs only when D1 is a *RECUR form, and this peculiarity occurs only during recognizer analysis, which is explained below, along with the treatment of *AND and *RECUR forms.

```
Rule:
Where D1 is not a *RECUR descriptor,
(UNIFY-DESCRIPTORS (*NOT D1) D2) =>
  (UNIFY-DESCRIPTORS (CANONICALIZE-DESCRIPTOR (*NOT D1)) D2)
```

```
Rule:
Where D is not an *OR descriptor:
(UNIFY-DESCRIPTORS (*NOT (*RECUR <rec-name>)) D) =>
  <<(*AND (*NOT (*RECUR <rec-name>)) D) NIL NIL>>

otherwise
(UNIFY-DESCRIPTORS (*NOT (*RECUR <rec-name>)) (*OR D1 .. DN)) =>
  (*OR <<(*AND (*NOT (*RECUR <rec-name>)) D1) NIL NIL>>
       ..
       <<(*AND (*NOT (*RECUR <rec-name>)) DN) NIL NIL>>)
```

As an optimization in certain simple cases, we employ a classical unification algorithm, UNIFY. We do this when both descriptors are composed solely of type variables, instances of the simple types $CHARACTER, $INTEGER, $NIL, $NON-INTEGER-RATIONAL, $NON-T-NIL-SYMBOL, $STRING, or $T, or *CONS or *DLIST descriptors whose components are also suitable descriptors. UNIFY takes two such descriptors and returns either *failure if unification fails or a unifying substitution list. Thus, we have the rule:

```
Rule:
Where D1 and D2 are suitable for classical unification:
(UNIFY-DESCRIPTORS D1 D2) =>
  let unify-result = (UNIFY D1 D2)
    if unify-result = *failure
       (*EMPTY NIL NIL)
       <<(APPLY-SUBSTS unify-result D1) unify-result NIL>>
```

Now consider the general unification of two *CONS descriptors. Since this is a complex operation, we decompose it into several principal functions, expressed as follows:

```
Rule:
Where D1 is of the form (*CONS D1-CAR D1-CDR)
  and D2 is of the form (*CONS D2-CAR D2-CDR)
(UNIFY-DESCRIPTORS D1 D2) =>
  let unify-cars = (UNIFY-DESCRIPTORS D1-CAR D2-CAR)
    if (EQUAL (TERM unify-cars) *EMPTY)
       then <<*EMPTY NIL NIL>>
       else if unify-cars is of the form (*OR u-form1 .. u-formn)
               then (*OR (UNIFY-CDR-UNIFIED-FORMS u-form1 D1 D2)
                         ...
                         (UNIFY-CDR-UNIFIED-FORMS u-formn D1 D2))
               else (UNIFY-CDR-UNIFIED-FORMS unify-cars D1 D2)

where
UNIFY-CDR-UNIFIED-FORMS (unify-cars D1 D2) =
  let unify-cdrs =
        (UNIFY-DESCRIPTORS
          (APPLY-RESTRICTIONS (RESTRICTIONS unify-cars)
                              (APPLY-SUBSTS (SUBSTS unify-cars)) D1-CDR)
          (APPLY-RESTRICTIONS (RESTRICTIONS unify-cars)
                              (APPLY-SUBSTS (SUBSTS unify-cars)) D2-CDR))
    if (EQUAL (TERM unify-cdrs) *EMPTY)
       then <<*EMPTY NIL NIL>>
       else if unify-cars is of the form (*OR u-form1 .. u-formn)
               then (*OR (MAKE-CONS-UNIFIED-FORM u-form1 unify-cdrs)
                         ...
                         (MAKE-CONS-UNIFIED-FORM u-formn unify-cdrs))
               else (MAKE-CONS-UNIFIED-FORM unify-cars unify-cdrs)

where
MAKE-CONS-UNIFIED-FORM (unify-cars unify-cdrs) =
  <<(*CONS (APPLY-RESTRICTIONS
             (RESTRICTIONS unify-cdrs)
             (APPLY-SUBSTS (SUBSTS unify-cdrs) (TERM unify-cars)))
           (TERM unify-cdrs))
    (COMPOSE-SUBSTS
      (APPLY-RESTRICTIONS (RESTRICTIONS unify-cdrs) (SUBSTS unify-cars))
      (SUBSTS unify-cdrs))
    (COMPOSE-RESTRICTIONS
      (APPLY-RESTRICTIONS (RESTRICTIONS unify-cdrs)
                          (APPLY-SUBSTS (SUBSTS unify-cdrs)
                                        (RESTRICTIONS unify-cars)))
      (RESTRICTIONS unify-cdrs))>>
```

The use of the SUBSTS lists when unifying CONSes is essentially the same as with classical unification. The substitutions from the CAR are applied to the CDRs before unifying, and the substitutions from the

CARs and CDRs are merged to become the substitutions for the CONS.

The RESTRICTIONS list is used to deal with the presence of disjunction in the descriptor language. If one of the CARs is a variable, we return a *SUBST form,

```
(*SUBST <variable> <descriptor>)
```

for unify-cars, as already discussed. The value set represented by a *SUBST is just that of the <descriptor>, but the form is annotated with the variable for future reference. We carry the <descriptor> form into the unification of the CDRs (through substitution for the variable in the CDRs). If, in unifying the CDRs, this form is further constrained, we must reflect this constraint everywhere the variable once occurred. A restriction is formed every time such a constraint occurs. As we construct the *CONS result from its unified CARs and CDRs, we apply these restrictions wherever there was a *SUBST form for the variable in question. This can be viewed simply as an elaborate way to ensure that the substitutions discovered in the unification are uniformly applied throughout the result.

```
Rule:
Unification of lists of descriptors, denoted (*DLIST D1 D2 .. DN) works
exactly like the unification of *CONS descriptors.   We treat the list
(*DLIST D1 D2 .. DN), for unification purposes, as if it were of the
form (*CONS D1 (*CONS D2 .. DN)).  Actually, the *DLIST construction is
the general case, and *CONS is the special case, but for the purposes of
explanation, *CONS was enough to bear.  *DLIST unification is only
meaningful when both arguments are *DLISTs and have the same arity.
```

```
Rule:
Where D3 is one of the simple descriptors $CHARACTER, $INTEGER, $NIL,
$NON-INTEGER-RATIONAL, $NON-T-NIL-SYMBOL, $STRING, or $T,
 (UNIFY-DESCRIPTORS (*CONS D1 D2) D3) => <<*EMPTY NIL NIL>>
```

The following peculiar operation occurs only during the analysis of recognizer functions, as this is the only time *AND forms are built, and they are subsequently filtered away.

```
Rule:
(UNIFY-DESCRIPTORS (*AND D1 (*RECUR <rec-name>)) D2) =>
  let uform = (UNIFY-DESCRIPTORS D1 D2)
    if (TERM uform) = *EMPTY
       <<*EMPTY NIL NIL>>
       if uform is of the form (*OR uform_1 .. uform_n)
          then (*OR <<(*AND (TERM uform_1) (*RECUR <rec-name>))
                      (SUBSTS uform_1)
                      (RESTRICTIONS uform_1)>>
                    ...
                    <<(*AND (TERM uform_n) (*RECUR <rec-name>))
                      (SUBSTS uform_n)
                      (RESTRICTIONS uform_n)>>)
          else <<(*AND (TERM uform) (*RECUR <rec-name>))
                 (SUBSTS uform)
                 (RESTRICTIONS uform)>>
```

Here is one place where the *AND forms are built during the analysis of recognizers. Again, this only happens during that analysis, because any other time a (*RECUR <rec-name>) only occurs within a *REC, since we always open up *REC forms by replacing the nested *RECUR form with the *REC form.

```
Rule:
(UNIFY-DESCRIPTORS (*RECUR <rec-name>) D) =>
  if D is of the form (*OR D1 .. DN)
     then (*OR (*AND D1 (*RECUR <rec-name>))
```

```
              ...
              (*AND DN (*RECUR <rec-name>)))
     else (*AND D (*RECUR <rec-name>))
```

The top level function of the unifier is named UNIFY-DESCRIPTORS-INTERFACE.

4.4.2 CANONICALIZE-DESCRIPTOR

CANONICALIZE-DESCRIPTOR attempts to put type descriptors into canonical form. There is not always a single unique canonicalization computed by this function for different descriptors characterizing the same particular set of values. We simply employ a collection of reductions which tend toward canonicalization. Since these reductions amount to heuristic choices, we may not discover an optimal form. For example, two equivalent descriptors which do not canonicalize to the same form are:

```
(*REC FOO1 (*CONS $INTEGER (*OR $NIL (*RECUR FOO1))))
and
(*CONS $INTEGER (*REC FOO2 (*OR $NIL (*CONS $INTEGER (*RECUR FOO2)))))
```

Of course, since recursive descriptors always recur from a point which is nested within both an *OR and a *CONS, there is no reason we could not force the top level form within the *REC to be definitely one or the other. Consider the first descriptor. If we required an *OR at the top level, we would force the outer (*CONS $INTEGER ..) outside the *REC and also inside the *OR around the *RECUR, and we would get a form equal to the second. But the simple fact is that in the current implementation, we do not do this. A nice upgrade to the implementation would be to install this canonicalization. Many other such omissions could be found and possibly implemented.

We will state the actions which CANONICALIZE-DESCRIPTOR performs in terms of a sequence of rules. Each rule is of the form

```
  <descriptor1> => <descriptor2>
```

where the left hand side is a descriptor whose form generically indicates the form of a descriptor which is transformed by the rule, and the right hand side gives the form of the result, usually in terms of descriptors appearing within the form on the left. Let D, D1, D2, and D3 represent arbitrary type descriptors. Let R1, R2, and R3 represent arbitary, but distinct names associated with *REC descriptors. Let D-REC, D-REC1, and D-REC2 be descriptors which contain *RECUR forms, useful, of course, only inside *REC descriptors. Finally, note that *OR is an n-ary descriptor. But for purposes of expressing these rules, without loss of generality we can consider it to be ternary, binary, unary, or 0-ary as convenient.

```
Canonicalization Rule 1:
 (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
      $NON-T-NIL-SYMBOL $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
 => *UNIVERSAL

Canonicalization Rule 2:
 (*OR D *EMPTY) => (*OR D)

The rule above also connotes (*OR D1 *EMPTY D2) => (*OR D1 D2), etc.

Canonicalization Rule 3:
 (*OR D) => D

Canonicalization Rule 4:
 (*OR D1 D2) => (*OR D2 D1)
```

There is a function DESCRIPTOR-ORDER (D1 D2) which defines a complete partial order on descriptors. We perform an ordering canonicalization in accordance with the rule above if (DESCRIPTOR-ORDER D1 D2) is NIL.

```
Canonicalization Rule 5:
 (*OR D1 (*OR D2 D3)) => (*OR D1 D2 D3)

Canonicalization Rule 6:
 (*OR D1 D1 D2) => (*OR D1 D2)

Canonicalization Rule 7:
 (*OR D1 (*REC R1 (*OR D1 D2 D-REC)))
 =>
 (*OR (*REC R1 (*OR D1 D2 D-REC)))

Canonicalization Rule 8:
 (*OR (*CONS D1 D2) (*CONS D3 D2))
 =>
 (*OR (*CONS (*OR D1 D3) D2))

Canonicalization Rule 9:
 (*OR (*CONS D1 D2) (*CONS D1 D3))
 =>
 (*OR (*CONS D1 (*OR D2 D3)))

Canonicalization Rule 10:
 (*OR D *UNIVERSAL) => *UNIVERSAL

Canonicalization Rule 11:
 (*OR) => *EMPTY

Canonicalization Rule 12:
 (*OR (*CONS D1 D2) (*CONS D3 D2)) => (*OR (*CONS (*OR D1 D3) D2))

Canonicalization Rule 13:
 (*OR (*CONS D1 D2) (*CONS D1 D3)) => (*OR (*CONS D1 (*OR D2 D3)))

Canonicalization Rule 14:
 (*CONS D *EMPTY) => *EMPTY

Canonicalization Rule 15:
 (*CONS *EMPTY D) => *EMPTY

Canonicalization Rule 16:
 (*OR (*DLIST D1 D2) (*DLIST D3 D2)) => (*OR (*DLIST (*OR D1 D3) D2))

Canonicalization Rule 17:
 (*OR (*DLIST D1 D2) (*DLIST D1 D3)) => (*OR (*DLIST D1 (*OR D2 D3)))

Canonicalization Rule 18:
 (*DLIST .. *EMPTY ..) => *EMPTY

Canonicalization Rule 19:
 (*NOT $T)
 =>
 (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
      $NON-T-NIL-SYMBOL $STRING (*CONS *UNIVERSAL *UNIVERSAL))

Canonicalization Rule 20:
 (*NOT $NIL)
 =>
 (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
      $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))

Canonicalization Rule 21:
 (*NOT $INTEGER)
 =>
 (*OR $CHARACTER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
      $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
```

```
Canonicalization Rule 22:
 (*NOT $NON-INTEGER-RATIONAL)
 =>
 (*OR $CHARACTER $INTEGER $NIL $NON-T-NIL-SYMBOL $STRING $T
      (*CONS *UNIVERSAL *UNIVERSAL))

Canonicalization Rule 23:
 (*NOT $CHARACTER)
 =>
 (*OR $INTEGER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
      $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))

Canonicalization Rule 24:
 (*NOT $STRING)
 =>
 (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
      $NON-T-NIL-SYMBOL $T (*CONS *UNIVERSAL *UNIVERSAL))

Canonicalization Rule 25:
 (*NOT $NON-T-NIL-SYMBOL)
 =>
 (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL $STRING
      $T (*CONS *UNIVERSAL *UNIVERSAL))

Canonicalization Rule 26:
 (*NOT *EMPTY) => *UNIVERSAL

Canonicalization Rule 27:
 (*NOT *UNIVERSAL) => *EMPTY

Canonicalization Rule 28:
 (*NOT (*NOT D)) => D

Rule Canonicalization 29:
 (*NOT (*CONS D1 D2))
 =>
 (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
      $NON-T-NIL-SYMBOL $STRING $T
      (*CONS (*NOT D1) *UNIVERSAL)
      (*CONS *UNIVERSAL (*NOT D2)))

Canonicalization Rule 30:
 (*NOT (*OR D1 D2))
 =>
 (UNIFY-DESCRIPTORS (*NOT D1) (*NOT D2))

Canonicalization Rule 31:
 D-REC / ((*RECUR R) . (*REC R D-REC))
 =>
 (*REC R D-REC)
```

Canonicalization Rule 31 shows how the expansion of a *REC descriptor can canonicalize to the *REC descriptor itself. For example,

```
(*OR $NIL (*CONS $INTEGER
                 (*REC INT-LISTP
                       (*OR $NIL (*CONS $INTEGER (*RECUR INT-LISTP))))))
```

folds to:

```
(*REC INT-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INT-LISTP))))
```

When we reverse this folding, we refer to the operation as OPEN-REC-DESCRIPTOR.

```
Canonicalization Rule 32:
 (*NOT (*REC R1 D-REC))
```

```
 =>
 (*REC R2 (*NOT (OPEN-REC-DESCRIPTOR (*REC R1 D-REC))))
    / (((*NOT (*REC R1 D-REC)) . (*RECUR R2)))
```

There is a lot of information in the rule above. Read the form on the right hand side as a *REC form with some new label R2 different from R1, whose body is the *not of the body of R1 opened up once, and with the (*RECUR R2) marker replacing the original descriptor. For example, consider the following as our left hand side.

```
(*NOT (*REC INT-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INT-LISTP)))))
```

First we open up INT-LISTP:

```
(*NOT (*OR $NIL
           (*CONS $INTEGER
                  (*REC INT-LISTP
                        (*OR $NIL (*CONS $INTEGER (*RECUR INT-LISTP)))))
```

Then we partially canonicalize the *NOT, but without opening up the *rec again:

```
(*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING
     $T
     (*CONS *UNIVERSAL
            (*NOT (*REC INT-LISTP
                        (*OR $NIL (*CONS $INTEGER (*RECUR INT-LISTP))))))
     (*CONS (*OR $CHARACTER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
                 $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
           *UNIVERSAL))
```

But since this form represents the same set of objects as the original form, and since that original form is embedded within it, we have discovered a new recursive form. We give it a name (NOT-INT-LISTP for the sake of exposition), replace the embedded occurrence with the *RECUR point, and encapsulate in a *REC:

```
(*REC NOT-INT-LISTP
      (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
           $STRING $T
           (*CONS *UNIVERSAL (*RECUR NOT-INT-LISTP)
           (*CONS (*OR $CHARACTER $NIL $NON-INTEGER-RATIONAL
                       $NON-T-NIL-SYMBOL $STRING $T
                       (*CONS *UNIVERSAL *UNIVERSAL))
                  *UNIVERSAL)))
```

```
Canonicalization Rule 33:
 (*NOT (*AND D1 D2))
 =>
 (*OR (*NOT D1) (*NOT D2))
```

```
Canonicalization Rule 34:
 (*FIX (*DLIST D1 .. DM)
       (... (*FIX (*DLIST D1 .. DM) DN) ...))
 =>
 (*FIX (*DLIST D1 .. DM)
       (... (*FIX-RECUR (*DLIST D1 .. DM)) ...))
```

*FIX is a descriptor form used in SOLVE-EQUATIONS (See Section 4.4.7) to find closed forms constructed through function recursion. Though the canonicalization of *FIX forms is not quite of the same flavor as other canonicalizations, it is implemented in CANONICALIZE-DESCRIPTOR anyway. *FIX forms are the mechanism used by the algorithm to discover closed recursive descriptors for forms which are constructed on recursive function calls. One may view *FIX as having the same meaning as *REC, with the *DLIST form representing the label. But the *FIX form signals that a certain variety of heuristics should be employed. This canonicalization rule is very similar to Rule 31, and almost the inverse of OPEN-REC-DESCRIPTOR, the difference being that DN is only an approximation of

```
(... (*FIX (*DLIST D1 .. DM) DN) ...)
```

When the inference tool reaches stability and we employ this rule, it is a perfect approximation. That is the essence of the iterative stabilization algorithm.

Canonicalization Rules 35 - 37 below for *REC descriptors are actually implemented in CANONICALIZE-REC-DESCRIPTOR, which is not in the call tree of CANONICALIZE-DESCRIPTOR. They are separated because they are useful only under certain known circumstances, and there is no need to burden the general canonicalizer with the useless overhead.

```
Canonicalization Rule 35:
 (*REC R (*OR .. (*RECUR R) ..)) => (*REC R (*OR .. *EMPTY ..))
```

By Rule 35, we mean that if the *RECUR form is a top-level disjunct of the body, we eliminate it, since it serves no purpose.

```
Canonicalization Rule 36:
 Where D does not contain (*RECUR R),
  (*REC R D) => D

Canonicalization Rule 37:
 (*REC R (*CONS D (*RECUR R))) => *EMPTY
```

Rule 37 represents any case where the body of a *REC is just a nest of *CONS descriptors with no *OR providing a terminating disjunct.

4.4.3 PREPASS

PREPASS takes a Lisp expression FORM and the system database and returns an IF-normalized form, which will always evaluate to the same value as FORM in any environment. IF-normalization consists mainly of transforming the test form in the IF so that it always evaluates to either T or NIL. Since IF splits on NIL vs. non-NIL values in the test, a non-NIL value serves the same purpose as a T in an IF test. If a test form can evaluate to a value other than T or NIL, PREPASS simply wraps (NULL (NULL ..)) around the test. Otherwise, it leaves the test alone, and prepasses the THEN and ELSE arms recursively. As an optimization, PREPASS also checks whether an IF test will always return NIL. If so, it simplifies the IF to its prepassed ELSE arm. Likewise for the THEN arm if it trivially sees a test returning non-NIL. The inference algorithm performs IF-normalization prior to processing function guards and bodies. Though IF-normalization does not affect the value produced by evaluation of the IF, it can make the difference in deciding whether a form can be treated as a type predicate.

The inference algorithm gives special consideration to forms which it considers to be type predicates, and in particular to forms which conform to the requirements of a recognizer function body. The algorithm exploits the properties of type predicates to make simplifying assumptions which allow different techniques to be used in formulating signatures. These techniques can produce very specific signatures, so it is worthwhile to apply them whenever possible. One of the requirements in recognizer bodies is that IF tests always yield Boolean results.

A recognizer function in essence is a unary function which determines whether its argument conforms to a descriptor in the type language. For any variable-free type descriptor, one could define a perfect recognizer function. One of the criteria for a recognizer function is that it always returns either T or NIL.

One reason recognizers are important to the type system is that if a call to a recognizer is used as a test in an IF expression, optimal modifications to the type context of the IF may be made to support the analysis of the THEN and the ELSE arms. Moreover, this significance scales up, since one criterion for the body of a recognizer is that its IF tests are recognizer calls or equivalent predicates. IF-normalization, then, can help transform a function body into a form suitable for treatment as a recognizer.

An example of a transformation performed by PREPASS is the form

```
(IF (CDR X) (FOO X) (BAR X))
```

The tool does not treat (CDR X) as a type predicate, since it can return a non-Boolean value. Therefore, the algorithm will not optimally refine the type assumptions about X when considering (FOO X) and (BAR X). However, for heuristic reasons, it will treat

```
(IF (NULL (NULL (CDR X))) (FOO X) (BAR X))
```

more generously. So PREPASS makes this transformation.

4.4.4 TYPE-PREDICATE-P

The function TYPE-PREDICATE-P is the principal function which determines whether a function qualifies as a recognizer and which computes the descriptor for the objects it recognizes. It is also used to derive a guard descriptor from a function guard and to examine IF tests for type information which can be merged into the type contexts for the THEN and ELSE arms.

Given a Lisp expression FORM, a list of names of formal parameters the function ARGLIST, a list of known RECOGNIZERS from the system database, and a TYPE-ALIST which associates variable names to descriptors characterizing their types in the environment of FORM, this complex function returns either NIL or a five-tuple:

```
(ALIST NEGATABLE UNSATISFIED-GUARDS BOOLEANP COMPLETE)
```

where

- ALIST is of the same form as the argument TYPE-ALIST and conveys the type information provided when FORM evaluates to a non-NIL value. For example, given the form

  ```
  (IF (INTEGERP X) (CHARACTERP Y) NIL)
  ```

  TYPE-PREDICATE-P yields the ALIST

  ```
  ((X . $INTEGER) (Y . $CHARACTER))
  ```

- NEGATABLE is either T or NIL and indicates whether the type information in ALIST can be negated if the FORM evaluates to NIL. In the case where FORM is an IF test we might wish to negate its information in the else arm. This is not always possible, with one trivial example being (EQUAL X "abc"). The ALIST produced for this form is ((X . $STRING)). But it is not true that the negation of FORM indicates that X is not a string. It could be a non-string or any string except "abc". Therefore, we say that the ALIST for this form is not negatable. For another example, (IF (INTEGERP X) (NULL Y) NIL) produces the ALIST ((X . $INTEGER) (Y . $NIL)). This is not negatable, since the type language does not have a notion of negating an ALIST.[17]

---

[17]In practice, what we do not support is negation of an ALIST of length other than one. This amounts to a shortcoming in the implementation, since it means, for example, that we cannot provide additional type information in the ELSE arm of an IF where a form like (IF (INTEGERP X) (NULL Y) NIL) was the test. A nice upgrade would be to express the result as a disjunctive normal form, so we could accommodate negation.

- UNSATISFIED-GUARDS is either T or NIL and indicates whether some guard violation in FORM was detected. This is not a complete test; it is only a fast one. A complete test is performed elsewhere. However, if NEGATABLE is T and UNSATISFIED-GUARDS is NIL, we know there are no guard violations in the form since terms which satisfy the negatable constraints are syntactically restricted to ensure that the guards of all subexpressions can be checked definitively. Specifically, this involves testing the guards on calls to CAR and CDR, where the parameter is some component of one of the variables in TYPE-ALIST.
- BOOLEANP is T if FORM can evaluate only to T or NIL, NIL otherwise.
- COMPLETE is T if when FORM evaluates to a non-NIL value, the ALIST completely captures what could be deduced from the variables in ALIST. For example, the form (INTEGERP X) is complete because the ALIST will be ((X . $INTEGER)). But (EQUAL X 3) is not complete. Though it produces the same ALIST as (INTEGERP X), the knowledge captured by the ALIST is not enough to guarantee the FORM would evaluate to T (i.e., X could have the value 4). On the other hand, (IF (INTEGERP X) (NULL Y) NIL) is complete, even though it is not negatable, since there is no way any values which conform to the resulting ALIST can cause the form to evaluate to NIL.

TYPE-PREDICATE-P returns NIL if there is no type information about the identifiers which can be garnered from the evaluation of the form. Examples of forms which are not type predicates are (BINARY-+ X Y) and (IF (INTEGERP X) (BINARY-+ X X) X)

Let us now examine how TYPE-PREDICATE-P makes its judgements, using examples to illustrate. These examples will give the flavor of the heuristics used. This is just a sampling, which places emphasis on where the algorithm yields information. Often, it is just as interesting why it fails to yield information, but we will not dwell on that. The notation

```
<form> <arglist>
   -> NIL or <ALIST NEGATABLE UNSATISFIED-GUARDS BOOLEANP COMPLETE>
```

will indicate that

```
(TYPE-PREDICATE-P <form> <arglist> RECOGNIZERS TYPE-ALIST)
```

produces NIL or a 5-tuple, and follow with an explanation if warranted. The TYPE-ALIST argument is significant only in determining the presence of guard violations. <arglist>s will be given by example or omission, to avoid copious notation. Assume any identifiers X, Y, ... appearing in the form are in the arglist. In expressing these schema, we will assume no guard violations are detected. These rules are presented as canonical examples, so functions like INTEGERP, CONSP, and TRUE-LISTP are used to exemplify treatment of recognizer functions, and FOO represents a non-recognizer.

```
T (X) -> (((X . *UNIVERSAL)) T NIL T T))
NIL (X) -> (((X . *EMPTY)) T NIL T T)
T (X Y) -> (((X . *UNIVERSAL) (Y . *UNIVERSAL)) NIL NIL T T)
NIL (X Y) -> (((X . *EMPTY) (Y . *EMPTY)) NIL NIL T T)
```

T recognizes all objects, NIL rejects all objects. If the arglist is of length greater than one, the result is not negatable.

```
<atom> (..) -> NIL
```

Atoms other than T or NIL are not type predicates.

```
(EQUAL (CDR X) T) (..) -> (((X . (*CONS *UNIVERSAL $T))) T NIL T T)

(EQUAL '(#\A) (CAR X)) (..) ->
  (((X . (*CONS (*CONS $CHARACTER $NIL) *UNIVERSAL))) NIL NIL T NIL)
```

When the form is an EQUAL, one argument is some nest of destructors (CAR or CDR) applied to a parameter from ARGLIST, and the other argument is a literal (a quoted form or an constant of one of the primitive types), we build the *CONS structure appropriate to the destructors and embed the descriptor for the literal. Under the assumption that there is no guard violation, i.e., when the type-alist guarantees the *CONS structure is adequate, the resulting descriptor is negatable if the literal is T or NIL, otherwise not.

Note the following curiosity. Whereas the (EQUAL (CDR X) T) example above yielded (X . (*CONS *UNIVERSAL $T)), the same predicate, using NIL instead of T, yields

```
(EQUAL (CDR X) NIL)) (..) ->
  (((X *OR $NIL (*CONS *UNIVERSAL $NIL))) T NIL T T)
```

This is, of course, because (CDR X) is NIL when X is NIL or X is a CONS whose CDR is NIL.

```
(INTEGERP X) (..) -> (((X . $INTEGER)) T NIL T T)
(INTEGERP (CDR X)) -> (((X . (*CONS *UNIVERSAL $INTEGER))) T NIL T T)
```

If the form is a call to a recognizer function on some component of a formal, we build the *CONS structure appropriate to the destructors and embed the descriptor associated with the recognizer function. The descriptor is negatable when there is no guard violation found because recognizer calls produce negatable results. Recognizers always return Boolean values and are complete.

```
(NOT (INTEGERP X)) ->
  (((X . (*OR $CHARACTER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
              $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
   T NIL T T)

(NULL (NULL (CDR X))) ->
  (((X . (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
              $STRING $T
              (*CONS *UNIVERSAL
                     (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                          $NON-T-NIL-SYMBOL $STRING $T
                          (*CONS *UNIVERSAL *UNIVERSAL))))))
   T NIL T T)
```

We give special treatment to the NULL function and its equivalents (NOT, for instance). In these cases, like the one above, we do not require the argument to be a series of destructors applied to a formal. It is sufficient for the argument to be a negatable type predicate. If it is, we can just negate the descriptor for the nested form. If it is not (in a case like (NULL (CDR X)), we can still try the general test for recognizers, given below. This, in fact, is the base case for applications of a series of NULL surrogates to a destructed formal. This special treatment may seem like an arcane special case, but remember that PREPASS doubly negates expressions appearing in IF tests if they are not recognizer calls.

It may seem strange that the descriptors $CHARACTER $INTEGER, $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL, $STRING, and $T are included as disjuncts in the result above, since (CDR X) is not well-defined for objects of these types. However, the information in the result ALIST is intended to be merged via unification with that in the TYPE-ALIST argument, and thus if the TYPE-ALIST is such that X is NIL or a CONS, the merge will rule out the unruly cases. If not, the unsatisfied-guards flag in the result will be T.

When we encounter an IF form, we recur on its test and on its THEN and ELSE arms. If the test was a type predicate, we merge its results into the type-alist via unification for the recursive call on the THEN arm. If the test was also negatable, we negate it and merge for the ELSE arm. Then we try a number of special cases, as follows:

```
(IF (IF (CONSP X) (CONSP Y) NIL) T (INTEGERP X)) ->
  (((X . (*OR $INTEGER (*CONS *UNIVERSAL *UNIVERSAL)))) NIL NIL T NIL)
```

If evaluating this IF form returns T by way of the THEN arm, we know both X and Y are CONSes, but if we return T by way of the ELSE arm, we know only that X is an integer. So if this form returns non-NIL, we know something about X, but not necessarily anything about Y. Moreover, since this form may return NIL when X is a CONS but Y is not, we cannot negate what little we know. And since we capture only information about X, but a non-NIL result may require something of Y, we cannot say the result is complete, either.

```
(IF (CONSP X) (TRUE-LISTP X) T) ->
   (((X . (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
               $NON-T-NIL-SYMBOL $STRING $T
               (*CONS *UNIVERSAL
                      (*REC TRUE-LISTP
                            (*OR $NIL (*CONS *UNIVERSAL
                                             (*RECUR TRUE-LISTP)))))))
     T NIL T T)
```

Evaluation of this form will succeed whenever X is a non-CONS or it is both a CONS and a TRUE-LISTP, and fail otherwise, which makes it negatable and complete. The result descriptor for X is the *OR of the ones yielded by the two recognizer calls.

```
(IF (IF (CONSP X) (CONSP Y) NIL) (TRUE-LISTP X) T) -> NIL
```

This could succeed solely because Y is a non-CONS, but it could also succeed for reasons not relating to Y. We deduce nothing.

```
(IF (CONSP X) (INTEGERP Y) NIL) ->
  (((X . (*CONS *UNIVERSAL *UNIVERSAL)) (Y . $INTEGER)) NIL NIL T T)
```

This will succeed only when X is a CONS and Y is an integer. However, it can fail if either is not true, so we cannot negate the result. The result is complete, however, because the alist is sufficient to guarantee a positive result.

```
(IF (CONSP X) (FOO X) NIL) ->
  (((X . (*CONS *UNIVERSAL *UNIVERSAL))) NIL NIL NIL NIL)
```

Here, suppose that what FOO returns yields no particular type information about X. This can only succeed if X is a CONS, but it can fail for any reason. So, the result is neither negatable nor complete.

```
(IF (CONSP X) NIL (IF (INTEGERP X) (INTEGERP Y) NIL)) ->
  (((X . $INTEGER) (Y . $INTEGER)) NIL NIL T T)
```

This form returns a non-NIL value only if X and Y are integers. Moreover, this is complete. But we know nothing if we fail because we could fail either when X is a non-integer or Y is a non-integer, and we do not have a means for negating an alist.

```
(IF (NOT (CONSP X)) NIL (IF (INTEGERP (CAR X)) (INTEGERP Y) NIL)) ->
  (((X . (*CONS $INTEGER *UNIVERSAL)) (Y . $INTEGER)) NIL NIL T T)

(IF (CONSP X) NIL (FOO X)) ->
  (((X . (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
              $NON-T-NIL-SYMBOL $STRING $T)))
   NIL NIL NIL NIL)
```

If this form returns non-NIL, we know the negatable IF test failed, but we know nothing more.

```
(IF (CONSP X)
```

```
    (IF (INTEGERP (CAR X)) (INTEGERP Y) NIL)
    (IF (CHARACTERP X) (CHARACTERP Z) NIL)) ->
(((X . (*OR $CHARACTER (*CONS $INTEGER *UNIVERSAL)))) NIL NIL T NIL)
```

If this form returns non-NIL, we know that X is either a character or a CONS whose CAR is an integer, but we know nothing about Y or Z. We could succeed when Y is an integer or when Z is a character, but we cannot relate these facts in a type alist.

```
(IF (EQUAL (CAR X) 3) (CONSP (CDR X)) (INTEGERP (CAR (CDR X)))) ->
  (((X . (*CONS $INTEGER (*CONS *UNIVERSAL *UNIVERSAL)))) NIL NIL T NIL)
```

If our IF test form is not negatable, then we do not try to merge any information from the ELSE arm. We cannot negate our descriptor.

```
(IF (FOO X) (CONSP X) NIL) ->
  (((X . (*CONS *UNIVERSAL *UNIVERSAL))) NIL NIL T NIL)
```

FOO gives us no type information, but since the ELSE is NIL, the only way we can succeed is if X is a CONS. We do not try to negate, since FOO alone could cause the result to be NIL.

4.4.5 RECOGNIZERP and DESCRIPTOR-FROM-FNDEF

RECOGNIZERP is the predicate which says whether a function may be classified as a recognizer. It is essentially a combination of a call to TYPE-PREDICATE-P on the body of a function and a set of additional checks. A recognizer must have an argument list of length one, and it must either have no guard or a guard of T. Its body must be a type predicate, i.e., TYPE-PREDICATE-P must return an alist. Furthermore, the alist must be of length one and be negatable. There may be no guard violations. Finally, the result must be Boolean (T or NIL). If a function passes these constraints, we know that the check for guard violations in TYPE-PREDICATE-P is strong enough to guarantee the guards of all called functions are verified. If all this is true, RECOGNIZERP returns the descriptor for its parameter from the ALIST from TYPE-PREDICATE-P.

RECOGNIZERP sets up its call to TYPE-PREDICATE-P in a way which can produce a result which is a little irregular in the type language. TYPE-PREDICATE-P looks up function names in the RECOGNIZERS table to determine when a called function is a recognizer. If it is, TYPE-PREDICATE-P extracts from the table the descriptor associated with the function. RECOGNIZERP allows TYPE-PREDICATE-P to assume the function whose body it is analyzing is a recognizer, since, if every other attribute of the function body is consistent with the notion of a recognizer, the recursive call will be, too. But, since we are only in the process of computing the descriptor for the function, what we can store as the descriptor characterizing the recognizer is limited. In fact, we store the partial descriptor (*RECUR <fnname>), which is the recursion point in a *REC descriptor labelled <fnname>. Thus, when we encounter a recursive function call in the body, we embed this *RECUR form at the appropriate point in the descriptor being composed by TYPE-PREDICATE-P. Upon exit from TYPE-PREDICATE-P, we might have a descriptor with a *RECUR point, but no encapsulating *REC. For example, when analyzing the function TRUE-LISTP:

```
(DEFUN TRUE-LISTP (X)
  (IF (NULL X) (NULL X) (IF (CONSP X) (TRUE-LISTP (CDR X)) NIL)))
```

we return from TYPE-PREDICATE-P with the descriptor

```
(*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))).
```

We close this form by simply wrapping *REC around it with the TRUE-LISTP label:

```
(*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))).
```

But there may be another quirk. Consider the function:

```
(DEFUN FOO (X)
  (IF (NULL X)
      (NULL X)
      (IF (CONSP X)
          (IF (CONSP (CDR X)) (FOO (CDR X)) NIL)
          NIL)))
```

The (CONSP (CDR X)) test governing the recursive call yields type information about the recursive parameter which amounts to a constraint on what the recursive call will recognize. Normally, TYPE-PREDICATE-P would use descriptor unification to merge the information from the IF test

```
((X . (*CONS *UNIVERSAL (*CONS *UNIVERSAL *UNIVERSAL))))
```

with the descriptor from the THEN arm:

```
((X . (*CONS *UNIVERSAL (*RECUR FOO))))
```

But (*RECUR FOO) is not meaningful in the absense of an enclosing *REC form, and at the point where we would do this unification, we are not yet ready to encapsulate with a *REC.

The answer to this quandary is to defer the merge until we do have a *REC form. We allow TYPE-PREDICATE-P to construct a special form which indicates a deferred unification.[18] We call this form *AND. TYPE-PREDICATE-P will produce the following alist entry for the body of foo.

```
(X . (*OR $NIL
          (*CONS *UNIVERSAL
                 (*AND (*CONS *UNIVERSAL *UNIVERSAL) (*RECUR FOO)))))
```

RECOGNIZERP will return this *OR descriptor.

It is up to the top level function for discovering recognizers, DESCRIPTOR-FROM-FNDEF, to complete the closure of this form. First, if it receives a form with a *RECUR point for the function being analyzed, it does the *REC encapsulation as previously mentioned. Then it proceeds with what we call **AND validation*, i.e., determining if the constraint (in this case (*CONS *UNIVERSAL *UNIVERSAL)) is consistent with the opened up recursive descriptor, and formulating a refined descriptor to reflect the restrictions which the constraint and the *REC place on each other.

The essence of *AND validation is to unify the constraint with the unwinding of the *RECUR form, and to let this result take the place of the *AND form in the *REC descriptor. But this is a deceptive simplification of the task, for a number of reasons. For one thing, the constraint may close off the recursion, so we have to be prepared to canonicalize our "*REC" form to a non-recursive one. (In fact, this is what will happen with our FOO example above.) But more problematically, the unified result may itself have an *AND form in it from expanding the *RECUR to the containing *REC, which of course has an *AND nested in it. Furthermore, the unified form or its constraint may be different than before, the result of the constraint actually restricting the possible values from the unfolded *RECUR. Thus, we have a stabilization problem.

---

[18]Actually, the form is constructed in UNIFY-DESCRIPTORS when we attempt to unify a *RECUR form with something else, a scenario which would never occur if the *RECUR were nested in a *REC.

The process is essentially as follows. If the unification produces a descriptor which is just the recursive expansion of the one we started with, we contract the expansion, stripping the *AND constraint, and we are done. If the unification produces something different, then if the result contains an *AND, we *AND-validate it again, in hopes of finding a new *REC descriptor to embed within the expansion of the old one. If there is no *AND in the unified form, we are done, and we just embed the result in the form where the *AND was originally.

Here are some examples of calls to VALIDATE-AND. The first is from our FOO example above. Here the constraint can only be fulfilled if the recursion does not terminate. Therefore, the *AND form collapses to *EMPTY, and so does its containing *CONS. Since $NIL includes no (*RECUR FOO), we drop the *REC, leaving only $NIL.

```
(VALIDATE-AND
  '(*REC FOO (*OR $NIL
                  (*CONS *UNIVERSAL
                         (*AND (*CONS *UNIVERSAL *UNIVERSAL)
                               (*RECUR FOO)))))
  'FOO)
 =
$NIL
```

(VALIDATE-AND's second argument is just a coding artifact of no interest in this discussion.) In the following, the constraint does not allow NIL, so we force the $NIL outside the recursive form.

```
(VALIDATE-AND '(*REC FOO
                (*OR $INTEGER $NIL
                 (*CONS $INTEGER
                  (*AND (*OR $INTEGER (*CONS *UNIVERSAL *UNIVERSAL))
                        (*RECUR FOO)))))
              'FOO)
 =
(*OR $NIL (*REC !REC1 (*OR $INTEGER (*CONS $INTEGER (*RECUR !REC1)))))
```

In the next example, the *CONS constraint is disallowed, but the $INTEGER is valid, albeit non-recursive.

```
(VALIDATE-AND '(*REC FOO
                (*OR $INTEGER $NIL
                 (*CONS $INTEGER
                  (*OR (*AND $INTEGER (*RECUR FOO))
                   (*AND (*CONS *UNIVERSAL (*CONS *UNIVERSAL *UNIVERSAL))
                         (*RECUR FOO))))))
              'FOO)
 =
(*OR $INTEGER $NIL (*CONS $INTEGER $INTEGER))
```

The processing of *AND forms makes possible the accurate signification of many ordinary functions. For instance, the RECOGNIZERP function applied to:

```
(DEFUN PROPER-CONSP (X)
  (IF (CONSP X)
      (IF (CONSP (CDR X)) (PROPER-CONSP (CDR X)) (EQUAL (CDR X) NIL))
      NIL))
```

yields the descriptor

```
(*CONS *UNIVERSAL
       (*OR $NIL (*AND (*CONS *UNIVERSAL *UNIVERSAL)
                       (*RECUR PROPER-CONSP))))
```

*AND validation of this descriptor produces:

```
(*REC !REC1 (*CONS *UNIVERSAL (*OR $NIL (*RECUR !REC1))))
```

which fully characterizes the objects recognized by PROPER-CONSP.

4.4.6 DERIVE-EQUATIONS

Perhaps a better name for this function would have been SCHEMATIZE-BODY. DERIVE-EQUATIONS transforms the body of the function into a table, with one entry in the table for each subexpression in the body. Each entry is of the form

```
(<form> <marker> <type alist> <method>)
```

where <form> is the Lisp form being analyzed, the <marker> is a gensym used for table lookup, the <type alist> characterizes what we know about the types of the variables in the context of the subexpression, and the method is a schema which provides direction in computing the type of the subexpression.

We perform some type inference computations while constructing this table. Whenever an IF is encountered, we extract what type information we can from the predicate, using TYPE-PREDICATE-P, and merge it into distinct type alists for the THEN and ELSE arms, utilizing UNIFY-DESCRIPTORS. Of course, we can merge into the ELSE arm only if TYPE-PREDICATE-P tells us its type alist is negatable.

The methods are schematic aids to the computation of the result type for each expression. They vary for different kinds of expressions. For identifiers, the method is simply the type descriptor for the variable in the type-alist. For quoted forms, the method is the descriptor directly derived from the form, using the function DESCRIPTOR-FROM-QUOTED-FORM. For IF forms, the method is a list of length two which contains the index markers pointing to the table entries for the forms in the THEN arm and the ELSE arm of the IF. For function calls, the method is a list of length two whose first element is a list of index markers pointing to the entries for its actual argument forms, and whose second element is a new type variable which will represent the result type for the function call when we "solve the equation" represented by this table entry.

Consider the following function:

```
(NEW-TOP-LEVEL-FORM '(DEFUN LAST (X)
                       (IF (ATOM X) X (LAST (CDR X)))))
```

DERIVE-EQUATIONS produces the following table representing the body of this function.

```
foo

((X *MARKER-2 ((X . &5)) &5)
 ((ATOM X) *MARKER-3 ((X . &5)) ((*MARKER-2) &6))
 (X *MARKER-4
    ((X *OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $STRING $T))
    (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
         $NON-T-NIL-SYMBOL $STRING $T))
 (X *MARKER-5
    ((X *CONS *UNIVERSAL *UNIVERSAL))
    (*CONS *UNIVERSAL *UNIVERSAL))
 ((CDR X) *MARKER-6
          ((X *CONS *UNIVERSAL *UNIVERSAL))
          ((*MARKER-5) &7))
 ((LAST (CDR X)) *MARKER-7
```

```
                  ((X *CONS *UNIVERSAL *UNIVERSAL))
                  ((*MARKER-6) &8))
 ((IF (ATOM X) X (LAST (CDR X))) *MARKER-8
                                  ((X . &5))
                                  (*MARKER-4 *MARKER-7)))
```

Each subexpression has a unique marker. Notice that the type-alist for the second occurrence of X (the one marked with *MARKER-4) maps X to the descriptor

```
(*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
     $NON-T-NIL-SYMBOL $STRING $T)
```

rather than the variable in the outermost type-alist. This is because this occurrence of X is governed by the (ATOM X) test with a positive result. Thus, X is an atom, and we can rule out any possibility of it being a *CONS. The opposite is true in the third occurrence (*MARKER-5). Since we are in the ELSE arm, we know the atom test failed, so the type-alist entry for X has been refined to be (*CONS *UNIVERSAL *UNIVERSAL).

Notice also that the method for the first X, &4, is just the type-alist entry for X. In the method for (ATOM X), i.e., ((*MARKER-2) &5), *MARKER-2 points to the argument, which is the X from the first table entry. &5 is a new variable which will represent the type of (ATOM X). In the method for the IF form, (*MARKER-4 *MARKER-7), *MARKER-4 points to the table entry for the THEN arm, X, and *MARKER-7 to the entry for the ELSE arm, (LAST (CDR X)).

When we discuss SOLVE-EQUATIONS below, we shall see how this table is employed as a basis for the type computation.

4.4.7 SOLVE-EQUATIONS

First, let us introduce the parameters supplied to SOLVE-EQUATIONS.

```
EQUATIONS -- The table generated by DERIVE-EQUATIONS

FNNAME -- The name of the function being analyzed

GUARD-DESCRIPTORS -- The guard descriptors for the function being
  analyzed, computed by INFER-SIGNATURE from the guard expression
  using TYPE-PREDICATE-P

WORKING-SEGMENTS -- A list of segments representing the current
  approximation of the signature segments for this function,
  initially NIL

FUNCTION-SIGNATURES -- The main component of the system database

UNARY-DESTRUCTOR-FN-MAP -- A table which decomposes various unary
  destructor functions (e.g., CADDR) into CARs and CDRs

ARGLIST -- The formal argument list for the current function

CONSTANT-DESCRIPTORS -- A component of the database which holds
  the descriptors of defined constants (But in the current
  implementation of the system, constants are not accepted.
  Rather, they should be submitted as functions with no arguments.)

RECURSION-COUNT -- The maximum number of iterations allowed before
  the algorithm gives up in its quest for stabilization

NO-RECURSIVE-CALLS -- A flag which takes the value NIL when the
  function is recursive, T if non-recursive
```

The highest level view of SOLVE-EQUATIONS is as follows. We traverse the table of EQUATIONS, computing the type(s) of each subexpression. For each subexpression, we produce a list of "segments", where a segment maps the types of the variables in the environment to a result type. The form of a segment is:

```
(<desc1> .. <descn>) -> <desc>
```

where <desc1> .. <descn> represent the types of the variables at this point (as opposed to the types of actual parameters in the subexpression function call), and <desc> represents a possible type of the subexpression, assuming the variable types. There can be more than one segment for a given subexpression because different combinations of variable types may produce different result types. Ultimately, the set of segments for the outermost subexpression, i.e., the function body, will become the segments of the signature for the function.

As SOLVE-EQUATIONS passes through the equations, it builds a table of results, one entry for each equation. An entry in this table is the marker for the equation CONS-ed onto the segments computed for the equation. The markers serve as keys for looking up previously computed segments. Let us call this table PARTIAL-SOLUTION.

Recall the form of an equation, given in the previous section. How we solve an equation depends on the method stored as a component of the equation. Some cases are easy. If the expression is a variable, the method is simply the descriptor for the variable in the type-alist saved in the equation. The segment produced simply maps the variable types from the type alist to the descriptor for this variable, which was conveniently stored as the method. There is one exception to this rule. If the descriptor for the variable is an *OR descriptor, we produce one segment for each disjunct of the *OR. This gives us much greater specificity in our results. Thus, if we encounter a reference to Y in a context where our type-alist is

```
((X . (*OR $INTEGER $NIL)) (Y . &1))
```

we create the segment:

```
((*OR $INTEGER $NIL) &1) -> &1
```

But if the reference is to X, we get the two segments:

```
($INTEGER &1) -> $INTEGER
($NIL &1) -> $NIL
```

If the expression is a literal (a quoted form or a base value from a primitive type), the method is the descriptor describing the literal. SOLVE-EQUATIONS produces a segment mapping the argment descriptors to the literal descriptor.

If the expression is an IF expression, the method is a list of two markers. The first marker points to the segments for the THEN arm in PARTIAL-SOLUTION, and the second marker points to the segments for the ELSE arm. These, of course, have already been computed because the table is in Lisp evaluation order (as if IF were strict). The segments for the IF are simply the appended segments for the THEN and ELSE.

The most interesting case is the function call. We have already computed the segments for each of its arguments. We look these segments up by using the markers in the method for this function call. Recall, a function call method is a list of length two whose CAR is a list of the same arity as the parameter list for the function being called, where the elements of the list are markers pointing to the segments for each actual argument in the PARTIAL-SOLUTION. We need to formulate all the possible distinct type-alists provided by these segments. This is essentially a cross-product operation, which is necessary because the contexts in the various segments do not necessarily match. For example, where there are two variables, X and Y, in the environment, the segments for the first argument might be:

```
(((*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
  (*REC INTEGER-LISTP
        (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> $INTEGER)
(($NIL
   (*REC INTEGER-LISTP
         (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> $NIL)
```

(Read this to say that if X is a non-empty integer list and Y is an integer list, the first argument may be an integer. If X is NIL and Y is an integer list, the first argument may be NIL.) For the second argument, we might have:

```
(((*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
  (*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
   -> $INTEGER)
(((*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
  $NIL)
   -> $NIL)
```

We need to factor these segments into one another to get the distinct type environments which will produce the various argument patterns to our function call. As usual, UNIFY-DESCRIPTORS is the combinator which is used to merge and factor the environments. Each of the contexts from the segments for the first actual argument are unified pairwise with each of the contexts from the segments for the second argument. The result of each such unification is paired with the list formed by the respective result descriptors of the original segments, to form what we will call an *arg-solution*. Each arg-solution maps a context of type descriptors for the variables in the environment to the types computed for the actual parameters of the function call. In our example, the environments, and the arg-solutions they produce, are:

```
(((*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
  (*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
  -> ($INTEGER $INTEGER))

(((*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
  $NIL)
 -> ($INTEGER $NIL))

(($NIL
   (*CONS $INTEGER
          (*REC INTEGER-LISTP
                (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
 -> ($NIL $INTEGER))

(($NIL $NIL) -> ($NIL $NIL))
```

Collectively, these arg-solutions represent all the type scenarios to be treated on the function call.

Now that we know which argument patterns we will submit to our function, we are ready to see what result each may yield. We take each arg-solution in sequence, performing the process described below.

The first task is to see if we can detect the possibility of a guard violation on the called function for our argument pattern. Another way to phrase the question is to see whether the guard would further constrain what we know about our parameters. If it would, then we will judge we have a guard violation. Following this line of reasoning, we note that descriptor unification is a constraining operation. We apply UNIFY-DESCRIPTORS to the function guard, fetched from the database, and our argument pattern. If the resulting descriptor is identical to our original argument pattern, then we have satisfied the guard. Since we dwell in a world including type variables, we do not require the result be identical, only that it be isomorphic, i.e., identical modulo a 1-1 and onto substitution of variables for variables. If UNIFY-DESCRIPTORS returns an *OR of possible unified forms, each must be isomorphic. If we do not have this isomorphism, we declare a guard violation and halt the computation.[19]

Recall once again that the fact we declare a guard violation does not necessarily mean the violation can occur. It merely means that the inference system cannot guarantee that it will not. Some other kind of proof may be in order to verify the guard. Conversely, the fact that we do not signal a guard violation does not mean that one will not occur. Our claim that there is no guard violation is only solid when the guard descriptor we have stored for the called function is complete. Even this claim is qualified. Since this algorithm is not formalized in any sense, we cannot make any formal argument to back up the claim.

With guard verification out of the way for our arg-solution, we can proceed to derive its possible result types. The second element in the method for our equation was a type variable which represents the result type of the function call. We construct a list of descriptors consisting of the argument descriptors from our arg-solution with our result variable added to the end. Then we fetch the segments for the function being called and massage each segment into a similar list, replacing all variables in the segment with fresh ones. We pair each of our arg-solution vectors with each of the segment vectors, applying UNIFY-DESCRIPTORS. UNIFY-DESCRIPTORS returns *EMPTY if an argument pattern will not unify with the descriptors representing the formal arguments for any segment. This means simply that the segment is irrelevant to our pattern, indicating that it corresponds to a control path through the called function which will not be followed on arguments of the type we are supplying. If we get a non-*EMPTY result, however, our result variable will have been bound to a type descriptor which constitutes a projection of our arguments through the segment to its result. For example, consider calling the function CAR with an argument of type

```
(*CONS (*OR $INTEGER $NON-INTEGER-RATIONAL) *UNIVERSAL)
```

Let us suppose our result variable is &2. The signature segments for CAR are:

```
($NIL) -> $NIL
(*CONS &1 *UNIVERSAL) -> &1
```

When we unify our parameter-result pattern:

```
((*CONS (*OR $INTEGER $NON-INTEGER-RATIONAL) *UNIVERSAL) &2)
```

with ($NIL $NIL), we get *EMPTY, because $NIL will not unify with our *CONS. But in the second segment, unifying against

[19] This isomorphism test is somewhat awkward. A more natural formulation would result from extending the containment algorithm used in the signature checker to support the full language of type descriptors used in the inference algorithm.

```
((*CONS &1 *UNIVERSAL) &1)
```

we get the result

```
((*CONS (*OR $INTEGER $NON-INTEGER-RATIONAL) *UNIVERSAL)
 (*OR $INTEGER $NON-INTEGER-RATIONAL))
```

with (*OR $INTEGER $NON-INTEGER-RATIONAL) having been substituted for our result variable &2. Thus, we know that our call to CAR can have a result of type (*OR $INTEGER $NON-INTEGER-RATIONAL). We construct a segment which maps the types of the variables in our current environment, i.e., the types from the first part of our arg-solution, to our computed result type. We get one such segment for each segment we match from the called function's signature (in this case, there was just one match). We perform this operation for each of the arg-solutions from our original list. Finally, we tag our collection of segments with the marker for this subexpression, performing some canonicalization to reduce the number of segments, and add the result to PARTIAL-SOLUTION, for later reference.

Of course, things are not as simple as in the case just described. There is the question of dealing with recursive function calls, which we will address later. But there is another significant detail to add to the account of the non-recursive call. It can be illuminated by a variation on our previous example. Suppose in our environment, the variable X was a list of integers, i.e.,

```
(*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
```

For simplicity, let us say X is the only variable in the environment, and we are examining the call (CAR X). Our arg solution, from the X subexpression entry in PARTIAL-SOLUTION, is just:

```
(((*REC INTEGER-LISTP
        (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
 (*REC INTEGER-LISTP
       (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
```

i.e, in an environment where X is an integer list, the expression X produces an integer list. The guard for CAR,

```
(*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL))
```

is satisfied. Now, we unify against the segments for CAR. Both match. If we simply followed the procedure just explained, we would wind up with the segments:

```
((*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> $NIL
```

and

```
((*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> $INTEGER
```

Clearly, this is not optimal, as the real result will be NIL only if X is NIL, and will be an integer only if X is a non-empty integer list. We would rather have the segments:

```
($NIL) -> $NIL
((*CONS $INTEGER
        (*REC INTEGER-LISTP
              (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
   -> $INTEGER
```

We can think of this as being a kind of feedback from the argument expressions to the type binding environment for our variables. This feedback does not always occur. The arguments to a function call may or may not be variables (or variable components). They may just be some expression which is computed from some other expression, etc., with the variables appearing deeply buried, if at all. The problem is determining whether and how a match of different segments for the called function reflects a type distinction in a local variable. We can examine the subexpression arguments and determine some circumstances when such a split on segments clearly does reflect this distinction. Specifically, we give special treatment to arguments which are variables or components of variables, accessed by a nest of applications of CAR and CDR to the variable.

A previously mentioned feature of UNIFY-DESCRIPTORS is the key. Recall, a restrictions list is one of the forms returned by UNIFY-DESCRIPTORS. If a form within one of the input descriptors is marked with an annotation which associates it with a type variable, and if that subform is restricted in the course of unification, then an entry is made in the restrictions list which maps the variable to its restricted form. For example, if we evaluate:

```
(UNIFY-DESCRIPTORS '(*CONS *UNIVERSAL (*SUBST &1 (*OR $INTEGER $NIL)))
                   '(*CONS $INTEGER $NIL)
                   T)
```

we get the result:

```
((*CONS $INTEGER (*SUBST &1 $NIL)) NIL ((&1 . $NIL)))
```

where ((&1 . $NIL)) is our restriction list. This signifies that (*OR $INTEGER $NIL), which was annotated as being associated with &1, was restricted to $NIL.

We exploit this feature when matching segments by annotating, with a *SUBST employing a fresh type variable, any descriptor denoting the type of a Lisp variable or a variable component. Furthermore, we maintain a data structure which enables us to locate, within the descriptor for the variable, which component of the variable was passed in the parameter list. Then, if the unification returns with a restriction on that variable, when we form the segment mapping our variable types to our result type, we replace the appropriate component of the variable type descriptor with the restriction imposed from the unification. This enables us, in the example which yielded imprecise results with the naive implementation, to formulate exactly the segments we desire.

As currently implemented, this technique only works with nests of applications of CAR and CDR.[20] We can think of these functions as being "destructors". A nice upgrade to the implementation would be to develop a generalized notion of destructor functions, so that new ones can be added to the class as they are defined and, in turn, receive the benefit of this hook in the implementation.

To illustrate this discussion, let us consider a simple example:

```
(DEFUN FEEDBACK (X Y)
  (DECLARE (XARGS :GUARD (IF (INTEGER-LISTP X) (INTEGER-LISTP Y) NIL)))
  (BAR (CAR X) (CAR Y)))
```

Assume the signature for BAR has the segments:

[20] Actually it works with a collection of functions defined in an extended base of Common Lisp, such as CADR, CDDR, etc., and FIRST, SECOND, etc. which are defined to be CAR and CDR nests. This collection is passed to SOLVE-EQUATIONS as the formal parameter UNARY-DESTRUCTOR-FN-MAP.

```
((*OR $INTEGER *NON-INTEGER-RATIONAL)
 (*OR $INTEGER *NON-INTEGER-RATIONAL))
  -> (*OR $INTEGER *NON-INTEGER-RATIONAL)

($NIL (*OR $INTEGER *NON-INTEGER-RATIONAL)) -> $NIL

((*OR $INTEGER *NON-INTEGER-RATIONAL) $NIL) -> $NIL

($NIL $NIL) -> $INTEGER
```

and the guard for BAR, which we will assume is complete, is:

```
((*OR $INTEGER $NIL *NON-INTEGER-RATIONAL)
 (*OR $INTEGER $NIL *NON-INTEGER-RATIONAL))
```

We know the signature for CAR has segments:

```
($NIL) -> $NIL
(*CONS &1 *UNIVERSAL) -> &1
```

We enter the body of FEEDBACK with the type-alist

```
((X . (*REC INTEGER-LISTP
            (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
 (Y . (*REC INTEGER-LISTP
            (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
```

The segment produced for the occurrence of X is

```
((*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
 (*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
   ->
 (*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
```

I.e., when X and Y are both integer lists, X is an integer list.

Now we encounter (CAR X). Since X is the only argument, and its entry in PARTIAL-SOLUTION contains only one segment, we produce only one arg-solution,

```
(((*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
  (*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
 ((*REC INTEGER-LISTP
        (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
```

We grab the result from this arg-solution and use it to construct the form to match against the segments for CAR. We use the *SUBST form to annotate the type representing the occurrence of X. We use the *DLIST descriptor form to signify that a list of descriptors is being given to DUNIFY-DESCRIPTORS. The arguments to DUNIFY-DESCRIPTORS are:

```
(*DLIST (*SUBST &3 (*REC INTEGER-LISTP
                         (*OR $NIL (*CONS $INTEGER
                                          (*RECUR INTEGER-LISTP)))))
        &2)

(*DLIST $NIL $NIL)
```

where the latter is from the first segment in the signature of CAR. &2 in the first argument is the variable representing the result type. UNIFY-DESCRIPTORS returns:

```
((*DLIST (*SUBST &3 $NIL) $NIL) ((&2 . $NIL)) ((&3 . $NIL)))
```

Thus, our result variable &2 has been replaced with a result descriptor, $NIL, and the restriction on &3 tells us that we need to restrict X to $NIL when we construct our segment for this computation.

Unifying against the other segment for CAR goes much the same way, and the segments we finally produce for (CAR X) are:

```
((*CONS $INTEGER
        (*REC INTEGER-LISTP
              (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
 (*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> $INTEGER
($NIL
 (*REC INTEGER-LISTP (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> $NIL)
```

These also happen to be the segments for (CAR Y). So what are the potential argument patterns we present to BAR? We take the segments for the two arguments to BAR, (CAR X) and (CAR Y), and perform our cross-product computation to determine the set of arg-solutions:

```
((((*CONS $INTEGER
          (*REC INTEGER-LISTP
                (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   (*CONS $INTEGER
          (*REC INTEGER-LISTP
                (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
  ($INTEGER $INTEGER))

 (((*CONS $INTEGER
          (*REC INTEGER-LISTP
                (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   $NIL)
  ($INTEGER $NIL))

 (($NIL
    (*CONS $INTEGER
           (*REC INTEGER-LISTP
                 (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
  ($NIL $INTEGER))

 (($NIL $NIL) ($NIL $NIL)))
```

Thus, we will present four patterns to match against each of BAR's segments: 1) two integers, 2) an integer and NIL, 3) NIL and an integer, and 4) two NILs. In each case, both arguments will be annotated as being components of their respective variables, X and Y, and we will maintain a data structure which will enable us to reflect these patterns back into the type-alist entries for X and Y. Each argument pattern satisfies the guard for BAR, i.e., the guard, when unified with the argument pattern, does not restrict it.

The segments for BAR, as we said, are:

```
((*OR $INTEGER *NON-INTEGER-RATIONAL)
 (*OR $INTEGER *NON-INTEGER-RATIONAL))
  -> (*OR $INTEGER *NON-INTEGER-RATIONAL)

($NIL (*OR $INTEGER *NON-INTEGER-RATIONAL)) -> $NIL

((*OR $INTEGER *NON-INTEGER-RATIONAL) $NIL) -> $NIL

($NIL $NIL) -> $INTEGER
```

($INTEGER $INTEGER) matches only against the first segment, producing the result type (*OR $INTEGER *NON-INTEGER-RATIONAL). We map the context from this arg-solution to this result type, producing the segment

```
(((*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
  (*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
   ->  (*OR $INTEGER $NON-INTEGER-RATIONAL))
```

Moving along to our next arg-solution, ($INTEGER $NIL) matches only the third segment, producing the result type $NIL, etc. Our final collection of segments for (BAR (CAR X) (CAR Y)) is:

```
(((*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
  (*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
   ->  (*OR $INTEGER $NON-INTEGER-RATIONAL))

(((*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
  $NIL)
   -> $NIL)

(($NIL
  (*CONS $INTEGER
         (*REC INTEGER-LISTP
               (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
   -> $NIL)

(($NIL $NIL) -> $INTEGER))
```

Since this is the outermost form of the body, and since this is a non-recursive function, we are done. We are confident that FEEDBACK has no guard violations, since its guard is complete and the guards for INTEGER-LISTP, CAR, CDR, and BAR are complete, and we detected no guard violations in either the guard[21] or the body. Its signature is as above.

So how do we deal with recursive functions? Our computation of result types of subexpressions depends on the existence of a signature for the called function, and for a recursive function call we are only in the process of computing that signature. Indeed, we have no hint of the signature initially. To deal with this lack of information, we employ an algorithm which computes an iterative approximation of the signature, where one iteration consists of making one pass through our table of equations.

To set the context for a more detailed discussion, recall the overview presented earlier:

> During the first pass, we have no segments to match against recursive function calls, so we know nothing about the results of those calls. However, this does not prevent us from producing a first approximation of our segments on this pass. On the second pass, we can use this approximation to produce a refined approximation. Eventually, we hope that the approximation we produce will be equivalent to the one we produced on the previous pass. When this happens, the algorithm has stabilized. We canonicalize our segments into a suitable form, and we are finished.
>
> Ensuring that we have reached stability is non-trivial, however. The function is recursively composing

[21]We did not illustrate how we established the absence of guard violations in the guard for FEEDBACK, but the process is identical to that for checking it in the function body. If the guard is a negatable type predicate, the checks in TYPE-PREDICATE-P are sufficient. Otherwise, the regimen for checking general function bodies is applied.

> structures which contribute to the result. If we proceed naively, these structures will just grow with each iteration, our segments will reflect that growth, and nothing will ever stabilize. Therefore, we employ a technique for determining when these structures are taking on a form which can be described with a closed recursive descriptor. With such a closed recursive descriptor, the expansion and folding which occurs during an iteration of the algorithm will produce the same result we had in the previous pass, thus leading to stabilization.

We use a new descriptor form, the *FIX descriptor, as the basis for this technique. The form of a *FIX descriptor is:

```
(*FIX (*DLIST <desc1> .. <descn>) <fix body form> )
```

where <fix body form> is an arbitrary descriptor, possibly containing another *FIX descriptor, or more importantly, yet another new form:

```
(*FIX-RECUR (*DLIST <desc1> .. <descn>))
```

embedded within it. The interpretation of a *FIX descriptor is the same as the interpretation of a *REC descriptor, with (*DLIST <desc1> .. <descn>) playing the role of the <recname> and *FIX-RECUR playing the role of *RECUR.

We construct a *FIX form as a result descriptor for any recursive function call. The descriptors used to fill in the (*DLIST <desc1> .. <descn>) are the descriptors for the argument types, one for each actual parameter. The body is the disjunction (see below) of the result types computed from unifying with all the segments from our current approximation of the signature. We call the segments of our most recent approximation the *working segments*. On our first pass, when we have no working segments, the <fix body form> will be the descriptor *EMPTY. On later passes, the disjunction may include references to other *FIX descriptors. This would indicate that the result from a previous recursive call appears in the result for the current one.

We employ the *DLIST in a *FIX form as we do the label in a *REC descriptor. It represents the point of recursion in the form, and if we are in a form with multiple *FIXes, we will know which *FIX to reflect back to. Having *FIX be distinct from *REC allows us to employ some special-purpose heuristics, and the information in the (*DLIST <desc1> .. <descn>) will be critical to figuring out when to close off the developing recursive form, as we shall see.

When we construct a new *FIX form, we attempt to make an important canonicalization with respect to embedded *FIX forms. (See Canonicalization Rule 34 in Section 4.4.2.) If we have an embedded *FIX form with the same descriptors in its *DLIST component as in the outer *FIX, we judge that we have discovered a recursion point in our descriptor. An example of such a form is:

```
(*FIX (*DLIST (*REC INTEGER-LISTP
                    (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
      (*OR $NIL
           (*CONS $NON-T-NIL-SYMBOL
                  (*FIX (*DLIST
                         (*REC INTEGER-LISTP
                               (*OR $NIL
                                    (*CONS $INTEGER
                                           (*RECUR INTEGER-LISTP)))))
                        *EMPTY))))
```

Notice that the <fix body form>s do not match in any way. But, the outer <fix body form> is always at least as good an approximation of the result of a recursive function call as the inner <fix body form>, when the arguments are typed according to the *DLIST. Thus, the inner <fix body form> is not particularly helpful. Moreover, this is an effective point at which to form a closure from our developing descriptor, so that it does not just grow indefinitely. We replace the embedded *FIX form with:

```
(*FIX-RECUR (*DLIST (*REC INTEGER-LISTP
                    (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))))
```

This gives us the new descriptor:

```
(*FIX (*DLIST (*REC INTEGER-LISTP
                    (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
      (*OR $NIL
           (*CONS $NON-T-NIL-SYMBOL
                  (*FIX-RECUR
                    (*DLIST
                      (*REC INTEGER-LISTP
                            (*OR $NIL
                                 (*CONS $INTEGER
                                        (*RECUR INTEGER-LISTP)))))))))
```

Merely finding an embedded *FIX form with a matching *DLIST does not mean we are done. The next approximation may be better than this one. On the other hand, the next approximation may be the same as this one, which would indicate we are stabilizing. We continue iterating until the entire collection of segments for our function body is identical to the previous iteration's working segments. When it is, we declare victory, transform our *FIX descriptors to *REC descriptors, perform some canonicalizations, and emerge with a signature for the recursive function.

Let us illustrate the technique with an example.

```
(DEFUN APPEND (X Y)
  (DECLARE (XARGS :GUARD (TRUE-LISTP X)))
  (IF (NULL X) Y (CONS (CAR X) (APPEND (CDR X) Y))))
```

A TRUE-LISTP is a CONS structure whose last CDR is NIL, and is characterized by the descriptor:

```
(*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
```

We proceed as with a non-recursive function until we reach the recursive call. At this point, our type alist is

```
((X . (*CONS *UNIVERSAL
             (*REC TRUE-LISTP
                   (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))))
 (Y . &8))
```

The segment previously computed for (CDR X) is:

```
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &8)
   -> (*REC TRUE-LISTP
            (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
```

and for Y:

```
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &8)
   -> &8)
```

Since the contexts in these two segments match, we have only one arg-solution, mapping the context to the argument list

```
((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
 &8)
```

There are no segments for APPEND with which to match, so our best-guess result type is *EMPTY. Our segment, then, mapping the types of X and Y to our *FIX form, is:

```
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &8) ->
 (*FIX (*DLIST (*REC TRUE-LISTP
                     (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
               *UNIVERSAL)
   *EMPTY))
```

Notice that we replaced the variable &8 with *UNIVERSAL in the *DLIST. This is because an artifact of the implementation can cause a variable in one iteration to appear with a different name in a succeeding iteration. Thus, leaving the variable in the *FIX *DLIST could cause the folding operation we perform at the end of each iteration through the equations to fail. Converting the variable to *UNIVERSAL circumvents this problem.

Having constructed this partial solution for our recursive call, we complete the first iteration through the equations and produce the first pass working segments:

```
((($NIL &8) -> &8)
 (((*CONS &10 (*REC TRUE-LISTP
                    (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   *UNIVERSAL) ->
  (*CONS &10
         (*FIX
           (*DLIST (*REC TRUE-LISTP
                         (*OR $NIL
                              (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
                   *UNIVERSAL)
           *EMPTY))))
```

The first segment came from the (NULL X) branch of the function, where we return Y. The second segment is from the other branch, where we CONS the CAR of X, represented by the type variable &10, onto the result of the recursive call, our *FIX descriptor.

On our next iteration, when we hit the recursive call, and find that our argument pattern matches both these segments. The body of our *FIX descriptor is therefore an *OR whose first disjunct is the variable representing Y and whose second is the *CONS with the nested *FIX from the second segment. Thus, we construct the following *FIX descriptor for the value returned on the recursive call.

```
(*FIX (*DLIST (*REC TRUE-LISTP
                    (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
              *UNIVERSAL)
      (*OR &8
           (*CONS &10
                  (*FIX
                    (*DLIST (*REC TRUE-LISTP
                                  (*OR $NIL (*CONS *UNIVERSAL
                                                   (*RECUR TRUE-LISTP))))
                            *UNIVERSAL)
                    *EMPTY))))
```

Now we notice that within this *FIX we have another *FIX with the same parameter *DLIST (but a less precise body). This is our signal to replace the nested *FIX by a *FIX-RECUR, giving us:

```
(*FIX (*DLIST (*REC TRUE-LISTP
                    (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
              *UNIVERSAL)
      (*OR &8
           (*CONS &10
                  (*FIX-RECUR
                    (*DLIST (*REC TRUE-LISTP
                                  (*OR $NIL (*CONS *UNIVERSAL
                                                   (*RECUR TRUE-LISTP))))
                            *UNIVERSAL)))))
```

Finishing this iteration, we have the segments:

```
(($NIL &8) -> &8)
(((*CONS &10 (*REC TRUE-LISTP
                   (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &8)
 ->
 (*CONS &10
        (*FIX
         (*DLIST (*REC TRUE-LISTP
                       (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
                 *UNIVERSAL)
         (*OR &8
              (*CONS &10
                     (*FIX-RECUR
                       (*DLIST (*REC TRUE-LISTP
                                     (*OR $NIL
                                          (*CONS *UNIVERSAL
                                                 (*RECUR TRUE-LISTP))))
                               *UNIVERSAL)))))))
```

On our next pass, our recursive call initially yields the segment:

```
(*FIX
  (*DLIST (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
          *UNIVERSAL)
  (*OR &8
       (*CONS
        &10
        (*FIX
          (*DLIST (*REC TRUE-LISTP
                        (*OR $NIL (*CONS *UNIVERSAL
                                         (*RECUR TRUE-LISTP))))
                  *UNIVERSAL)
          (*OR &8
               (*CONS &10
                      (*FIX-RECUR
                       (*DLIST (*REC TRUE-LISTP
                                     (*OR $NIL
                                          (*CONS *UNIVERSAL
                                                 (*RECUR TRUE-LISTP))))
                              *UNIVERSAL)))))))
```

Again, we find a nested *FIX with the same *DLIST as the outer one, so we replace it with the *FIX-RECUR form to get:

```
(*FIX (*DLIST (*REC TRUE-LISTP
                    (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
              *UNIVERSAL)
      (*OR &8
```

```
           (*CONS &10
                  (*FIX-RECUR
                    (*DLIST (*REC TRUE-LISTP
                                  (*OR $NIL (*CONS *UNIVERSAL
                                                   (*RECUR TRUE-LISTP))))
                            *UNIVERSAL)))))
```

This is the same segment we got on the previous iteration. It should come as no surprise, then, that the new working segments we get when we complete this iteration are the same as the ones for the previous iteration. This signals that we have achieved stability.

Now all that remains is to convert our *FIX descriptor into a *REC descriptor. This is a straightforward process of generating a new rec-name, replacing our *FIX-RECUR with a *RECUR with that rec-name label, replacing the *FIX label with *REC, and replacing the *DLIST with the rec-name, and (sadly) replacing any variables which appear in replicating components of the new *REC descriptor with *UNIVERSAL. This replacement is necessary because a variable replicated in the repeated opening of a *REC descriptor, if interpreted in the formal semantics, INTERP-SIMPLE, would signify a replicated value, and this is clearly not what is signified here. But we say the replacement with *UNIVERSAL is sad because we could preserve some additional information if we replaced the variable instead with a different kind of variable, one which could be instantiated with another type descriptor. For APPEND (as we shall see), such a variable would allow the signature for APPEND to be such that if we called APPEND with objects which were both integer lists, we could deduce that the result was an integer list. As it is, we will know only that the result terminates in an integer list, but the specific knowledge about the types of the elements of the first argument will be lost in the result. An excellent enhancement to the tool would be addition of this new kind of variable to the formal semantic model.

But, regrets aside, SOLVE-EQUATIONS yields the segments of our final signature:

```
(($NIL &8) -> &8)

(((*CONS &10 (*REC TRUE-LISTP
                   (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &8)
   -> (*CONS &10 (*REC !REC4 (*OR &8
                                  (*CONS *UNIVERSAL (*RECUR !REC4))))))
```

Just as with a *REC descriptor, a *FIX descriptor with no *FIX-RECUR is equivalent to its body. This allows *FIX forms with no recursion to be reduced to simple forms. Thus, with a recursive function like:

```
(DEFUN LEN (X)
  (DECLARE (XARGS :GUARD (TRUE-LISTP X)))
  (IF (CONSP X) (BINARY-+ 1. (LEN (CDR X))) 0.))
```

the final segment can fold all the way to:

```
(((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   ->  $INTEGER)
```

This same property also lets us contain the treatment of recursive calls where the arguments are grounded, non-recursive forms. For example,

```
(DEFUN FOO (X)
  (IF (EQUAL X 3)
      3
      (IF (NULL X)
          NIL
```

```
          (IF (ATOM X)
              (FOO NIL)
              (IF (EQUAL (CDR X) #\a)
                  (CONS (FOO (CAR X)) (FOO (CDR X)))
                  (CONS (FOO (CAR X)) (FOO 3)))))))
```

produces the signature segments:

```
(($INTEGER) -> $INTEGER)

(((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $STRING $T))
   -> $NIL)

(((*CONS *UNIVERSAL $CHARACTER))
   -> (*CONS (*REC !REC9
                   (*OR $INTEGER
                        $NIL
                        (*CONS (*RECUR !REC9)
                               (*OR $INTEGER $NIL))))
             $NIL))

(((*CONS *UNIVERSAL *UNIVERSAL))
   -> (*CONS (*REC !REC10
                   (*OR $INTEGER
                        $NIL
                        (*CONS (*RECUR !REC10)
                               (*OR $INTEGER $NIL))))
             (*OR $INTEGER $NIL))))
```

In this function, the only recursive calls on recursively-typed arguments are the instances of (FOO (CAR X)), and these are the only ones which engender *REC forms in the result.

One intuitive question which might arise in evaluating an algorithm like this is consideration of where it loses information. One notable place is the point when, after matching our argument pattern against the working segments on a recursive function call, we generate a single segment whose result type is the disjunction of the result types from the matching segments. In the non-recursive case, were we to match more than one segment, we would produce more than one segment, possibly splitting on the types of the context variables. For recursive calls, we prefer to leave the context intact and produce a single, merged result. Although we are losing some specificity, this merging has a very sensible connection to recursive types. Functions composing a recursive result typically split their actions into some base case, in which they return a ground term, and a recursive case, in which they compose upon the result of a recursive call. This structure is typically mirrored in recursive type descriptors, which contain a disjunction where some disjuncts are non-recursive and others contain embedded *RECUR points. By forming our disjunctive descriptors when we match multiple segments on a recursive call, we are letting the function structure guide our construction of recursive descriptors.

An unfortunate aspect of the choice to take the disjunction in this manner is that, for some functions, it is a very poor strategy. It is most effective for functions which are building recursive structures as they traverse their input, like APPEND, and for functions which return a result whose type is not highly dependent on the input type structure, like LEN. For functions, like recognizer functions, which return simple objects whose type is highly dependent on the input type structure, this method results in analysis which probes only one level into input parameter types. For instance, consider the TRUE-LISTP function, modified to return 'OK and 0 instead of T and NIL.

```
(DEFUN SAD-BUT-TRUE-LISTP (X)
  (IF (NULL X) 'OK (IF (CONSP X) (SAD-BUT-TRUE-LISTP (CDR X)) 0)))
```

We get the segments:

```
(($NIL) -> $NON-T-NIL-SYMBOL)
(((*CONS *UNIVERSAL *UNIVERSAL)) -> (*OR $INTEGER $NON-T-NIL-SYMBOL))
(((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
       $STRING $T))
   -> $INTEGER)
```

Clearly, this is less satisfactory than the perfect segments we get for TRUE-LISTP itself, the result of its treatment as a recognizer:

```
(((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   -> $T)
(((*REC !REC1
        (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
             $STRING $T (*CONS *UNIVERSAL (*RECUR !REC1)))))
   -> $NIL)
```

This is simply another case where it would be possible to refine the algorithm to get more accurate results. A nice enhancement would be to make some heuristic choice about whether we would try to focus on building *FIX forms in the context portion of the segments rather than in the result type. If this were possible, then we could perhaps produce as strong a signature in the general case as we do for recognizers. As it stands, given the limits on time spent developing and implementing the algorithm, it seemed a practical choice to let the general algorithm focus on building *FIX forms in the result type, since a large portion of the functions for which we might wish to focus on the arguments are, in fact, recognizers, and we have very effective methods for dealing with them.

4.4.8 INFER-SIGNATURE

INFER-SIGNATURE is the top-level function in the type inference algorithm. Given a function and the database of signatures, it returns a new signature, consisting of the guard descriptors, a flag saying whether the guard is complete, a flag saying whether all functions in the call tree of the function have complete guards, the list of segments for the function, and an item which, if the function is a recognizer, is the descriptor characterizing the type which the function recognizes, NIL otherwise.

INFER-SIGNATURE's job is relatively simple. It calls DESCRIPTOR-FROM-FNDEF, which determines if the function qualifies as a recognizer. If so, DESCRIPTOR-FROM-FNDEF returns the descriptor corresponding to the type it recognizes, and INFER-SIGNATURE constructs the segments for the function by mapping this descriptor to $T and by mapping its (canonicalized) negation to $NIL. The guard is *UNIVERSAL, the guard-complete flag is T, and the all-functions-called-complete flag is T.

If the function is not a recognizer, INFER-SIGNATURE makes sure the guard does not call the function recursively, since this would be ill-formed, and then invokes PREPASS on both the guard and the body. Next it invokes DERIVE-EQUATIONS and SOLVE-EQUATIONS on the guard form to ensure there are no guard violations in the guard. Either a guard violation or a recursive call in the guard results in failure. If neither is detected, INFER-SIGNATURE invokes TYPE-PREDICATE-P on the prepassed guard form to construct the vector of descriptors characterizing the guard. The interpretation of this vector is that if the guard evaluates to a non-NIL value, the descriptors will characterize the parameters, and if the guard descriptors are complete, satisfaction of the guard descriptors by the actual parameter descriptors guarantees satisfaction of the actual guard. Again, there is no formal argument behind this claim, though the checker will attempt to establish its own version when its time comes.

These descriptors are then used as the initial type-alist for invoking DERIVE-EQUATIONS and SOLVE-EQUATIONS on the prepassed function body. Unless that algorithm detects a guard violation, it will produce the segments which will then be stored in the signature, along with the guard descriptors, the flag (computed by TYPE-PREDICATE-P) saying whether the guard was complete, the flag easily computed by checking the guard-complete flags for all functions called in the body, and the NIL value for the recognizerp flag.

Chapter 5
THE FORMAL SEMANTIC MODEL

This chapter gives a bottom-up construction of the formal basis for the specification and proof. We start by defining the type descriptor language which will be formalized, and by giving well-formedness predicates for type descriptors. Then we present the well-formedness predicates for signatures. Next, we provide a simple Lisp evaluator, E, which is used in the specification and proof to provide the value of a Lisp form in a given binding environment. Then, we give a formal semantics for type descriptors and finally a full statement of the formal semantics of function signatures. The presentation is designed to reveal concepts distinctly and to motivate an understanding of the choices contributing to the chosen model. At the end of the chapter, there are pointers to discussions of several of the very different semantic models which were considered before settling on this one.

5.1 A Discussion of the Style of Formalism

Before embarking on the formal treatment, we will attempt to characterize the style of what is to follow and to provide some motivation for the choices which led to this style.

An applicative subset of Common Lisp is the language in which the formal semantics is cast. It is also the language used in the implementation of the algorithm, and it is therefore a comfortable language in which to cast functions of the algorithm for formal analysis.

A carefully chosen applicative subset of Common Lisp could be used as a completely formal logic [Boyer 90]. The evaluator E, which we will introduce later in this chapter and which provides a semantics for the Lisp subset supported by the type system, is one possible semantic basis for using Common Lisp in this manner.

But the proof we will present is not a formal proof. Our use of Common Lisp as the vernacular for the formal analysis is not completely formal. For example, non-total functions are used in the discussion. Most of the functions introduced in this chapter which are a part of the formal semantics are stated with guards, in the same style used in the supported Lisp subset, so that the statement of their proper domain is formal. But later on when we introduce functions representing components of the checker algorithm, we do not provide guards. Rather, we accompany the definitions with textual accounts of what is assumed about the function arguments. Moreover, two functions introduced in the formal semantics as SUBRs are defined in English rather than formally. These are WELL-FORMED-WORLD, which is a guard on one of the arguments to E, and E-SUBRP, which is the subsidiary of E which evaluates functions which are in the initial library of primitive functions. Another informal aspect of the analysis is that some of the functions representing components of the algorithm do not terminate on all arguments. This is pointed out where appropriate, but the ramifications of having non-terminating functions in a formal setting are not elaborated, beyond claiming that we have only partial correctness proofs. Finally, we have not been explicit about the machinations of Lisp as a logic supporting the type system, as we have with the machinations of E as an interpreter for the language supported by the type system. In all these senses, the reader is expected to fill in the gaps, which is certainly not characteristic of formal proof.

Yet we go on to conduct the proofs with great rigor, exploring cases exhaustively and writing our accounts in extensive detail, in a manner which is more consistent with styles of formal proof. Though we have tried to emphasize important points textually, the reader still risks having his view of the proof obscured by detail. So by not treating Common Lisp as a formal logic, but by presenting our proof in rigorous detail, we seem to be sacrificing both the expressive ease of informal proof and the absolute certainty of formal proof. What motivated this choice of style?

Common Lisp linguistically unifies the discourse of the entire problem. As a powerfully expressive programming language, it was well-suited to developing the prototype implementation, which was an invaluable exploratory tool in developing the algorithms. And as previously stated, Common Lisp can be used as a formal logic, and it is uniquely suited to seamless discourse about Common Lisp functions. The researcher's personal comfort with Lisp played no small factor; the type inference problem itself was quite enough with which to grapple.

When we embarked on the proof, we were genuinely uncertain of the correctness of the algorithm. Since the checker is a computational algorithm with a great deal of case analysis and sometimes tricky detail, there was reason to suspect errors, though we had no particular areas of suspicion. Only by extensive, if not exhaustive, exploration of the details could we raise our level of assurance of the soundness of the algorithm. This level of concern was borne out in several instances. In one case, a significant error in the checker's unification algorithm was overlooked in an apparently plausible proof sketch laid out at the level of a typical informal proof. Only by later fleshing out the proof with rigorous detail did we uncover the error. In another case involving the containment algorithm, an early draft of the proof addressed a particular detail insufficiently to satisfy one reader. Had the proof not already been expressed in rigorous detail, arguably the point in question might not have become apparent to the reader. Though in the end no error was found in the algorithm, further exploration of the problem revealed questions which merited careful consideration. In short, regardless of the level of formality, exhaustive rigor was necessary to give us the level of confidence in the algorithm which we sought. And since this thesis was to represent a complete account of the work rather than a summary, we felt an obligation to present the work performed in essentially the style with which it was conducted.

While adopting this level of rigor, why did we not take the next step to formality? The humble graduate student feels himself to be extremely fallible. Given the nature of the system being proved, we arguably believe that the only sure path to a formal proof would be with mechanized assistance. Without it, the proof would suffer from the tedium of formality compounding the tedium of rigor. With the human tendency in the face of mountains of detail to overlook cases, and more importantly, to lose track of the central ideas being questioned, we feared that attempting what would amount to a human-generated mechanical proof would border on folly. Yet to have embarked on a mechanically assisted proof, perhaps with Nqthm [Boyer & Moore 88], seemed no less a folly. One of the precepts of mechanical proof as it is practiced with current technology is that the human needs to have the conceptual proof firmly in mind before seeking mechanized assurance. As previously stated, this was clearly not the case. And we could see that the level of effort necessary to reach this point and then execute a mechanical proof would well exceed the time available.

In essence, the level of assurance we received by using this style of proof was greater than we could have achieved with a high level, informal proof, and the commitment of effort was significantly more modest than would have been required for a formal proof.

Moreover, it is far from clear that this was the appropriate point in this research effort to conduct a mechanical proof. The system as it stands occupies one stage in its evolutionary development. We knew at the time we embarked on the proof that there were many ways we might wish to later enhance the system. Our goal was to establish a solid proof of concept which both established the validity of the type system as currently developed and which laid the groundwork for later formal proof. From this basis we could decide upon the next commitment, choosing from options like making further enhancements to the system in support of its current scope, enlarging its scope to include other language constructs, and re-implementing the system in the context of a more general proof environment. In any case, we thought it prudent to hold off the major commitment of performing a mechanized formal proof.

Nevertheless, one of the ultimate goals of this effort, which has not yet been achieved, is a formal, mechanically checked proof of the soundness of this system, possibly with Common Lisp as the formal logic. As we sought, we have built a significant bridge to this end. Beyond a high-level, informal proof of the system, we have produced an extensive, very detailed, albeit informal sketch of a formal proof. With it, we can better evaluate the feasibility of performing a formal, mechanized proof, and the task of ultimately constructing such a proof on this basis will be much less daunting than upon the basis of a high-level, informal proof.

5.2 Type Descriptors and Their Well-Formedness

The following is the grammar for type descriptors with which we will be concerned in our semantic model and in the checker.[22]

```
<descriptor> ::= <simple descriptor>  |  <variable>  |  *EMPTY  |
                 *UNIVERSAL | (*CONS <descriptor> <descriptor> )  |
                 (*OR <descriptor>* )  | <rec descriptor>

<simple descriptor> ::=
    $CHARACTER | $INTEGER | $NIL | $NON-INTEGER-RATIONAL |
    $NON-T-NIL-SYMBOL | $STRING | $T

<rec descriptor> ::= (*REC <rec name> <recur descriptor> )

<rec name> ::= a symbol whose first character is not "&"

<variable> ::= a symbol whose first character is "&"

<recur descriptor> ::=
   <simple descriptor>  |  <variable>  |  *EMPTY  |
   *UNIVERSAL | (*CONS <recur descriptor> <recur descriptor> )  |
   (*OR <recur descriptor>* )  | <rec descriptor> |
   (*RECUR <rec name>)
```

Furthermore, <recur descriptor> is constrained so that when it takes the form (*RECUR <rec name>), the <rec name> must be identical to that associated with the immediately containing *REC form. The grammar for <recur descriptor> is just the grammar for <descriptor> extended with the *RECUR form. We sometimes call the <recur descriptor> inside a <rec descriptor> the *body* of the <rec descriptor>.

Type variables may only appear within those portions of a *REC descriptor which are not replicated

[22]Here we do not need to deal with the transient forms generated by the inference tool and relieved internally, i.e., <and descriptor>, <not descriptor>, <subst descriptor>, *MERGE-FIX-POINT, <fix descriptor>, <fix-recur descriptor>, *ISO-RECUR, *FREE-TYPE-VAR, and **RECUR-MARKER**. The inference tool descriptor grammar is in Section 4.3.

within any disjunct when the descriptor is opened. (See the definition of OPEN-REC-DESCRIPTOR-ABSOLUTE in Section 5.5.) The following definition will help explain the well-formedness requirement below.

Definition: Consider a normal form for a *REC descriptor in which the body is an *OR descriptor, and no *OR occurs within any of its disjuncts unless it is part of a nested *REC form. I.e., all disjunction in the body of the *REC is raised to the top.[23] **A** *replicating component*, **or** *replicating disjunct of a *REC descriptor* **is any disjunct of this top-level *OR which contains a *RECUR for the *REC. A** *terminating component* **or** *terminating disjunct* **is any disjunct which does not contain a *RECUR for the *REC.**

To illustrate this definition, consider the descriptor

```
(*REC FOO (*OR $NIL (*CONS &2 (*RECUR FOO))))
```

This descriptor is already in *OR-lifted form. $NIL is a terminating disjunct, since it contains no (*RECUR FOO) form. (*CONS &2 (*RECUR FOO)) is a replicating disjunct. Now consider

```
(*REC BAR (*OR &1 (*CONS (*OR $INTEGER $T) (*RECUR BAR))))
```

Raising the disjunction in this form produces

```
(*REC BAR (*OR &1
               (*CONS $INTEGER (*RECUR BAR))
               (*CONS $T (*RECUR BAR))))
```

&1 is a terminating disjunct, and the two *CONS forms are replicating disjuncts.

A well-formedness requirement for descriptors must hold for any descriptors considered formally or manipulated by the checker. The top-level function defining the well-formedness predicate is WELL-FORMED-DESCRIPTOR.

Aside from conformance with the grammar above, the following properties must hold:

1. All *REC descriptors with the same name must be identical.
2. Any *RECUR form must have the same label as the nearest enclosing *REC.
3. Any *RECUR form must be nested within both an *OR and a *CONS within the *REC body. (A *RECUR within an *OR but not within a *CONS is redundant. A *RECUR within a *CONS but not within an *OR would represent an infinite structure. A *RECUR within neither an *OR or a *CONS would be vacuous.)
4. A type variable may not appear within a replicating component of a *REC.

As we shall see when we define the semantics for descriptors, little of practical value is lost by imposing this final requirement.

---

[23] The lifting of *OR in the body is accomplished by repeated application of the reverse of two canonicalization rules:

```
Rule 8: (*OR (*CONS d1 d2) (*CONS d3 d2)) => (*CONS (*OR d1 d3) d2)
Rule 9: (*OR (*CONS d1 d2) (*CONS d1 d3)) => (*CONS d1 (*OR d2 d3))
```

The flattening of the *OR structure is by

```
Rule 4: (*OR .. (*OR d1 .. d2) ..) => (*OR .. d1 .. d2 ..)
```

Rules 8 and 9 can be applied in reverse because their proofs show their two sides to be equivalent. These rules are given in Section 6.7, and their proofs are presented in Appendix B.6.

The function WELL-FORMED-DESCRIPTOR and its subsidiaries capture these notions of well-definedness. They are given in Appendix G.1.

A naive notion of the semantics of descriptors is that a descriptor defines a set of Lisp values, and a particular value satisfies the descriptor if it is a member of the set. But a descriptor containing type variables cannot define a set in the absence of some interpretation for variables. The notion of variables embedded in our formal semantics is that a type variable represents an arbitrary type containing a single value. Then the set of values we seek is defined by both the descriptor in question plus a binding list which maps each variable in the descriptor to some Lisp value. We will explore this more fully as we develop the semantic model.

5.3 Well Formedness of Function Signatures

A signature is well-formed iff:

1. The length of the guard descriptor is the same as the arity of the function.
2. The guard descriptor is a list of well-formed descriptors.
3. The left hand side of each segment is a list of well-formed descriptors whose length is the same as the arity of the function.
4. The right hand side of each segment is a well-formed descriptor.
5. All *REC descriptors throughout the signature with the same name must be identical.

Functions which check the well-formedness of a type signature returned by the inference tool are given in Appendix G.2. The top level function is WELL-FORMED-SIGNATURE.

The state consists of the signatures of a set of function symbols, each associated with a definition. We will frequently refer to the state by the name FS in the exposition and proof of soundness. In the implementation code it goes by the name FUNCTION-SIGNATURES. The state is well formed if each of its signatures is well formed. The function WELL-FORMED-STATE-ASSUMING-ARITY, given in Appendix G.3, checks the state, mapping over it with WELL-FORMED-SIGNATURE.

5.4 A Simple Lisp Evaluator

Occasionally in the specification and proof, we will call upon an evaluator to provide the value of a Lisp form in some environment. Here we define that evaluator, which provides a simple semantics for the evaluation of Lisp expressions. Let the environment be:

```
ENV: A list of pairs, where the CAR of each pair is a variable
  name and the CDR is a value in our Lisp universe.  This list
  represents the bindings of variables to values in the environment.

WORLD: A list of tuples (<fnname> <args> <guard> <body>), where <fnname>
  is a function name (and not IF or QUOTE), <args> is its formal
  argument list, <guard> is a Lisp form under <args> and WORLD,
  and, when (subrp <fnname>) is nil, <body> is a Lisp form under
  <args> and WORLD, and when (subrp <fnname>), <body> is irrelevant,
  but may be thought of as a list of ordered pairs mapping each
  list of arguments satisfying the guard to some result value.
  X is a Lisp form under <args> and WORLD iff X is a variable in
  <args>, a literal in the data space of our Lisp subset, a list
  of length two whose CAR is the atom QUOTE and whose CADR is a
  value in the data space of our subset, a list of length four whose
  CAR is the atom IF and whose remaining elements are well-formed
```

```
  Lisp forms under <args> and WORLD, or a list, representing a
  function call, whose CAR is the some <fnname> in WORLD and whose CDR
  is a list of the same length as the <args> list for that function
  and whose elements are all well-formed Lisp forms under <args>
  and WORLD.
```

Our interpreter is a function E, defined below. E takes a FORM which is a well-formed Lisp form whose variables are bound in ENV, an environment ENV (as above), a WORLD (as above), and a non-negative integer value CLOCK. It can return either a value in the data space of our Lisp subset or one of two values *not* in that data space: one signifies a break due to the clock being exhausted, while the other signifies a break due to a guard not being satisfied. We present a statement of the proper domain of E with a guard form in the same style as that of the Lisp dialect defined by E. The functions referenced in the guard are presented in Appendix G.4. For comment on this style, see the discussion in Section 5.1.

```
(DEFUN E (FORM ENV WORLD CLOCK)
  (DECLARE (XARGS :GUARD (AND (WELL-FORMED-ENV ENV)
                              (WELL-FORMED-WORLD WORLD)
                              (WELL-FORMED-CLOCK CLOCK)
                              (WELL-FORMED-FORM FORM ENV WORLD))))
  (IF (< CLOCK 1)
      (BREAK-OUT-OF-TIME)
      (IF (AND (CONSP FORM) (ASSOC (CAR FORM) WORLD))
          ;; Function call
          (LET ((ACTUALS (MAP-E (CDR FORM) ENV WORLD CLOCK)))
            (IF (BREAKP ACTUALS)
                ACTUALS
                (LET ((FNCALL-ENV
                        (PAIRLIS (CADR (ASSOC (CAR FORM) WORLD))
                                 ACTUALS)))
                  ;; Evaluate the guard
                  (LET ((GUARD-VAL (E (CADDR (ASSOC (CAR FORM) WORLD))
                                      FNCALL-ENV
                                      WORLD
                                      CLOCK)))
                    (IF (BREAKP GUARD-VAL)
                        GUARD-VAL
                        (IF (EQUAL GUARD-VAL NIL)
                            (BREAK-GUARD-VIOLATION)
                            (IF (SUBRP (CAR FORM))
                                (E-SUBRP (CAR FORM) ACTUALS WORLD)
                                (E (CADDDR (ASSOC (CAR FORM) WORLD))
                                   FNCALL-ENV
                                   WORLD
                                   (- CLOCK 1)))))))))
          (IF (AND (CONSP FORM) (EQL (CAR FORM) 'IF))
              (LET ((TEST-VAL (E (CADR FORM) ENV WORLD CLOCK)))
                (IF (BREAKP TEST-VAL)
                    TEST-VAL
                    (IF (EQUAL TEST-VAL NIL)
                        (E (CADDDR FORM) ENV WORLD CLOCK)
                        (E (CADDR FORM) ENV WORLD CLOCK))))
              (IF (LITERALP FORM)
                  FORM
                  (IF (AND (CONSP FORM) (EQUAL (CAR FORM) 'QUOTE))
                      (CADR FORM)
                      ;; FORM is a variable
                      (CDR (ASSOC FORM ENV)))))))))

(DEFUN MAP-E (FORMS ENV WORLD CLOCK)
  (IF (NULL FORMS)
      NIL
      (LET ((NEXT-VAL (E (CAR FORMS) ENV WORLD CLOCK)))
        (IF (BREAKP NEXT-VAL)
            NEXT-VAL
            (LET ((REST-VALS (MAP-E (CDR FORMS) ENV WORLD CLOCK)))
              (IF (BREAKP REST-VALS)
```

```
                  REST-VALS
                  (CONS NEXT-VAL REST-VALS)))))))

(DEFUN SUBRP (FNNAME)
  (MEMBER FNNAME
          ;; The functions in our initial world
          '(CONS CAR CDR BINARY-+ UNARY-- BINARY-* UNARY-/ < EQUAL
                 CONSP INTEGERP RATIONALP STRINGP CHARACTERP SYMBOLP NULL
                 DENOMINATOR NUMERATOR SYMBOL-NAME SYMBOL-PACKAGE-NAME)))

(DEFUN E-SUBRP (FNNAME ACTUALS WORLD)
  ;; When the function is a subr, one may envision the <body> to be
  ;; an infinite list of pairs mapping values in its domain to the
  ;; result returned by the subr for those values.  The list is
  ;; of infinite length because the domain of every primitive function
  ;; happens to be infinite.  In this scheme of things, the body of
  ;; E-SUBRP would be
  ;;  (CDR (ASSOC ACTUALS (CADDDR (ASSOC FNNAME WORLD))))
  ;; To avoid having to consider such infinite lists in our semantic
  ;; model, however, we simply define E-SUBRP to be a "sub-primitive"
  ;; function which returns the value returned by the function named
  ;; by the FNNAME parameter on the values given by ACTUALS.
  <<subr value>> )

(DEFUN BREAK-OUT-OF-TIME ()
  ;; The floating point number .1 represents an out-of-time break.
  .1)

(DEFUN BREAK-OUT-OF-TIMEP (VAL)
  (EQUAL VAL (BREAK-OUT-OF-TIME)))

(DEFUN BREAK-GUARD-VIOLATION ()
  ;; The floating point number .2 represents a guard violation.
  .2)

(DEFUN BREAK-GUARD-VIOLATIONP (VAL)
  (EQUAL VAL (BREAK-GUARD-VIOLATION)))

(DEFUN BREAKP (VAL)
  (OR (BREAK-OUT-OF-TIMEP VAL) (BREAK-GUARD-VIOLATIONP VAL)))
```

E is a total function. This claim rests on the fact that a termination measure on E goes down on all recursive calls. This termination measure is defined by a three-tuple composed of CLOCK, the size of the world necessary to evaluate a term, and the size of the term (the length of its flattened structure). These are given in order of decreasing significance. The count goes down whenever the clock goes down (i.e., on entry to the body of a function). The second count element, the world size, is of use only when we evaluate the guard in conjunction with a function call. In this case, CLOCK does not go down, and the size of the guard term may be larger than the size of the function call term. But since the guard form never calls the function in which it appears, we may remove this function from the world for purposes of evaluating the guard. Hence this count goes down on the recursive call to evaluate a guard. For all other recursive calls, the size of the term goes down.

A trivial observation is that, given two calls to E which are identical except for the value of clock, if neither call returns (break-out-of-time), both calls will return the same value.

5.5 The Formal Semantics of Type Descriptors and Signatures

Here, we will expose the notions in the semantic model in two phases, first providing a model for variable-free descriptors, and then extending the model to encompass type variables.

As mentioned earlier, a naive notion of the semantics of descriptors is that a descriptor defines a set of Lisp values, and that a value satisfies a descriptor if it is a member of the associated set. This notion can be easily cast as a predicate SATISFIES-1, which determines whether a value is in the set defined by a variable-free descriptor. (SATISFIES-1 TD V) returns T if the value V is a member of the set defined by the descriptor TD, and NIL otherwise. We will later define a more general interpreter function which handles variables, but for now, we factor the problem to illustrate the significance of variables.

```
(DEFUN SATISFIES-1 (DESCRIPTOR VALUE)
  (CASE DESCRIPTOR
    (*EMPTY NIL)
    (*UNIVERSAL T)
    ($NIL (NULL VALUE))
    ($T (EQUAL VALUE T))
    ($NON-T-NIL-SYMBOL
      (AND (SYMBOLP VALUE) (NOT (NULL VALUE)) (NOT (EQUAL VALUE T))))
    ($INTEGER (INTEGERP VALUE))
    ($NON-INTEGER-RATIONAL
      (AND (RATIONALP VALUE) (NOT (INTEGERP VALUE))))
    ($STRING (STRINGP VALUE))
    ($CHARACTER (CHARACTERP VALUE))
    (OTHERWISE
      (CASE (CAR DESCRIPTOR)
        (*CONS (AND (CONSP VALUE)
                    (SATISFIES-1 (CADR DESCRIPTOR) (CAR VALUE))
                    (SATISFIES-1 (CADDR DESCRIPTOR) (CDR VALUE))))
        (*OR (SATISFIES-OR (CDR DESCRIPTOR) VALUE))
        ;; OPEN-REC-DESCRIPTOR-ABSOLUTE opens a *REC descriptor.
        ;; For example,
        ;; (OPEN-REC-DESCRIPTOR-ABSOLUTE
        ;;   '(*REC TL (*OR $NIL (*CONS *UNIVERSAL (*RECUR TL))))
        ;;  = (*OR $NIL (*CONS *UNIVERSAL
        ;;                     (*REC TL (*OR $NIL (*CONS *UNIVERSAL
        ;;                                          (*RECUR TL))))))
        (*REC
          (SATISFIES-1 (OPEN-REC-DESCRIPTOR-ABSOLUTE DESCRIPTOR) VALUE))
        (OTHERWISE NIL))))))

(DEFUN SATISFIES-OR (DESCRIPTORS VALUE)
  (IF (NULL DESCRIPTORS)
      NIL
      (OR (SATISFIES-1 (CAR DESCRIPTORS) VALUE)
          (SATISFIES-OR (CDR DESCRIPTORS) VALUE))))

(DEFUN SATISFIES (DESCRIPTOR-LIST VALUE-LIST)
  ;; We require DESCRIPTOR-LIST and VALUE-LIST to have the same length.
  (IF (NULL DESCRIPTOR-LIST)
      T
      (IF (SATISFIES-1 (CAR DESCRIPTOR-LIST) (CAR VALUE-LIST))
          (SATISFIES (CDR DESCRIPTOR-LIST) (CDR VALUE-LIST))
          NIL)))
```

SATISFIES handles lists of descriptors and values, which will come into play in our semantics for function segments. SATISFIES-1 handles a single descriptor and value. Thus,

```
 (SATISFIES-1
   (*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
   (2 4 "FOO"))
 = T
```

```
(SATISFIES-1
  (*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
  (A . B))
= NIL
```

But variables play an important role in the semantics of descriptors and function signatures. The notion of variables seems well-suited to describing, for instance, that the type of the result of the CAR function, applied to a CONS object, is the same as the type of the first value in the ordered pair defined by the CONS. The interpretation of multiple occurrences of a variable as type identity, rather than value identity, threatened to seriously complicate the formal semantics of signatures. (See Appendix E for the discussion of a semantic model considered but abandoned.) We were unable to provide a model of type variables which, in the presence of disjunction, would provide the kind of specificity we required for handling direct data transfer through a signature (as with CAR and CONS) and yet provide an instantiation capability which would support the appearance of a variable in a replicating component of a *REC. In the latter case, we certainly did not want to abide by the constraint that every value associated with the type variable had to be EQUAL. We chose for the time being to abandon the use of type variables in replicating components and to adopt a model that multiple occurrences of a type variable represent multiple occurrences of a single data value. A type variable, then, may have only a single-point instantiation.

At first glance, it would seem that a variable appearing in a *REC descriptor could be replicated on each recursion, thus requiring under these semantics that each occurrence represent the same value. This does not seem useful in most circumstances. We might like a variable in a *REC descriptor to represent multiple occurrences of some particular type, to be instantiated later. But in our implementation, we have restricted *REC descriptors so that variables may appear only in non-replicating disjuncts, so replication on unfolding does not occur. The notion of a variable which can be instantiated with a type is not present, though we discuss elsewhere (See Section 8.3) that one potentially valuable extension to the tool would be a new class of variables specifically for this purpose in *REC descriptors.

We can use our SATISFIES-1 function as part of a semantics for descriptors and function segments including variables. Doing so may further illuminate the intuition behind our type variables. Consider the type descriptor (*CONS &1 &1). We can factor the satisfaction problem for descriptors with variables into two parts. For a value X to satisfy (*CONS &1 &1), we would require

```
(SATISFIES-1 (*CONS *UNIVERSAL *UNIVERSAL) X)
```

But we would also require the CAR and CDR of X to be equal. So the expanded predicate for satisfying (*CONS &1 &1) is:

```
(AND (CONSP X)
     (SATISFIES-1 *UNIVERSAL (CAR X))
     (SATISFIES-1 *UNIVERSAL (CDR X))
     (EQUAL (CAR X) (CDR X))
```

Thus, to treat descriptors containing variables, we transform them all to *UNIVERSAL for submission to the SATISFIES-1 predicate, but then we add the requirement that all data objects whose types are represented by the same type variable within a descriptor or segment must be equal.

But proceeding in this way can become unwieldy. Consider the APPEND function.

```
(DEFUN APPEND (X Y)
  (DECLARE (XARGS :GUARD (TRUE-LISTP X)))
  (IF (NULL X)
      Y
```

```
      (CONS (CAR X) (APPEND (CDR X) Y))))
```

The signature for this function might be:

```
Guard:
  ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
   *UNIVERSAL)

Segments:
($NIL &2) -> &2

((*CONS &3 (*REC TRUE-LISTP
                 (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
 &2)
  -> (*CONS &3 (*REC !REC4 (*OR &2 (*CONS *UNIVERSAL (*RECUR !REC4)))))
```

Using SATISFIES, the interpretation of these segments is as follows. First, if the values x and y satisfy the real guard of the function, then they also satisfy the guard descriptors.

```
for all Lisp values X and Y,
 (TRUE-LISTP X)
 ->
 ( (SATISFIES (*REC TRUE-LISTP
                    (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
              X)
     and
   (SATISFIES *UNIVERSAL Y) )
```

Second, if the values x and y satisfy the real guard of the function, then one of the segments is appropriate. The (SATISFIES *UNIVERSAL ..) forms are unnecessary, but reflect the earlier discussion that a variable is transformed to *UNIVERSAL with the additional equality tests added. (Here we use "(APPEND X Y)" to denote the value returned by APPEND for the values X and Y.)

```
for all Lisp values X and Y,
 (TRUE-LISTP X)
 ->
 ( ((SATISFIES $NIL X) and (SATISFIES *UNIVERSAL Y) and
      (SATISFIES *UNIVERSAL (APPEND X Y)) and (EQUAL Y (APPEND X Y)))
  or
   ((SATISFIES
      (*CONS *UNIVERSAL
             (*REC TRUE-LISTP
                   (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
      X)
     and
    (SATISFIES *UNIVERSAL Y)
     and
    (SATISFIES
      (*CONS *UNIVERSAL
             (*REC !REC4
                   (*OR *UNIVERSAL (*CONS *UNIVERSAL (*RECUR !REC4)))))
      (APPEND X Y))
     and
    (EQUAL (CAR X) (CAR (APPEND X Y)))
     and
    (EQUAL Y <<some CDR of (APPEND X Y)>>)) )
```

The notion of <<some CDR of (APPEND X Y)>> obviously needs better treatment. To deal with it, we employ a recursively defined interpreter which can handle both the type predicate issues and the equality issues in tandem. This interpreter, defined by the functions INTERP-SIMPLE and INTERP-SIMPLE-1, is the one upon which we base the formal semantics for our system. Given a well-formed descriptor, a value, and a binding mapping type variables to Lisp values, INTERP-SIMPLE-1 returns T if the value satisfies the descriptor under the bindings, and NIL otherwise. INTERP-SIMPLE takes a list of descriptors, a list (of the same length) of values, and a binding. It applies INTERP-SIMPLE-1 to corresponding descriptor-value pairs, all under the same binding. All must return T in order for INTERP-SIMPLE to return T.

Essentially, the rules embodied in the interpreter are as follows. Any value satisfies *UNIVERSAL, and no value satisfies *EMPTY. Any value of the appropriate primitive type satisfies a primitive descriptor. A value satisfies a type variable if the variable is bound to the identical value in the bindings. A value satisfies a *CONS descriptor if it is a CONS whose CAR satisfies the first argument to the *CONS under the bindings and whose CDR satisfies the second argument. A value satisfies an *OR if it satisfies one of the disjuncts under the bindings. A value satisfies a *REC if it satisfies the body of the *REC with the *REC substituted for the *RECUR form. This allows the interpreter to descend as deeply into the value's structure as necessary, opening up the *REC descriptor as it goes. All the preceding requirements are checked by INTERP-SIMPLE-1. INTERP-SIMPLE is employed to check that lists of values satisfy lists of descriptors under some bindings, assuming the lists are of the same length. It could have been integrated with INTERP-SIMPLE-1, but since lists of descriptors never appear within a descriptor, removing it from INTERP-SIMPLE-1 simplifies case analysis in some proofs.

The functions mentioned in the guard are defined just below. As previously discussed in Section 5.1, we give the guards on our semantic interpreter functions in the same style we implement with our E function, so as to specify their proper domain. The definitions of the functions called in the guard appear in Appendix G.5.

```
(DEFUN INTERP-SIMPLE (DESCRIPTORS VALUES BINDINGS)
  ;; Where X, Y, and (APPEND X Y) denote values (We use (APPEND X Y)
  ;; to suggest a typical application.)
  ;; (INTERP-SIMPLE '($NIL &2 &2)
  ;;                '(X Y (APPEND X Y))
  ;;                '((&2 . Y)))
  ;; performs the following test:
  ;; (AND (NULL X)
  ;;      T
  ;;      (EQUAL Y Y)
  ;;      T
  ;;      (EQUAL (APPEND X Y) Y)
  (DECLARE
    (XARGS :GUARD (AND (ALL-WELL-FORMED-DESCRIPTORS DESCRIPTORS)
                       (ALL-WELL-FORMED-VALUES VALUES)
                       (WELL-FORMED-BINDINGS BINDINGS DESCRIPTORS))))
  (IF (NULL DESCRIPTORS)
      T
      (AND (INTERP-SIMPLE-1 (CAR DESCRIPTORS) (CAR VALUES) BINDINGS)
           (INTERP-SIMPLE (CDR DESCRIPTORS) (CDR VALUES) BINDINGS))))

(DEFUN INTERP-SIMPLE-1 (DESCRIPTOR VALUE BINDINGS)
  (DECLARE
    (XARGS :GUARD (AND (WELL-FORMED-DESCRIPTOR DESCRIPTOR)
                       (WELL-FORMED-VALUE VALUE)
                       (WELL-FORMED-BINDINGS BINDINGS DESCRIPTOR))))
  (CASE DESCRIPTOR
    (*EMPTY NIL)
    (*UNIVERSAL T)
    ($NIL (NULL VALUE))
    ($T (EQUAL VALUE T))
    ($NON-T-NIL-SYMBOL
      (AND (SYMBOLP VALUE) (NOT (NULL VALUE)) (NOT (EQUAL VALUE T))))
    ($INTEGER (INTEGERP VALUE))
    ($NON-INTEGER-RATIONAL
      (AND (RATIONALP VALUE) (NOT (INTEGERP VALUE))))
    ($STRING (STRINGP VALUE))
    ($CHARACTER (CHARACTERP VALUE))
```

```
    (OTHERWISE
      (IF (VARIABLE-NAMEP DESCRIPTOR)
          (AND (ASSOC DESCRIPTOR BINDINGS)
               (EQUAL VALUE (CDR (ASSOC DESCRIPTOR BINDINGS))))
          (CASE (CAR DESCRIPTOR)
            (*CONS
              (IF (CONSP VALUE)
                  (AND (INTERP-SIMPLE-1 (CADR DESCRIPTOR)
                                        (CAR VALUE)
                                        BINDINGS)
                       (INTERP-SIMPLE-1 (CADDR DESCRIPTOR)
                                        (CDR VALUE)
                                        BINDINGS))
                  NIL))
            (*OR (INTERP-SIMPLE-OR (CDR DESCRIPTOR) VALUE BINDINGS))
            ;; OPEN-REC-DESCRIPTOR-ABSOLUTE opens a *REC descriptor.
            ;; For example,
            ;; (OPEN-REC-DESCRIPTOR-ABSOLUTE
            ;;    '(*REC TL (*OR $NIL (*CONS *UNIVERSAL (*RECUR TL))))
            ;;  = (*OR $NIL
            ;;         (*CONS *UNIVERSAL
            ;;                (*REC TL (*OR $NIL (*CONS *UNIVERSAL
            ;;                                          (*RECUR TL))))))
            (*REC (INTERP-SIMPLE-1
                    (OPEN-REC-DESCRIPTOR-ABSOLUTE DESCRIPTOR)
                    VALUE
                    BINDINGS))
            (OTHERWISE NIL))))))

(DEFUN INTERP-SIMPLE-OR (DESCRIPTORS VALUE BINDINGS)
  (DECLARE
    (XARGS :GUARD (AND (ALL-WELL-FORMED-DESCRIPTORS DESCRIPTORS)
                       (WELL-FORMED-VALUE VALUE)
                       (WELL-FORMED-BINDINGS BINDINGS DESCRIPTORS))))
  (IF (NULL DESCRIPTORS)
      NIL
      (OR (INTERP-SIMPLE-1 (CAR DESCRIPTORS) VALUE BINDINGS)
          (INTERP-SIMPLE-OR (CDR DESCRIPTORS) VALUE BINDINGS))))
```

The (VARIABLE-NAMEP DESCRIPTOR) case, which was not present in our SATISFIES-1 function, establishes the relationship between the variable, the value, and the binding. Note that when no variables appear in (TD1i..TDni),

```
(SATISFIES (TD1i..TDni) (V1..Vn))
```

is equivalent to

```
(INTERP-SIMPLE (TD1i..TDni) (V1..Vn) NIL)
```

This is the case with guard descriptors and with the segments of recognizer functions, which are always variable-free.

Before giving the formal semantics for a signature, let us introduce a notational convention which will help keep the exposition clean.

Definition: foo^{world,clock}

This is a notational convention. Where arg$_1$ .. arg$_n$ are Lisp values and a$_1$ .. a$_n$ are the formal parameters of foo, by

foo^{world,clock}(arg$_1$ .. arg$_n$)

we denote

```
   (E (foo a_1 .. a_n)
      ((a_1 . arg_1) .. (a_n . arg_n))
      world
      clock)
```

The formal semantics for a signature is as follows. Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
Definition: GOOD-SIGNATUREP (FNDEF GUARD SEGMENTS WORLD CLOCK)

For any function foo of arity n in our Lisp subset, whose
definition is denoted

 (defun foo (a_1 .. a_n)
   (declare (xargs :guard guard-form))
   body)

and whose SEGMENTS are denoted
 ((td_{1,1} .. td_{1,n}) -> td_1) .. ((td_{m,1} .. td_{m,n}) -> td_m),
for any non-negative integer clock and world containing the
above definition of foo,

(good-signaturep foo guard segments world clock)
 =
(and
  (well-formed-signature-1 guard segments n)
  for any Lisp values arg_1 .. arg_n,
   (and
H1  (not (break-out-of-timep (foo^{world,clock}(arg_1 .. arg_n))))
H2  (not (null (E guard-form ((a_1 . arg_1) .. (a_n . arg_n)) world clock))))
   =>
   (and
C1   (not (break-guard-violationp (foo^{world,clock}(arg_1 .. arg_n))))
C2   for some k in [1..m]
       for some binding b of type variables to Lisp values
           covering td_{k,1} .. td_{k,n} and td_k
         (I (td_{k,1} .. td_{k,n} td_k)
            (arg_1 .. arg_n (foo^{world,clock}(arg_1 .. arg_n)))
            b) ) )
```

That is, if the guard of foo is satisfied by the values of the actual parameters arg_1 .. arg_n and there is adequate clock to completely evaluate the function call, then the result of the evaluation will not be a guard-violation break, and there is some segment for which, under some binding of all its type variables to Lisp values, the actual parameter values satisfy the formal parameter descriptors under the binding and the result of evaluating the function foo on those actual values satisfies the result descriptor under the same binding.

In the definition of GOOD-SIGNATUREP, WELL-FORMED-SIGNATURE-1 just checks that arities are consistent, checks the well-formedness of all the descriptors, and checks that the guard descriptors are variable-free.

```
Definition: VALID-FS (FS WORLD CLOCK)

(valid-fs fs world clock)
 =
For every signature (guard segments) in fs corresponding to a
function foo of arity n in world, whose definition is denoted
 (defun foo (a_1 .. a_n)
```

```
   (declare (xargs :guard guard-form))
   body),

 (and (tc-all-called-functions-complete guard-form fs)
      (tc-all-called-functions-complete body fs))
 =>
 (good-signaturep foo guard segments world clock)
```

Thus, the signature of a function FOO in FS can be soundly utilized only if the guard descriptors of all the functions called in FOO, including FOO itself, are complete. The guard descriptors will be complete if the guard is a conjunction of recognizer calls on disjoint formal parameters.

When a new function is presented to the type inference system, the system's essential task is to formulate a signature for which the VALID-FS predicate holds.

In Section 5.7.1, we discuss an alternate semantic model we considered which utilized signatures of the same form but provided a different interpretation, in which all, rather than some, segments whose left hand side was satisfied by the actual parameters yielded a correct type for the result.

For an example of our semantics, consider the CDDR function, defined as follows.

```
(DEFUN CDDR (X)
  (DECLARE (XARGS :GUARD (CDDR-ABLE X)))
  (CDR (CDR X)))
```

where

```
(DEFUN CDDR-ABLE (X)
  (IF (NULL X)
      (NULL X)
      (IF (CONSP X)
          (IF (NULL (CDR X)) (NULL (CDR X)) (CONSP (CDR X)))
          NIL)))[24]
```

Its signature is:

```
Guard: (*OR $NIL (*CONS *UNIVERSAL
                        (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL))))

Signature segments:
  ((*OR $NIL (*CONS *UNIVERSAL $NIL))) -> $NIL
  ((*CONS *UNIVERSAL (*CONS *UNIVERSAL &1))) -> &1
```

Given a value X satisfying the guard, and enough clock to complete the evaluation, either X satisfies

```
(*OR $NIL (*CONS *UNIVERSAL $NIL))
```

and (CDDR X) satisfies $NIL (i.e., (CDDR X) = NIL), or X satisfies

```
(*CONS *UNIVERSAL (*CONS *UNIVERSAL *UNIVERSAL))
```

and there is some value which is both the CDR of the CDR of X and the value of (CDDR X).

---

[24] The body of this function is the IF representation of

```
(OR (NULL X) (AND (CONSP X) (OR (NULL (CDR X)) (CONSP (CDR X)))))]
```

5.6 Other Components of the Signature

A simple example will help motivate a presentation of the formal significance of the other information computed and stored by the inference tool. Consider a function:

```
(DEFUN FOO (X)
  (DECLARE (XARGS :GUARD (CONSP X)))
  (IF (CONSP (CAR X))
      (CONS "CAR is a CONS" NIL)
      (CONS 0 (CDR X))))
```

Given this function, the inference tool stores the following information.

```
Function: FOO
Guard computed by the tool:
 ((*CONS *UNIVERSAL *UNIVERSAL))
Guard complete: T
All called functions complete: T
TC Guard:
 ((*CONS *UNIVERSAL *UNIVERSAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*CONS (*CONS *UNIVERSAL *UNIVERSAL) *UNIVERSAL))
   -> (*CONS $STRING $NIL))
 (((*CONS (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
               $NON-T-NIL-SYMBOL $STRING $T)
    (*FREE-TYPE-VAR 1.)))
   -> (*CONS $INTEGER (*FREE-TYPE-VAR 1.)))
TC segments contained in Segments: T
Recognizer descriptor:
  NIL
TC validates recognizer: NIL
Signature is certified sound: T
```

We have already discussed the guard descriptors and segments. Let us discuss the formal aspects of the others in sequence.

We have said informally that a guard descriptor is complete if any value satisfying the guard descriptor will satisfy the real guard. This is the sense of the "Guard complete" flag, but this particular flag represents a statement made by the untrusted inference tool, so it carries no formal weight and must be viewed as no more than an interesting conjecture. Similarly, the "All called functions complete" flag represents the inference algorithm’s statement that all the functions called in a function body have complete guard descriptors, and this carries no formal value.

By contrast, the "TC Guard complete" flag means that the guard complies with the following definition:

Definition: The guard descriptors for a function are *complete*
if the guard in the function definition is a conjunction of calls to recognizer functions (see below) on distinct formal parameters.
Note: When a guard is of this form, then its descriptor will be such that for any actual parameter values, the guard will evaluate to a non-NIL Lisp value (given sufficient clock) if and only if the values satisfy the guard descriptor under interpretation by INTERP-SIMPLE. (Any binding will do for this interpretation, since the guard descriptor is variable-free.) This property will be established by the proofs of lemmas GUARD-COMPLETE and TC-INFER-SIGNATURE-GUARD-OK in Chapter 7.

"TC All called functions complete" means that all functions called in either the guard or function body, and all the functions called in those functions, and so forth throughout their hereditary call trees have this same property. These simple claims have been validated by the trusted algorithm, and by the reasoning embodied in Lemma GUARD-COMPLETE, presented originally in Section 3.6.1, define the conditions under which we know that the checker's failure to detect a guard violation guarantees that the actual guard will evaluate to a non-NIL value, specifically T.

The "TC Guard Replaced by Tool Guard" flag indicates that the checker opted to replace the guard it computed itself with the guard originally computed by the inference tool. This is only done when three tests are met. First, the guards must be different. (Else, why bother?) Second, the containment algorithm must determine that the checker guard is contained in the inference tool guard. Third, the inference tool guard must be contained in the checker guard. This mutual containment guarantees that the two guard forms are equivalent. The reason for preferring the inference tool guard is a presumption that it is in a nicer canonical form. This replacement rarely occurs in practice, but on rare occasions it allows for a more compact representation of the guard.

The "TC segments contained in Segments" flag indicates the finding by TC-SIGNATURE that under some binding, each of the checker segments is contained in some inference tool segment. This is a critical hypothesis in Lemma TC-SIGNATURE-OK, proved in Section 7.3.3.

Definition: We say a function is a *recognizer* if its definition possesses all the following properties:

1. **It has exactly one formal parameter.**
2. **Its guard is T, either explicitly or by omission, and therefore its guard descriptor is (*UNIVERSAL).**
3. **It returns only T or NIL, and there exists for it a correct signature containing exactly two variable-free segments, one with a result type of $T and the other with a result type of $NIL, such that the argument descriptors of the segments characterize disjoint sets.[25]**
4. **The guards of all functions hereditarily in its call tree are complete.**

The "Recognizer descriptor" item, when non-NIL, indicates that the inference algorithm considers this function to be a recognizer, and the value stored here is a descriptor characterizing the values for which the function returns T. As it is another judgement by the inference algorithm, it has no formal significance. However, the "TC validates recognizer" flag, stored by the checker, indicates that the checker agrees the function is a recognizer.

Finally, the "Signature is certified sound" flag indicates that the signature can be trusted, i.e., the guard is complete, the segments have passed the containment test, if the function has a recognizer descriptor, the checker validated its status as a recognizer, and every function in the hereditary call tree of this one also meets these same tests. This flag is never used except in computing the value of the flag for subsequently submitted functions, and it plays no role in the formal analysis. It is simply provided as a convenience for reading output, as its value summarizes all checks a reader would need to make in order to see that a signature has been certified by the tool.

---

[25] When the system admits a function as a recognizer, it establishes this disjointness by computation, unifying the argument descriptors and checking that the result is *EMPTY. The formal justification that this test is adequate is simple and is illustrated in the proof of Lemma RECOGNIZER-SEGMENTS-COMPLETE in Appendix B.2.

5.7 Alternate Semantic Models

A number of semantic models were explored before arriving at the one finally adopted. Perhaps more than any other single phenomenon, the difficulty of finding a model which suited the nature of the problem gives evidence to the general difficulty of the problem itself, its statement, its solution, and its proof of soundness. In this section we present several of the alternatives which received significant thought. Perhaps by doing so we can shed further light on the problem itself. Comparison and contrast may lead to a better understanding of the solution presented in the rest of the thesis.

5.7.1 OR Semantics versus AND Semantics

For the sake of discussion here, I call the semantic model we finally adopted "OR semantics", the intuition being that for at least one, but not necessarily all, of the segments for which the actual parameters satisfy the left hand side (or parameter descriptors), the function result will satisfy the right hand side. This differs from "AND semantics", which would claim that for any segment whose left hand side is satisfied by the actuals, the right hand side would be satisfied by the result. AND semantics is the interpretation normally used in conjunction with the "->" in type notation.

The reason OR semantics was chosen over AND semantics has to do with the nature of Lisp evaluation and with how it affects the implementation of the type inference algorithm. Consider the function:

```
(DEFUN AND-OR-SEMANTICS-EXAMPLE (X)
  (IF (EQUAL X 3)
      10
      (IF (EQUAL X 5)
          'FOO
          T)))
```

We might wish for signature segments which would adhere to AND semantics:

```
($INTEGER) -> (*OR $INTEGER $NON-T-NIL-SYMBOL $T)

(*OR $CHARACTER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING $T
     (*CONS *UNIVERSAL *UNIVERSAL))
  -> $T
```

But producing a signature like this goes against the grain of an inference algorithm which will analyze the function in Lisp's recursive descent evaluation order. Ultimately, the results returned by the function will be computed at the tips of its IF structure. At each tip we have a context of the types of the formal variables, as determined by the guard and by the various IF tests through which control has passed, both in the function being analyzed and in functions called in the body, as captured in their signatures. This suggests that the orientation of the algorithm would be around generating a segment or a collection of segments at these tips. Since there can be a test like (EQUAL X 3), which is satisfied by some integers and not others, an $INTEGER parameter could factor into more than one of these tips.

A signature for AND-OR-SEMANTICS-EXAMPLE more in harmony with recursive descent analysis would be:

```
($INTEGER) -> $INTEGER
($INTEGER) -> $NON-T-NIL-SYMBOL
(*UNIVERSAL) -> $T
```

One segment is produced for each tip of the IF tree, using the type contexts provided by the accumulated IF tests governing the tip. This signature is not a good one under AND semantics, since it would suggest that if the argument were an integer, the result would be simultaneously an integer, a non-t-nil-symbol, and T. Given a preference for generating segments, at least initially, by recursive descent as described, we would need to significantly transform this signature for AND semantics in order to preserve much useful information. Ideally, we would transform it to the previously stated signature. This would require an elaborate factorization of the signature amounting to an inversion. We might, for instance, find a common descriptor among some of the left hand sides (like $INTEGER, in this case), and form new segments, first removing that common descriptor from all left hand sides where it occurs and then adding a segment mapping it to the disjunction of all the result types of the modified segments. In general, such manipulations could lead to loss of information. It seems much more straightforward to define a semantics which more neatly matches the results of a recursive descent derivation, i.e., OR semantics.

5.7.2 A More Constructive Semantics

The INTERP-SIMPLE model of semantics is of an abstract flavor because of the existential quantifier over vectors of values. Within the quantifier, though, is INTERP-SIMPLE, which is a constructive function. But we considered an even more constructive semantics which would have broadened the semantic function INTERP-SIMPLE to a new function INTERP which would encompass the quantifier. Given a list of type descriptors and a list (of the same arity) of values, INTERP would return either FAIL or a list of bindings lists, each of which would map type variables to Lisp expressions and would exhibit possible choices for the existential quantifier over values in the quantifier semantics.

```
Where we denote that if (INTERP (TD1i..TDni TDi) (O1..On (F O1..On)))
does not fail, it returns bindings i,1 .. bi,m,

(GUARD V1 .. Vn)
->
 (INTERP-SIMPLE (GD1..GDn) (V1 .. VN) NIL)
and
 for some i, 1 ≤ i ≤ k,
   (INTERP (TD1i..TDni TDi) (O1..On (F O1..On))) is not FAIL and
   for some j, 1 ≤ j ≤ m
   (INTERP-SIMPLE (TD1i..TDni TDi)
                  (V1..Vn (F V1..Vn))
                  bi,j)
```

That is, if the guard is satisfied by the actual parameters, for some segment, the guard descriptors are satisfied by the parameters, INTERP returns a non-empty list of substitution lists mapping the variables in the segment to a representation of Lisp values, under which the actual parameters satisfy the parameter descriptors in the segment and the result value satisfies the result descriptor.

To illustrate, consider again the CADR function. This interpretation means that if X satisfies the guard descriptor

```
(*OR $NIL (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))),
```

then INTERP will return a singleton list containing a singleton substitution, i.e., (((&20 . (CAR (CDR X))))). Thus, the "values" suggested by INTERP would be symbolic values, stated in terms of the variables in the formal parameter list.

This model was rejected in favor of the INTERP-SIMPLE semantics because this model placed the complexity of finding satisfactory vectors of values in the semantic definition functions, rather than in the inference system. The latter seems much preferable, since the discovery process is very complex and could involve heuristics which can be ignored formally, so long as the results they yield are checked formally. Thus, the semantic function INTERP-SIMPLE needs only to perform this check.

5.7.3 Other Models

We explored several other semantic models before settling on the INTERP-SIMPLE model. In none of the three alternate models mentioned below do we utilize the notion of value equality as the significance of multiple occurrences of type variables. Rather, the variable represents some unspecified *set* of data objects (or an unspecified recognizer function), uniformly over all occurrences, which is to say a variable may be instantiated by another type descriptor.

The first alternate model is described in terms of sets, and appears in Appendix C, titled "A Set-Based Semantics". At the end of this appendix is a somewhat extended version of the discussion of OR semantics versus AND semantics. Also included are proofs of several properties which would have contributed significantly to the proof of a system based on the model. This model was discarded because of a desire for a more computational style of semantics.

Another semantic model was posed in the vernacular of functions and meta-functions. Each descriptor corresponds to a recognizer function with a parameter for the value to be checked for satisfaction of the descriptor. A descriptor containing type variables corresponds to a function with one extra parameter for each type variable. These parameters are to be instantiated with functions corresponding to the type descriptors instantiating the variables. This direction was abandoned because its meta-theoretic approach posed more potential problems than we would encounter with a more straightforwardly computational model. In particular, the semantic definition itself generates function definitions, a notion which could become problematic. This direction was abandoned because its meta-theoretic approach was not, in the end, sufficiently computational for our taste, despite its functional trappings. It is described in the Appendix D, titled "A Function-Based Semantics".

The third model represented a final attempt to develop a computational style semantics in which type variables could be instantiated with other type descriptors of the general form. It is described in Appendix E, titled "A Semantics with Composed Substitutions". The difficulties encountered with this specification provided the impetus for moving to the semantics of variables which allowed for their instantiation only by Lisp values, or, more precisely, by types representing single values. This appendix shows, perhaps more clearly than the others, how we stumbled along in the process of developing the formal semantics.

All three of these models, when abandoned, were works in progress. As such, they are without doubt imperfect. They are provided both to show the various specification approaches attempted and to illustrate the difficulty of arriving at an approach which was appropriate to the problem and viable for proof of the algorithm.

Chapter 6
THE SIGNATURE CHECKER

This chapter describes the type checker algorithm, which validates signatures produced by the inference algorithm. The inference algorithm performs a number of operations which would be very difficult to specify and is not formally verified in any sense. In contrast, the checker employs relatively fewer and simpler concepts and is designed with verifiability in mind. Yet it is by no means a simple algorithm. A thorough understanding of the proof of its correctness will require an understanding of the description provided here.

In this chapter we first present an overview of the checker and its task. The remaining sections are devoted to describing each of the significant algorithms within the checker. With each, we present the lemmas providing a top level specification of the algorithm, along with a detailed description of the algorithm. In order, we will discuss the top level of the checker algorithm TC-SIGNATURE, the central recursive algorithm in the checker TC-INFER, the unification algorithm, the descriptor canonicalization algorithm, and the containment algorithm.

Throughout this chapter keep in mind that the "inference algorithm" is the unverified heuristic algorithm, described in Chapter 4, which generates a function signature, and the "checker algorithm", described here, is the one whose soundness we will formally prove.

6.1 The Top Level Specification

Let us first repeat the two lemmas which together make up the top level specification for the checker. The first and most important specifies that its validation of the inference tool signature is sound. Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
Lemma TC-SIGNATURE-OK

 For any n-ary function foo, whose definition is of the form
  (defun foo (a1 .. an)
    (declare (xargs :guard guard-form))
    body)
  where guard-form is a conjunction of recognizer calls on distinct
  formal parameters,
 for any world of Lisp functions world, including at least the above
  definition of foo and the definitions of all the functions in the
  call tree of foo,
 for any list of function signatures fs, including signatures for
  at least all the functions in the call tree of foo, except foo
  itself,
 for any non-negative integer clock,
```

```
 when (tc-signature foo fs) successfully validates a signature
  for foo,

H1  (and (valid-fs fs world clock)
H2       (and (not (equal (guard (tc-signature foo fs))
                          *guard-violation))
              (not (equal (segments (tc-signature foo fs))
                          *guard-violation)))
H3       (tc-all-called-functions-complete guard-form fs)
H4       (tc-all-called-functions-complete+ body fs foo t) )
    =>
    (valid-fs (cons (tc-signature foo fs) fs) world clock)
```

VALID-FS is defined in Section 5.5.

The next lemma justifies our claim about guard verification, which is that if a function's guard is an IF form representation of a conjunction of recognizer calls on distinct formal parameters, and if the checker does not detect a guard violation on a call to the function, then the real guard for the function, applied to the actual parameters of the call, will evaluate to T.

Lemma GUARD-COMPLETE

```
 Given a function of arity n with argument list (a1 .. an),
 and guard expression of the form
  (and (Ra1 a1) .. (Ran an))
 denoting a conjunction of calls to recognizer functions on distinct
 formal parameters, and where the recognizer function Rak
 has the segment (rtdk) -> $t,
 for any values arg1 .. argn, descriptors argtd1 .. argtdn,
 type variable binding b, non-negative integer clock, and a world of
 Lisp functions including all those in the call tree of the guard
 expression,

  (and
H1  (valid-fs fs world clock)
H2  (I (argtd1 .. argtdn) (arg1 .. argn) b)
H3  (contained-in-interface (*dlist argtd1 .. argtdn)
                            (*dlist rtd1 .. rtdn))
H4  (not (break-out-of-timep
           (E (and (Ra1 a1) .. (Ran an))
              ((a1 . arg1) .. (an . argn))
              world
              clock))) )
  =>
  (equal (E (and (Ra1 a1) .. (Ran an))
            ((a1 . arg1) .. (an . argn))
            world
            clock)
         t)

 Note: For the sake of uniformity in notation, let us say that
 there is one recognizer call for each parameter, where for
 parameters which are unrestricted in the guard expression we
 use a recognizer (DEFUN UNIVERSALP (X) T) whose segments
 are ((*universal) -> $t) and ((*empty) -> $nil).  In
 any real guard, any such recognizer call may be omitted from
 the guard without the loss of generality of this lemma.
```

6.2 Overview

The job of the signature checker is to validate the result produced by the inference algorithm. In essence, the checker performs the n+1st iteration of the iterative approximation algorithm whose first n iterations were performed by the main inference algorithm. But fortunately, its task is vastly simpler than that of the inference algorithm, for several reasons. Principal among these,

- The checker operates on a type descriptor language which is smaller than the one manipulated in the inference algorithm.
- The checker does not need to discover any closed recursive forms.
- The checker neither needs nor attempts any special tricks to handle recognizer functions.

For comparison of the descriptor languages the two algorithms handle, see their respective BNF grammars in Section 5.2 and Section 4.3. The checker needs to handle only the core language composed of primitive descriptors, *EMPTY, *UNIVERSAL, variables, and constructions from *OR, *CONS, and *REC. A number of other forms are employed in the inference algorithm in the search for closed form recursive descriptors. Most notable among these, because of the difficulties they would introduce, are:

- *NOT, the descriptor which prescribes the complement of the type prescribed by its argument,
- *FIX and *FIX-RECUR, the descriptors corresponding to a recursive form under consideration for closure, and
- *AND, a descriptor which, in a limited context, prescribes the set of values corresponding to the intersection of the value sets corresponding to its arguments.

Moreover, the checker adheres strictly to the singleton semantics for variables, which the inference algorithm bends somewhat in the search for closed recursive forms. By the time the inference algorithm has completed its work, all these interim forms have been discarded. This is fortunate, because even their formal specification, much less proofs about their manipulation, could be very difficult. By the time the checker receives a signature, it is expressed entirely in the core descriptor language, which is cleanly defined with the semantic definition functions.

After the inference tool computes a signature for a function and stores it in the database, the checker makes a single pass over the function, performing its own version of the type inference algorithm to get a new signature. In does so under the assumption that the signature segments which were formulated for the function by the inference algorithm are correct, under a rationale discussed briefly below. Since the guard contains no recursive calls to the function in question, the checker can formulate its own guard descriptors, using only verified signatures. It need not and does not make any assumptions about the correctness of the guard descriptors derived by the inference algorithm. It uses its own guard descriptors to formulate the initial type assumptions for analyzing the function body and to do guard checking on recursive calls.

After obtaining its own signature, the checker then performs a crucial test, using the predicate CONTAINED-IN-INTERFACE. It checks that for each segment in the checker signature, some binding of type variables to values exists such that under the binding, the segment is contained in some segment from the inference tool signature. If this is the case, and if the guards are complete for this function and every function in its call tree, then we consider the inference tool signature validated. Since the checker algorithm and the CONTAINED-IN algorithm are proven correct, if the signature passes these tests, we have full assurance of the correctness of the signature.

It is counterintuitive to consider using, for a recursive function, a signature we do not trust as the basis for deducing a signature which we do trust, since it seems that if the initial signature is incorrect, we are predicating our entire analysis on a false assumption. But imagine the graph of a recursive function as being the union of the graphs of an infinite number of non-recursive functions, with the first being the empty function, the second being the ordered pairs formed by cases where the function terminates without recursion, the next being cases where it terminates with one recursion, and so forth. For a given function f, call these non-recursive functions $\perp$, f^1, f^2, .. f^n. Each function definition f^i is non-recursive in the sense that it is formed by replacing the recursive calls of f with calls to f^{i-1}. An inductive argument, explained in detail in Section 7.2, can be used to demonstrate that if a containment relation holds between the signatures for f^{n+1} and f^n, then the signature for f^n is a sound approximation for the graph of f. The inference algorithm computes the signature for f^n, and the checker computes the signature for f^{n+1}.

If the containment relation holds, it means that the checker signature is a better approximation of the real function graph than the inference tool signature. Why, then, do we use the inference tool signature as our permanent result? The reason is that the inference tool signature is in a relatively compact form. By contrast, the checker signature usually reflects an expansion of cases with respect to the base signature. For each segment in the base signature, there may be many segments in the checker signature. Moreover, the type descriptors in the checker signature may be larger, reflecting, for example, the opening of a *REC descriptor. Furthermore, there may be a combinatorial explosion in the factoring of possible cases among the formal parameter descriptors. We sacrifice compactness in the checker signature for simplicity in the checker algorithm, where the soundness and the clarity of the code and its proof of correctness are primary requirements. So while the checker signature may be marginally more accurate than the original, we would pay a severe price in case explosion if we used it rather than the inference tool signature as our permanent reference. Since our proof validates the soundness of both, we choose the latter for the sake of efficiency as the basis for further inference computations.

The signature checker, like all the rest of the inference system implemention (save the highest level interactive function, which modifies the system state as functions are added) is written as a composition of purely applicative Common Lisp functions.

We will factor the detailed discussion of the checker algorithm into components, corresponding to its principal functional composition. Each component will be discussed in a separate section. The top level of the checker, invoked by calling TC-SIGNATURE, is discussed first. The top level function for deriving the checker's signature from the function text and the inference algorithm's signature is TC-INFER-SIGNATURE. The principal recursive function of the checker algorithm, the one which actually traverses the function guard and body, is TC-INFER. This central algorithm will be discussed and specified next. As with the inference algorithm, one of the main internal algorithms of the checker is its unification algorithm for type descriptors. We will outline how unification occurs and give the formal specification for the unifier, whose top level function is DUNIFY-DESCRIPTORS-INTERFACE. Next, we will discuss the checker's type descriptor canonicalization routines, which are employed throughout the checker. Finally, we will discuss the containment algorithm, whose top level function is CONTAINED-IN-INTERFACE. This algorithm provides the test which validates the correctness of our signature.

6.3 The Top Level of the Checker

TC-SIGNATURE, the top level function of the checker, essentially solicits and packages the results of the checker's analysis. The top level functions in the checker do the following.

1. Initially, they transform both the guard and the function body, using the function TC-PREPASS. TC-PREPASS normalizes IF forms by transforming them so the IF test (the first argument of the IF) always returns either T or NIL, and by performing the obvious simplification if the resulting predicate form is either of the constants T or NIL. Typically,

normalization consists of wrapping two calls to NULL around the test, if necessary. For technical reasons, the inference tool can be more precise when IF tests always return T or NIL. A simple example giving intuition about why this is true is when a formal parameter X appears as a test. Suppose in the context of the IF, X has a type which includes $NIL and some other possibilities. For example,

```
(*REC TRUE-LISTP
      (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
```

If X were left bare, barring the special treatment of IF which would complicate the semantics of the checker, the segment for the IF test would be the single segment whose result is the type of X. As we shall see later in the discussion of the checker algorithm, the type context would not be refined for X in the THEN and ELSE arms. But transforming the IF test to (NULL (NULL X)) tricks the checker into refining the context as a result of applying the segments for NULL. The formal soundness property of TC-PREPASS, which is stated in Lemma TC-PREPASS-OK and proved in Appendix B.1, is that in any environment and for any form, evaluating the prepassed form in the environment produces the same value as evaluating the original form in the environment. The definition of TC-PREPASS appears in Appendix G.6.

2. We determine whether the guard descriptors are complete. This is an entirely syntactic test on the guard expression, requiring only the knowledge of what functions in the world are known to be recognizer functions. A recognizer call is either a call to a recognizer function (See Section 5.6) or an equality test of the form (EQUAL <param> <value>) or (EQUAL <value> <param>), where <param> is one of the formal parameters and <value> is either T or NIL (the only singleton primitive types). The guard descriptors will be complete if the guard is an IF form representation of the conjunction of recognizer calls on the formal arguments, with no argument subject to more than one such call. For example, the IF form representation of

   ```
   (AND (INTEGERP X) (CONSP Y) (CHARACTERP Z))
   ```

   is:

   ```
   (IF (INTEGERP X) (IF (CONSP Y) (CHARACTERP Z) NIL) NIL)
   ```

   Though this restriction in form may on the surface seem more onerous than that employed in the inference algorithm, it is usually the case that a user could transform a guard conforming to the inference algorithm regimen to one which suits the checker’s rules.

3. We determine whether all the functions called in the body have complete guards. Stored with each function in the database is a flag which indicates whether its guard is complete according to the checker regimen above.[26] So this check is just a lookup on all called functions.

4. Regardless of whether the guard is complete, we derive the vector of guard descriptors. We do this by first obtaining segments for the guard by calling TC-INFER on the prepassed guard form with an abstract association list (or *alist*) simply associating a new type variable with each formal parameter, and with a concrete alist associating *UNIVERSAL with each formal. (Section 6.5 will describe the significance of these alists and clarify some of the following.) If a guard violation is found in the guard itself, we note this and do not attempt to validate the function segments. But in the normal case, we next gather all the maximal segments for which the result descriptor is anything other than $NIL, replace all occurrences of singleton variables (i.e., a variable which appears only once within a segment) with *UNIVERSAL, abort the computation if there are any remaining variables, and then form a vector of guard descriptors, one for each argument, by *OR-ing the contexts of the qualifying segments together and canonicalizing.

[26]The flag for the current function is passed in as a parameter, since it has not yet been stored in the system database.

5. If no guard violation is found in the guard, we derive the checker segments for the function body by calling TC-INFER with the body, the same abstract alist as with the guard, and a concrete alist pairwise associating each parameter with its guard descriptor, or if the guard descriptor is *UNIVERSAL, with the variable from the abstract alist. If a guard violation is detected in this analysis, we simply note such as the result. Otherwise, we return the list of maximal segments computed by TC-INFER. (For a discussion of maximal segments, see Section 6.5.

6. The inference algorithm also computed and stored a guard descriptor. In rare cases, the two guard descriptors are equivalent but different. In these cases, more often than not, the inference algorithm descriptor is in a nicer form, meaning that it may not contain some redundancy that the checker guard descriptor has. So there is an advantage in using the inference algorithm guard as the "official" one, in the sense that it is the one which will be used for all subsequent purposes. If the two guard descriptors are different, the checker determines whether they are equivalent by administering the containment test (described below in Section 6.8) in both directions. If each is contained in the other, they may be used interchangeably, so the tool chooses to use the inference algorithm guard. If either test fails, the checker's guard will be the one used for all subsequent purposes.

7. We administer the containment test for the checker segments, as described above. If containment is not determined, we declare the signature to be unsound, and do not bother to proceed.

8. If the segments are properly contained, then we check to see if the inference algorithm noted that the function being analyzed is a recognizer. If so, the checker validates this fact by doing the following tests on the now-validated segments in the inference tool signature.
    a. There must be no type variables in the segments.
    b. There are two segments, one with a $T result and the other with a $NIL result.
    c. We unify the argument descriptors of the two segments, and the result must be *EMPTY, signifying that there is no overlap of the domains for the two segments.

9. Finally, we return a structure which encapsulates all we have learned, with flags denoting the results of the various tests just performed. It is a 7-tuple, containing the checker guard, the guard containment flag, the guard complete flag, the ALL-CALLED-FUNCTIONS-COMPLETE flag, the checker segments, the segments contained flag, and the flag denoting whether the function was validated as a recognizer. (These flags were defined in Section 5.6.)

The interpretation of these results is that if all the above flags (excepting possibly the RECOGNIZERP flag) are T, then the signature is perfect, in the sense that it fully satisfies the formal specification GOOD-SIGNATUREP for function signatures. As such, it may be employed as part of a valid fs to make sound inferences in deriving the signatures of succeeding functions.

The formal specification of the top level of the checker is given by Lemmas TC-SIGNATURE-OK and GUARD-COMPLETE, which were presented in Section 6.1.

6.4 Guard Verification

In E, our model of evaluation for our Lisp subset, on every function call, we first bind the formal parameters to the actual parameter values, then evaluate the guard form of the called function in that binding environment. If the guard evaluates to NIL, a guard violation error occurs. If the guard evaluates to a non-NIL value, the body of the function is evaluated, and the value returned from the function call is the result returned from that evaluation. The native functions in our dialect are defined as *subrs*, meaning that if the guard evaluates to a non-NIL value, the value returned is computed by a table lookup, and no error can occur.[27]

Imagine imposing a requirement on new function definitions that this property is preserved, i.e., that if the guard for a function is non-NIL when evaluated on some actual parameters, then we are assured the function body will evaluate without error and return a value. Thus, roughly speaking, the function guard safely captures the domain restriction of a partial function. The purpose of guards in our Lisp subset, then, is to provide a predicate for specifying conditions on the arguments to a function which will guarantee that, if the function is called with arguments satisfying the predicate, evaluation of the function will proceed without error and produce a value.

The notion of *verifying a guard* is one of discharging the proof obligations necessary to demonstrate for a given function call in a function definition, that for any possible values of the definition's parameters, the guard form for that function will evaluate to a non-NIL value. There is one such obligation for each function call appearing in the definition of the function. The obligation says that the actual parameters satisfy the called function's guard under the assumptions derived from the IF tests governing the function call. For function calls in a function body (as opposed to function calls appearing in the guard form), we can also assume that the function's guard evaluated to a non-NIL value.

Assurance of the guard property would require a regimen where part of the process of defining a new function is discharging the guard proof obligation for each function call it contains. Together with a proof of termination, this would be sufficient to have the assurance we seek. This is because for functions called non-recursively, all guard proof obligations have already been discharged, so by satisfying the guard of each such function, we have satisfied the assumptions made in those proof obligations. For the recursive calls, the same argument applies, but the justification involves a computational induction on the arguments. This is essentially the strategy employed in the developmental Acl2 system [Boyer 90].

Using the evaluation model defined by our evaluator function E (in Section 5.4), we can prove that guard validation regimen just described is sufficient to guarantee the no-error property we hypothesized. Lemma GUARD-COMPLETE provides the basis for the inference system's role in performing guard verification, which is ultimately stated in Lemma TC-SIGNATURE-OK. Whenever the guard of a function is stated as a conjunction of recognizer calls on distinct parameters, and likewise throughout the call tree of the function, then our type inference system, by generating a signature without detecting a guard violation, satisfies all the guard proof obligations for our new function.

We could imagine our system accomplishing the verification of guard proof obligations by acting in one of at least two different roles. One way would be to view the inference system as generating and relieving each proof obligation implicitly as it analyzes a new function definition, as it does now. The other would be for some external proof obligation generator to formulate the proof obligation as an implication (cast as an IF form), and then to use the inference tool to derive a typing for that form whose result would be disjoint from $NIL. (Since recognizers are Boolean, the result would be $T.)

The stylistic constraint imposed on guards as a requirement for completeness is consistent with the capabilities of the type system. It does not allow for relations between the types of various arguments, such as

```
(OR (INTEGERP X) (SYMBOL-ALISTP Y))
```

[27] Alternatively, they could have been defined in terms of "sub-primitive" functions which are guaranteed to evaluate without error and return a value whenever the guard is satisfied. This strategy might be used for an implementation of our Lisp subset.

or

```
(MEMBER X Y)
```

as such restrictions make a link between the types of two variables which cannot be represented by the guard vector. It does not allow for guard expressions which depend on non-type-theoretic information, such as (PRIMEP X).[28] It also rules out some guard expressions where our guard descriptor is sufficiently precise, but in these cases, the requisite predicate could be re-phrased as a conjunction of recognizer calls. There are unlimited examples of this phenomenon, but to illustrate, the guard

```
(AND (CONSP X) (TRUE-LISTP X))
```

which would be rejected as not being complete because it is a conjunction of recognizer calls on *non-distinct* parameters, could be recast as a function call

```
(PROPER-CONSP X)
```

where, PROPER-CONSP, defined as follows, would qualify as a recognizer function:

```
(DEFUN PROPER-CONSP (X)
  (IF (CONSP X) (TRUE-LISTP X) NIL))
```

Another example is the guard (NOT (CONSP X)), which could also be encapsulated as a function which would qualify as a recognizer. So, many guards which do not satisfy the type system's requirements for guard completeness can be transformed into satisfactory guards via proper encapsulation.

6.5 The Central Algorithm TC-INFER

The central function in the type checker is TC-INFER. TC-INFER takes four arguments, a Lisp FORM being analyzed, an abstract type alist ABS-ALIST, a concrete type alist CONC-ALIST, and the database of function signatures, FS (FUNCTION-SIGNATURES in the code). It returns a list of 3-tuples, where each tuple is composed of a minimal segment, a concrete type alist, and a maximal segment.[29] Both minimal and maximal segments are of the same form, i.e., if the function we are checking has n arguments, each is an (n+1)-tuple of type descriptors, but can carry slightly different information. The concrete type alist has n entries and characterizes the types of the Lisp formals when the segment is a possible choice. It is sometimes more restrictive than the CONC-ALIST parameter, as we shall see later.

An ABS-ALIST associates each formal parameter with a descriptor, where this descriptor is typically a type variable or a structured descriptor with type variables embedded. A CONC-ALIST associates each formal with a descriptor which is typically biased toward descriptors other than variables (though variables are not excluded). Similarly, minimal segments are biased toward variables, and maximal segments biased toward other descriptors.

The need for variables is to capture the transfer of a particular piece of data from the parameters to the result, as with the application of a function like CAR. But often the advantage of variables is lost, for instance when their manipulation would force them into the recursive depths of *REC descriptors, where

---

[28]It is not sufficient for the argument to be an integer, and the type descriptor language could not capture the characteristic that a given integer is a prime number.

[29]The terms "minimal" and "maximal" are somewhat arbitrarily chosen. The sense of it is that the maximal segments carry the maximum information about type structure, while the minimal segments have a bias toward the use of type variables, emphasizing value sharing rather than type structure.

they are not allowed. Thus, it is frequently more desirable to carry the best information we can without variables, as with the concrete alist and the maximal segment. Were it not for a weakness in the type descriptor language, we might be able to satisfy both needs simultaneously. If the descriptor language had a construct for denoting that an object in a particular position is both identified with a type variable and characterized by another kind of descriptor [for example (*BOTH &1 (*CONS $INTEGER $NIL))], then there would be no need for the dichotomy of representation in the checker. But attempts to support this kind of construct were aborted because of the complexity it introduced into an already very complicated system. Given that there is no blended construct, the checker compensates by carrying multiple representations of both the formal parameters of the function and the segments produced by the checker.

6.5.1 The Formal Specification

The formal specification for TC-INFER is in terms of the function INTERP-SIMPLE, which establishes the consistency of Lisp values with type descriptors under a binding environment for the type variables in the descriptor. Essentially, the specification is that for any values supplied for the variables in the environment (i.e., those in the alists), if we have a valid database of signatures, a valid ABS-ALIST, and a valid CONC-ALIST, if the checker does not detect a guard violation, the guards of all the functions in the call tree of the form are complete, and clock is sufficient to allow a full evaluation of the form in the binding environment, then evaluation of the form will not result in a guard violation, and TC-INFER produces a valid list of 3-tuples. An ABS-ALIST or CONC-ALIST is valid if for each variable in the environment, the descriptor associated with the variable correctly characterizes the value of the variable, as witnessed by INTERP-SIMPLE. A list of 3-tuples is valid if there is at least one tuple in the list such that the CONC-ALIST and both the minimal and maximal segments are valid with respect to the values of the variables in the environment and the value produced by evaluating the form in that environment. Segment validity is also defined in terms of INTERP-SIMPLE. The segments in the conclusion may contain variables not in the original hypotheses, since variables can be imported with the segments for a called function. Thus, the type variable binding used by INTERP-SIMPLE in the conclusion may be different from the one in the hypotheses, but it is an *extension* of the original, meaning that all the individual variable-value pairs in the original are also in the extended binding. Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

Lemma TC-INFER-OK

For any Lisp form form, function signature list fs, Lisp world world
  including all functions hereditarily in the call tree of form,
for any non-negative integer clock, type variable bindings b,
 Lisp values arg_1 .. arg_m,
 Lisp variables a_1 .. a_m,
 binding environment env of the form ((a_1 . arg_1) .. (a_m . arg_m))
  where a_1 .. a_m include all the free variables in form,
 ABS-ALIST of the form ((a_1 . tda_1) .. (a_m . tda_m)),
 CONC-ALIST of the form ((a_1 . tdc_1) .. (a_m . tdc_m)),
and denoting
  (tc-infer form abs-alist conc-alist fs)
 by
  ((($mintd_{1,1}$ .. $mintd_{1,m}$ $mintd_1$)
    ((a_1 . $tdconc_{1,1}$) .. (a_m . $tdconc_{1,m}$))
    ($maxtd_{1,1}$ .. $maxtd_{1,m}$ $maxtd_1$))
   ..
   (($mintd_{n,1}$ .. $mintd_{n,m}$ $mintd_n$)
    ((a_1 . $tdconc_{n,1}$) .. (a_m . $tdconc_{n,m}$))
    ($maxtd_{n,1}$ .. $maxtd_{n,m}$ $maxtd_n$)))

H1 (and (valid-fs fs world clock)

```
H2      (I (tda1 .. tdam) (arg1 .. argm) b)
H3      (I (tdc1 .. tdcm) (arg1 .. argm) b)
H4      (not (equal (tc-infer form abs-alist conc-alist fs)
                    *guard-violation))
H5      (tc-all-called-functions-complete form fs)
H6      (not (break-out-of-timep (E form env world clock))) )
   =>
   (and
C1   (not (break-guard-violationp (E form env world clock)))
C2   for some i,
      for some binding b' covering the descriptors below,
       (and (I (mintdi,1 .. mintdi,m mintdi)
               (arg1 .. argm (E form env world clock))
               b')
            (I (maxtdi,1 .. maxtdi,m maxtdi)
               (arg1 .. argm (E form env world clock))
               b')
            (I (tdconci,1 .. tdconci,m) (arg1 .. argm) b')
            (extends-binding b' b)) )

Note:
H1 establishes that the signatures in the system state fs are valid.
H2 establishes that the abs-arglist is valid.
H3 establishes that the conc-arglist is valid.
H4 establishes that no guard violations are detected in the
   course of analyzing form.
H5 establishes that the guards of all functions in the call tree
   of form are complete.
H6 establishes that the evaluation of form terminates without
   exhausting the clock.
```

6.5.2 Detailed Description of TC-INFER

The Lisp dialect which the inference system supports is simple. There are only four different kinds of forms: variables, quoted forms (including self-quoting forms like integers, strings, T, and NIL), IF forms of arity three, and function calls. This section will describe how TC-INFER handles each of these.

Recall that TC-INFER takes as arguments one of these Lisp forms, an ABS-ALIST and a CONC-ALIST, both association lists mapping the variables in the environment to type descriptors, the name of the function in which the form occurs, the guard descriptor for that function, and the system state, FS. (FS is the name used to refer to the state in the exposition and proof. FUNCTION-SIGNATURES is the name used in the implementation code.) The function name and guard descriptor are used only to perform guard verification on recursive calls; the name allows recognition that a function call is recursive, and the guard must be provided as a parameter because the guard computed by the checker for the current function is not in FS. TC-INFER returns a list of 3-tuples, where each tuple is composed of a minimal segment, a concrete alist, and a maximal segment:

```
(((tdai,1..tdai,n) -> tdai)        << the minimal segment >>
 ((v1 . tdv1) .. (vn . tdvn))  << the concrete alist >>
 ((tdci,1..tdci,n) -> tdci)        << the maximal segment >>

where n = the arity of the function being checked
```

(The "->" is included as a reading aid. It is not actually part of the data structure.) The concrete alist represents the type context in which the segments were computed.

Variable occurrences are easy. We simply look up the type in the type alists. We return a list of a single 3-tuple, as follows. The minimal segment's context component is the list of the types of the parameters, drawn from ABS-ALIST, and its result component is the type associated with the variable in question, also taken from ABS-ALIST. The concrete alist in the tuple is the same as the input CONC-ALIST. The maximal segment's context component is the list of parameter types, taken from the CONC-ALIST, and the result is the type associated with the variable in CONC-ALIST. For example,

```
(TC-INFER 'Y                                            ; FORM
          '((X . &1) (Y . &2))                          ; ABS-ALIST
          '((X . $INTEGER) (Y . (*OR $NIL $T)))         ; CONC-ALIST
          'FOO                                          ; FNNAME
          '(*UNIVERSAL *UNIVERSAL)                      ; GUARD
          FS)
 =
((((&1 &2) -> &2)                               << the minimal segment >>
  ((X . $INTEGER) (Y . (*OR $NIL $T)))          << the concrete alist >>
  (($INTEGER (*OR $NIL $T)) -> (*OR $NIL $T)))) << the maximal segment >>
```

Thus, both segments map objects of the types indicated by the respective type alists to the binding of Y in those alists.

Quoted forms are also straightforward. As recognized by a simple predicate, a quoted form is either an integer (as determined by the Lisp function INTEGERP), T, NIL, a rational literal as determined by the Lisp function RATIONALP, a character (CHARACTERP), a string (STRINGP), a keyword (KEYWORDP), or a CONS whose CAR is the atom QUOTE. The descriptor is constructed by the function DESCRIPTOR-FROM-QUOTE, whose definition appears in Appendix G.7. The result returned by TC-INFER is constructed exactly as with the identifier case, except that the descriptor constructed by DESCRIPTOR-FROM-QUOTE appears in the result component of both segments instead of the type descriptor extracted from the type alists. Lemma DESCRIPTOR-FROM-QUOTE-OK provides the soundness specification for DESCRIPTOR-FROM-QUOTE.

Not surprisingly, function calls are the most involved case. First TC-INFER calls itself recursively on all the function arguments, using the same type alists. If *GUARD-VIOLATION is returned for any of them, it returns *GUARD-VIOLATION.

Next, it checks for a guard violation on the function being called. From the recursive calls of TC-INFER noted above, each actual parameter has associated with it a list of triples, each of which may have a different result type. For each parameter, all these result types are combined using *OR, and the result canonicalized. A vector, of the arity of the argument list and representing the possible types of the arguments, is constructed from these results. All variables in the vector are transformed to *UNIVERSAL. Note that both this universalization and the *OR combination are operations which, if anything, enlarge the collection of values represented in the descriptors. Having thus prepared this vector representing a conservatively large set of possible values for the actual parameters, we employ the containment relation, calling CONTAINED-IN-INTERFACE, to ascertain if our actual parameter vector is contained in the vector of guard descriptors computed for the function. If not, we are unable to demonstrate that a guard violation will not occur, so we return the atom *GUARD-VIOLATION, effectively terminating the inference process.

Now comes the task of generating appropriate segments. The general idea is to consider each possible type configuration for the actual parameters in conjunction with each segment for the called function to determine a set of possible results for each compatible combination. The operation which performs this joint consideration is descriptor unification. We unify two vectors, one containing information about the actual parameters and calling context, and the other containing information from the function signature. The values represented in the vectors are the values of the variables in the context, the values of the actual parameters, and the value of the result.

Finding each possible type configuration for the actuals is a complex operation when there is more than one actual parameter, since each parameter may be associated with multiple segments, and each segment may have a different context description, i.e., each segment may be the result of a different type configuration for the variables in the environment. The contexts derived for one parameter are not likely to neatly match the contexts for another parameter. So to determine the collection of possible contexts, we perform a cross product operation.

The top level function of the cross product operation is TC-MAKE-ARG-CROSS-PRODUCT. It has a single parameter, ARG-RESULTS, which is a list whose length is the arity of the function being called, and whose elements are the lists of 3-tuples returned by TC-INFER on the respective arguments. The operation takes place in two phases, where first we re-organize the data in ARG-RESULTS to prepare it for the second phase, the combining operation.

Everything is best illustrated with an abbreviated example, in which we use the following notation. c1..c6 represent minimal segment contexts, each being a vector of descriptors of length equal to the number of formal parameters in the function being analyzed (not the function being called). r1..r6 represent the corresponding segment result types. The alistij are the concrete alists from the 3-tuples. The maxsegij are the maximal segments, which are ignored for the formation of not only the cross product but even for the value of TC-INFER on a function call. The alistij-rhs are vectors of descriptors composed of the right hand sides of the alists alistij. In our example, the function being called has three arguments, the first argument produced three 3-tuples, the second argument just one, and the third argument two. Thus, we have

```
ARG-RESULTS = ((((c1 -> r1) alist11 maxseg11)
                ((c2 -> r2) alist12 maxseg12)
                ((c3 -> r3) alist13 maxseg13))
               (((c4 -> r4) alist21 maxseg21))
               (((c5 -> r5) alist31 maxseg31)
                ((c6 -> r6) alist32 maxseg32)))
```

The re-organization of ARG-RESULTS produces the following structure, a list of length equal to the number of (order-preserving) combinations of segments in ARG-RESULTS, whose elements have arity of the function being called (in this case 3).

```
(((c1 (r1) alist11-rhs) (c4 (r4) alist21-rhs) (c5 (r5) alist31-rhs))
 ((c1 (r1) alist11-rhs) (c4 (r4) alist21-rhs) (c6 (r6) alist32-rhs))
 ((c2 (r2) alist12-rhs) (c4 (r4) alist21-rhs) (c5 (r5) alist31-rhs))
 ((c2 (r2) alist12-rhs) (c4 (r4) alist21-rhs) (c6 (r6) alist32-rhs))
 ((c3 (r3) alist13-rhs) (c4 (r4) alist21-rhs) (c5 (r5) alist31-rhs))
 ((c3 (r3) alist13-rhs) (c4 (r4) alist21-rhs) (c6 (r6) alist32-rhs)))
```

The second phase of the cross product operation is to take each triple and smash it into one form, using DUNIFY-DESCRIPTORS-INTERFACE (described in Section 6.6), rejecting empties and duplicates, so we return a list whose elements are all distinct and of the form

```
((c -> (ri rj rk)) alist)
```

where c is the unification of the contexts from the triple, (ri rj rk) is the list of result values from the triple, and alist is the unification of the conc-alists.

For example, the first triple in the intermediate structure yields a list of results of the form

```
((c -> (r1 r4 r5)) alist-rhs)
```

where c is a unification of c1, c4, and c5, and alist-rhs is a unification of alist11-rhs, alist21-rhs, and alist31-rhs. There is a list of such forms because if the unification of c1, c4, and c5, and the alists returns an *OR, we construct a result for each disjunct, which in general means we returns a list of results. If the unification is not an *OR, the list is of length one. Since unification is a binary operation, the unification of the triple occurs pairwise. First we unify

```
(c1 (r1 *universal) alist11-rhs)
```

and

```
(c4 (*universal r4) alist21-rhs)
```

(Actually, we flatten these structure and prefix them each with *DLIST, so the unification can be handled as a simple unification of descriptor lists. Notice the padding of the second element of each piece, since each ri represents a distinct actual parameter in our function call. Then each result is further padded and unified with

```
 (c5 (*universal *universal r5) alist31-rhs))
```

Thus, the grand result of the cross product operation is a profile of possible types for our actual parameters, including the types of the Lisp variables in the context under each scenario. The result is a list of forms which represent all the combinations of type descriptors characterizing the types of variables in the environment and the types of the actual parameters to our function call, along with the concrete alist appropriate to each combination. In the sense that unification is a narrowing operation which excludes from its result pieces of the world which are not common to both parameters, the parts which are excluded here represent combinations of variable and parameter types which cannot possibly occur.

Having taken the cross product, the next phase of handling a function call is to unify each result with each of the signature segments for the function being called. This will allow us to project a type for the result of the function call. The unifier, given two vectors of length i, will produce a (possibly empty) list of i-ary vectors. Our vectors correspond to the following configuration:

(vtd$_1$..vtd$_n$ ptd$_1$..ptd$_m$ rtd)

```
where n = the number of variables in the context
      m = the number of parameters in the function call
      vtd_i = the descriptor characterizing the i-th variable
             in the context
      ptd_j = the descriptor characterizing the j-th parameter of the
             function call
      rtd = the descriptor characterizing the result of the function
            call
```

Now consider the two vectors that we unify to produce such configurations. The first vector, which characterizes information from the calling context, is specifically of the form:

(ctd$_1$..ctd$_n$ atd$_1$..atd$_m$ *universal)

```
 where ctd_j is the descriptor for the jth variable in
           the environment, from the cross product element
       atd_k is the descriptor for the kth actual parameter,
           also from the cross product element
       *universal represents the type of the function call result
           without constraining it
```

The vector derived from the signature segment is as follows:

(*universal$_1$ .. *universal$_n$ ftd$_1$..ftd$_m$ rtd)

```
 where each *universal_i represents the types of the variables
```

```
    in the environment without constraining them
ftdk = the descriptor from the segment characterizing
   the type of the kth formal parameter
rtd = the descriptor from the segment characterizing the result
```

The key function which constructs these vectors, unifies them, and composes the resulting 3-tuples is TC-MATCH-ACTUAL-HALF-SEG-TO-FORMAL-SEG.

After doing this unification, we are close to having our results for the function call. Given our unification result, we need to compose a list of 3-tuples for our final result. (Remember that we perform the above unification once for each pairing of cross product results and signature segments, so there may be many of these cases to consider. All the resulting 3-tuples will be gathered into a single list to be returned by TC-INFER.) Each of the single unified vectors from the segment-matching unification will ultimately produce a list of result 3-tuples. This is because one final unification needs to be performed in order to construct the maximal segment. For each of these vectors, however, we already have the contents of the minimal segment and the concrete alist. The minimal segment is composed directly from the segment-matching vector. Its context is the first n descriptors in the vector, i.e., the ones corresponding to the variables in the environment. The result descriptor is taken from the final descriptor in the vector. The concrete alist returned with each tuple is the one from the cross product element which contributed to this result.

Computing the maximal segment is what requires the last unification. We need to inject the type information from the concrete alist of the 3-tuple into its minimal segment. So we unify the (flattened) minimal segment with a vector formed by the right hand sides of that concrete alist and a *UNIVERSAL descriptor, the latter corresponding to the result type. If the unification produces no results, this means that the bindings of the identifiers in the CONC-ALIST are incompatible with this particular result from the segment-matching unification, and so no 3-tuple is returned for this combination. If multiple results are produced, we make a 3-tuple for each one. The maximal segment is constructed directly from the vector returned by this final unification.

Obviously, given the right descriptors in the setting of a function call, quite a case explosion can occur. Even though the checker's operations on function calls are fairly complex, they would be much more complicated if steps were taken to minimize the propogation of cases. This is the price of keeping the algorithm "simple". The payoff is in relative clarity of the code and the soundness proof.

An concrete example showing the steps described above for a function call appears just below in Section 6.5.3.

Having dispensed with function calls, we need only cover IF expressions to finish the discussion of TC-INFER. TC-INFER's handling of IF expressions is relatively straightforward. First, it calls itself recursively on the first argument of the IF, the test form, with the same alists provided for the IF. Here, as with any other point in the computation, if the guards fail to be validated for any function call, *GUARD-VIOLATION is returned and percolates back to the top level call.

The analysis of the IF test returns the usual list of 3-tuples. Each element of this list provides a context for considering the THEN and ELSE forms. For each result, if the result type (as characterized by the result component of the maximal segment) can possibly be something other than NIL, then we call TC-INFER recursively on the THEN form, with the same ABS-ALIST and with a CONC-ALIST formed from the context descriptors of the maximal segment. Likewise, if the result type can possilby be NIL, then we call TC-INFER recursively on the ELSE form. All the resulting segments, from these recursive calls for each context provided by the TEST form, are appended to form the result for the IF form.

6.5.3 Example -- TC-INFER Handling a Function Call

This section contains a pedagogical example detailing the steps which TC-INFER takes in handling a function call. A full, general discussion of the algorithm appears in Section 6.5.2 above. The best way to read this section would likely be to follow the description in that section along with each illustrated step in the example below.

Consider that our database already includes the following collection of functions, whose signatures, as computed by the inference system, are provided. Recognizer descriptors are shown where appropriate. Note that we use the "->" in this section to separate the left and right hand sides of a segment. This is for descriptive purposes only. The "->" does not appear in any concrete representations of segments.

```
(DEFUN BAR-Y-GUARDP (Y)
  (IF (INTEGERP Y) T (CHARACTERP Y)))

Function: BAR-Y-GUARDP
TC Guard:
 (*UNIVERSAL)
Segments:
 (((*OR $CHARACTER $INTEGER)) -> $T)
 (((*OR $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING $T
        (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $NIL)
Recognizer descriptor:
  (*OR $CHARACTER $INTEGER)

(DEFUN BAR (X Y)
  (DECLARE (XARGS :GUARD (IF (TRUE-LISTP X) (BAR-Y-GUARDP Y) NIL)))
  (IF (NULL X)
      NIL
      (CONS Y (BAR (CDR X) Y))))

Function: BAR
TC Guard:
 ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
  (*OR $CHARACTER $INTEGER))
Segments:
 (($NIL (*OR $CHARACTER $INTEGER)) -> $NIL)
 (((*CONS *UNIVERSAL
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $CHARACTER)
   -> (*CONS $CHARACTER
             (*REC !REC8 (*OR $NIL (*CONS $CHARACTER (*RECUR !REC8))))))
 (((*CONS *UNIVERSAL
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $INTEGER)
   -> (*CONS $INTEGER
             (*REC !REC9 (*OR $NIL (*CONS $INTEGER (*RECUR !REC9))))))

(DEFUN BIM (X Y)
  (DECLARE (XARGS :GUARD (TRUE-LISTP X)))
  (IF (NULL X) NIL Y))

Function: BIM
TC Guard:
 ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
  *UNIVERSAL)
Segments:
 (($NIL *UNIVERSAL) -> $NIL)
 (((*CONS *UNIVERSAL
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (*FREE-TYPE-VAR 1.))
```

```
   -> (*FREE-TYPE-VAR 1.))

(DEFUN FOO-Y-GUARDP (Y)
  (IF (INTEGERP Y) T (IF (CHARACTERP Y) T (NULL Y))))

Function: FOO-Y-GUARDP
TC Guard:
 (*UNIVERSAL)
Segments:
 (((*OR $CHARACTER $INTEGER $NIL)) -> $T)
 (((*OR $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING $T
        (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $NIL)
Recognizer descriptor:
  (*OR $CHARACTER $INTEGER $NIL)

(DEFUN FOO (X Y)
  (DECLARE (XARGS :GUARD (IF (TRUE-LISTP X) (FOO-Y-GUARDP Y) NIL)))
  (IF (NULL X)
      NIL
      (IF (INTEGERP (CAR X))
          (CONS (CAR X) Y)
          (IF (CHARACTERP (CAR X))
              (CONS (CAR X) NIL)
              NIL))))

Function: FOO
TC Guard:
 ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
  (*OR $CHARACTER $INTEGER $NIL))
Segments:
 (((*OR $NIL
    (*CONS (*OR $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING
                $T (*CONS *UNIVERSAL *UNIVERSAL))
           (*REC TRUE-LISTP
                 (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))))
   (*OR $CHARACTER $INTEGER $NIL))
   -> $NIL)
 (((*CONS $INTEGER
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $CHARACTER)
   -> (*CONS $INTEGER $CHARACTER))
 (((*CONS $INTEGER
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $INTEGER)
   -> (*CONS $INTEGER $INTEGER))
 (((*CONS $INTEGER
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $NIL)
   -> (*CONS $INTEGER $NIL))
 (((*CONS $CHARACTER
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (*OR $CHARACTER $INTEGER $NIL))
   -> (*CONS $CHARACTER $NIL))
```

Now consider the function below.

```
(DEFUN TC-INFER-EXAMPLE (X Y)
  (DECLARE (XARGS :GUARD (IF (TRUE-LISTP X) (BAR-Y-GUARDP Y) NIL)))
  (FOO (BAR X Y) (BIM X Y)))
```

We will focus on the call to FOO in the body.

The guard establishes the following concrete type alist at the point of the call to FOO.

```
((X . (*REC TRUE-LISTP
            (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
 (Y . (*OR $CHARACTER $INTEGER)))
```

The 3-tuples produced for (BAR X Y) contain the following segments. (For each tuple, the minimal and maximal segments are the same, and the CONC-ALIST is the same as the type alist above.

```
(($NIL $CHARACTER) -> $NIL)
(($NIL $INTEGER) -> $NIL)
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  $CHARACTER)
 -> (*CONS $CHARACTER
           (*REC !REC8 (*OR $NIL (*CONS $CHARACTER (*RECUR !REC8))))))
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  $INTEGER)
 -> (*CONS $INTEGER
           (*REC !REC9 (*OR $NIL (*CONS $INTEGER (*RECUR !REC9))))))
```

For (BIM X Y), the tool generates three 3-tuples, whose minimal segments are:

```
(($NIL &2) -> $NIL)
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &2)
 -> &2)
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &2)
 -> &2)
```

The latter segment is repeated because it occurs in conjunction with two different CONC-ALISTS and two different maximal segments. The corresponding maximal segments are:

```
(($NIL (*OR $CHARACTER $INTEGER)) -> $NIL))
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  $CHARACTER)
 -> $CHARACTER)
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  $INTEGER)
 -> $INTEGER)
```

First we perform the guard verification step. We combine, using *OR, all the result types from the maximal segments for (BAR X Y) to give a conservative type for the first argment, and likewise for (BIM X Y) and the second argument. Thus, the type constructed for the first argument is:

```
(*OR $NIL
     (*CONS $CHARACTER
            (*REC !REC8 (*OR $NIL (*CONS $CHARACTER (*RECUR !REC8)))))
     (*CONS $INTEGER
            (*REC !REC9 (*OR $NIL (*CONS $INTEGER (*RECUR !REC9))))))
```

and for the second argument:

```
(*OR $CHARACTER $INTEGER $NIL)
```

From the signature for FOO, we have the guard descriptors:

```
((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
 (*OR $CHARACTER $INTEGER $NIL))
```

The containment algorithm determines that the descriptors for the two arguments are contained in the guard descriptor for FOO, so we have passed the guard verification test.

Next we use TC-MAKE-ARG-CROSS-PRODUCT to find all possible combinations of types for the first and second arguments, so they can be used to match against the segments for FOO. TC-MAKE-ARG-CROSS-PRODUCT takes the minimal segments and CONC-ALISTs for the arguments and forms the cross product of all their possible combinations, and then uses unification to merge each combination into a single schema. The form of each such schema is a list of three lists. The first list gives descriptors for the variables in the environment; the second list gives descriptors for each of the arguments to the function call; and the third list is the right hand sides for the combined CONC-ALIST. (In our example, the CONC-ALIST component will not be very interesting. Since all the incoming CONC-ALISTs are identical, the resulting CONC-ALISTs will also be identical with the originals. But this is not always the case; IF forms in the arguments could result in different CONC-ALISTs on the input segments.) The result of the cross product operation in our example is:

```
((($NIL $CHARACTER)             ; types of X and Y
  ($NIL $NIL)                   ; types of (BAR X Y) and (BIM X Y)
  ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
   (*OR $CHARACTER $INTEGER)))  ; CONC-ALIST for X and Y
 (($NIL $INTEGER)               ; types of X and Y
  ($NIL $NIL)                   ; types of (BAR X Y) and (BIM X Y)
  ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
   (*OR $CHARACTER $INTEGER)))
 (((*CONS *UNIVERSAL
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $CHARACTER)                  ; types of X and Y
  ((*CONS $CHARACTER            ; types of (BAR X Y) and (BIM X Y)
          (*REC !REC8 (*OR $NIL (*CONS $CHARACTER (*RECUR !REC8)))))
   $CHARACTER)
  ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
   (*OR $CHARACTER $INTEGER)))
 (((*CONS *UNIVERSAL
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $INTEGER)                    ; types of X and Y
  ((*CONS $INTEGER
          (*REC !REC9 (*OR $NIL (*CONS $INTEGER (*RECUR !REC9)))))
   $INTEGER)                    ; types of (BAR X Y) and (BIM X Y)
  ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
   (*OR $CHARACTER $INTEGER))))
```

Now recall the segments for FOO:

```
 (((*OR $NIL
    (*CONS (*OR $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING
                $T (*CONS *UNIVERSAL *UNIVERSAL))
           (*REC TRUE-LISTP
                 (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))))
   (*OR $CHARACTER $INTEGER $NIL))
   -> $NIL)
 (((*CONS $INTEGER
```

```
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $CHARACTER)
   -> (*CONS $INTEGER $CHARACTER))
 (((*CONS $INTEGER
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $INTEGER)
   -> (*CONS $INTEGER $INTEGER))
 (((*CONS $INTEGER
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   $NIL)
   -> (*CONS $INTEGER $NIL))
 (((*CONS $CHARACTER
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (*OR $CHARACTER $INTEGER $NIL))
   -> (*CONS $CHARACTER $NIL))
```

TC-INFER unifies each cross product result with each segment for FOO in a grand unification which involves the type of each variable in the environment, the type of each parameter, and the type of the result of the call to FOO. Since there is no descriptor corresponding to TC-INFER-EXAMPLE's variable types in the segments for FOO, and since there is no descriptor for the result of FOO in the cross product, both schema are padded accordingly with *UNIVERSAL. Let us consider the unification performed for the first cross product element and the first segment of FOO. The cross product element is:

```
 (($NIL $CHARACTER)             ; types of X and Y
  ($NIL $NIL)                   ; types of (BAR X Y) and (BIM X Y)
  ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
   (*OR $CHARACTER $INTEGER)))
```

and the segment is:

```
 (((*OR $NIL
    (*CONS (*OR $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING $T
                (*CONS *UNIVERSAL *UNIVERSAL))
           (*REC TRUE-LISTP
                 (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))))
   (*OR $CHARACTER $INTEGER $NIL))
   -> $NIL)
```

The first two cross product element components are combined into a single list and extended with *UNIVERSAL, the latter representing the result of the call to FOO, giving the descriptor list:

```
(*DLIST $NIL $CHARACTER $NIL $NIL *UNIVERSAL)
```

FOO's segment is padded with *UNIVERSAL for each variable in TC-INFER-EXAMPLE's environment, giving the descriptor list:

```
(*DLIST *UNIVERSAL
        *UNIVERSAL
        (*OR $NIL
             (*CONS (*OR $NIL $NON-INTEGER-RATIONAL
                         $NON-T-NIL-SYMBOL $STRING $T
                         (*CONS *UNIVERSAL *UNIVERSAL))
                    (*REC TRUE-LISTP
                          (*OR $NIL (*CONS *UNIVERSAL
                                           (*RECUR TRUE-LISTP))))))
        (*OR $CHARACTER $INTEGER $NIL)
        $NIL)
```

These two descriptor lists are then unified, using DUNIFY-DESCRIPTORS-INTERFACE, giving the result:

```
(*DLIST $NIL $CHARACTER $NIL $NIL $NIL)
```

Since the immediate goal is to form a segment mapping the types of X and Y to the type of the result of the function call, we are no longer interested in the types of the function parameters in this result, so we ignore them. The minimal segment formed from the remnant is:

```
(($NIL $CHARACTER) -> $NIL)
```

Note that had the result of the unification been *EMPTY, this unification would have resulted in zero segments. Had the result been an *OR of *DLISTs, the result would have been one minimal segment for each disjunct.

The final step is to form the maximal segment by unifying the minimal segment with a descriptor list composed of the descriptors from the CONC-ALIST provided with this cross product element, padded with an extra element for the result type. I.e., we unify the descriptor from the minimal segment:

```
(*DLIST $NIL $CHARACTER $NIL)
```

with the extended CONC-ALIST:

```
(*DLIST (*REC TRUE-LISTP
              (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
        (*OR $CHARACTER $INTEGER)
        *UNIVERSAL)
```

The result is:

```
(*DLIST $NIL $CHARACTER $NIL)
```

So the maximal segment is identical with the minimal segment. The CONC-ALIST completing the 3-tuple is the one from the cross product element.

This process is repeated for each combination of cross product elements and segments from FOO, and all the results collected into a list of 3-tuples. The collection of 3-tuples for (FOO (BAR X Y) (BIM X Y)) is:

```
(((($NIL $CHARACTER) -> $NIL)           ; minimal segment
  ((X . (*REC TRUE-LISTP                ; CONC-ALIST
              (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (Y . (*OR $CHARACTER $INTEGER)))
  (($NIL $CHARACTER) -> $NIL))          ; maximal segment
 ((($NIL $INTEGER) -> $NIL)             ; minimal segment
  ((X . (*REC TRUE-LISTP                ; CONC-ALIST
              (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (Y . (*OR $CHARACTER $INTEGER)))
  (($NIL $INTEGER) $NIL))               ; maximal segment
 ((((*CONS *UNIVERSAL                   ; minimal segment
           (*REC TRUE-LISTP
                 (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
    $INTEGER)
   -> (*CONS $INTEGER $INTEGER))
  ((X . (*REC TRUE-LISTP                ; CONC-ALIST
              (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (Y . (*OR $CHARACTER $INTEGER)))
  (((*CONS *UNIVERSAL                   ; maximal segment
           (*REC TRUE-LISTP
                 (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
```

```
    -> $INTEGER)
   (*CONS $INTEGER $INTEGER)))
 ((((*CONS *UNIVERSAL                   ; minimal segment
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
    $CHARACTER)
   -> (*CONS $CHARACTER $NIL))
  ((X . (*REC TRUE-LISTP                ; CONC-ALIST
             (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (Y . (*OR $CHARACTER $INTEGER)))
  (((*CONS *UNIVERSAL                   ; maximal segment
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
    $CHARACTER)
   -> (*CONS $CHARACTER $NIL))))
```

Since this is the outermost form of the body, the maximal segments from this collection of tuples will become the segments for function TC-INFER-EXAMPLE.

6.6 The Unification Algorithm

In this system, as in many type inference systems, unification is a central algorithm. [Knight 89] The unification of descriptors is perhaps the central algorithm of both the inference algorithm and the signature checker. As we can see from the preceding discussion of the checker, it is at the heart of the process of deriving segments for function call results. Moreover, the entire function signature scheme, though reminiscent of the traditional notation of function signature in programming (modified for OR semantics, see the discussion in Section 5.7.1) is oriented toward applying the unifier to both the guard vector and the segments. Thus in large measure, to understand the inference algorithm and the type checker, one needs to understand at least the role which the unifier plays.

6.6.1 The Nature of Descriptor Unification

Informally, the specification of descriptor unification is as follows. Given two descriptors (or two lists of descriptors of the same arity), the unifier produces a descriptor (or list of descriptors of the same arity as the parameters) such that, under any given binding of type variables, any value (or list of values) which satisfies both the input descriptors under our INTERP-SIMPLE or INTERP-SIMPLE-1 function will also satisfy the result descriptors. In one sense, this specification encourages the notion that unification can produce descriptors which are larger than necessary, in order to satisfy this soundness constraint. After all, the unifier could always return *UNIVERSAL and satisfy this specification. But this is just our soundness requirement, and our usability requirement is that the unifier should produce the most specific descriptor possible which satisfies this constraint. In this sense, unification is essentially a narrowing operation, and one may think of unification in the spirit of an intersection operation, where we want the result to signify *only* values common to both input descriptors. Though this usability requirement is not formalized, if we relaxed our treatment of it in the implementation, we would surely generate uselessly general signatures.

6.6.2 The Formal Specification of the Unifier

DUNIFY-DESCRIPTORS-INTERFACE is the top level function of the unifier. It takes two formal parameters, the descriptors to be unified, and returns a descriptor representing the unified result. In the case where the arguments are single type descriptors, DUNIFY-DESCRIPTORS-INTERFACE returns a single type descriptor. Alternatively, it can take as arguments two *DLIST forms, each of the form (*DLIST td_1 .. td_n), where each td_i is a single type descriptor, and return either a *DLIST of length n or an *OR of *DLIST forms, each of length n. A *DLIST is simply a notation for packaging a list of simple descriptors into a single argument, serving like a mutual recursion flag so the function can take either single descriptors or lists of descriptors. When an *OR of *DLIST forms is returned, it signifies that any of the *DLISTS is a correct unification of the arguments. That it is packaged as an *OR is perhaps a case of poor naming, since this *OR has no formal relation to the normal *OR descriptor, but indicates to the caller that the result should be unpacked and considered a list of possible answers. In fact, *OR was used because within DUNIFY-DESCRIPTORS-INTERFACE is a call to DUNIFY-DESCRIPTORS, which can return a list of unified forms. These are combined with *OR, which is perfectly natural in the single descriptor case. When *DLIST forms are used as arguments, the values used in the specification Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK must be considered lists of values of the same length as the *DLISTs, and the calls of "I" are calls of INTERP-SIMPLE rather than INTERP-SIMPLE-1.

The formal specification of DUNIFY-DESCRIPTORS-INTERFACE is a straightforward restatement of the informal specification given above.

```
Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK
 For any descriptors tda and tdb, Lisp value v and fully
 instantiating binding of type variables to Lisp values b,

 (and (I tda v b)
      (I tdb v b))
 =>
 (I (dunify-descriptors-interface tda tdb) v b)
```

B is a binding of type variables to singleton values. By "fully instantiating", we mean simply that all the variables appearing in the descriptor in question are bound in B.

DUNIFY-DESCRIPTORS-INTERFACE is just a wrapper for the central recursive function of the algorithm, DUNIFY-DESCRIPTORS, which we will discuss briefly and then tie back into DUNIFY-DESCRIPTORS-INTERFACE. DUNIFY-DESCRIPTORS takes as parameters the two descriptors (or *DLISTs), a list of assumptions about the type variables in the form of a mapping SUBSTS from type variables to type descriptors, and a structure TERM-RECS which serves to restrain the unifier from descending into infinite recursion when dealing with *REC descriptors. It returns a list of pairs, where each pair consists of a descriptor and a substitution list. The correctness of the descriptor portion of this pair is given in terms of INTERP-SIMPLE-1 and INTERP-SIMPLE, as usual. The correctness assumption for the substitution list is expressed in terms of a function INTERP-SUBSTS, which interprets a substitution list in conjunction with a binding of type variables to values.

```
(DEFUN INTERP-SUBSTS (SUBSTS BINDINGS)
  ;; SUBSTS should be an alist whose keys are type variables and
  ;;  whose associated values are well-formed descriptors.
  ;; BINDINGS should be an alist associating type variables to
  ;;  well-formed descriptors.
  (IF (NULL SUBSTS)
      T
      (AND (INTERP-SIMPLE-1 (CDR (CAR SUBSTS))
                            (CDR (ASSOC (CAR (CAR SUBSTS)) BINDINGS))
                            BINDINGS)
           (INTERP-SUBSTS (CDR SUBSTS) BINDINGS))))
```

This function checks that under a given binding B, each substitution maps a variable to a descriptor which accurately characterizes the value to which the same variable is bound in B. I.e., for each substitution (&i . td_i), under a binding including (&i . v), (INTERP-SIMPLE-1 td_i v b) = T. The substitution list is used to accumulate information about variables already encountered on recursive descent into the problem.

A lemma which will be most critical to the proof of soundness for the unifier is the specification of DUNIFY-DESCRIPTORS, below. It omits mention of TERM-RECS, which has no bearing on soundness. That is, one can insert TERM-RECS in the call to DUNIFY-DESCRIPTORS, viewing it as universally quantified.

```
Lemma DUNIFY-DESCRIPTORS-OK
 Denoting (dunify-descriptors tda tdb substs)
  by ((td1 . substs1) .. (tdn . substsn)),
 for all v and fully instantiating b,

  (and
H1  (interp-substs substs b)
H2  (I tda v b)
H3  (I tdb v b))
  =>
  for some i,
    (and (interp-substs substsi b)
         (I tdi v b))
```

Paraphrased, this says that under any binding, if a value satisfies both input descriptors under the binding, and if the substitution is valid under the binding, then there exists some descriptor-subst pair in the result such that the subst is valid, and the value satisfies the descriptor.

6.6.3 Detailed Description of the Unifier

The bridge between DUNIFY-DESCRIPTORS-INTERFACE and DUNIFY-DESCRIPTORS is short. DUNIFY-DESCRIPTORS-INTERFACE calls DUNIFY-DESCRIPTORS with the input descriptors, a NIL substitution list, and a NIL TERM-RECS structure. It obtains the result, which is a list of descriptor-substs pairs, and for each pair applies the substs to the descriptors as a straightforward substitution. It continues applying this substitution until the result stabilizes. Then it combines all the resulting descriptors using *OR and canonicalizes.

In some part, DUNIFY-DESCRIPTORS can be easily understood as an intersection algorithm, laying consideration of variables aside and thinking about various combinations of descriptors as separate cases. But its greatest intricacy occurs where variables are concerned, and after giving a flavor for some of the other combinations, this is where we will focus the discussion.

The unification rules are as follows. They should be considered in order, as two descriptors may match more than one action. Thus, the triggering condition for any rule may be read to include the negation of previous triggering conditions.

1. If DESCRIPTOR1 equals DESCRIPTOR2, then the result is the singleton list containing (DESCRIPTOR1 . SUBSTS). (This corresponds to Case 1 in the proof of Lemma DUNIFY-DESCRIPTORS-OK.)
2. If either descriptor is *EMPTY, the result is NIL. (Cases 2 and 2')
3. If either descriptor is *UNIVERSAL, the result is the singleton pairing the other descriptor with SUBSTS. (Cases 3 and 3')
4. If DESCRIPTOR1 is of the form (*OR td_1 .. td_n), then if DESCRIPTOR2 is equal to some td_i, the result is the list containing only (DESCRIPTOR2 . SUBSTS), otherwise we append the lists from the results of unifying DESCRIPTOR2 with each td_i under the same SUBSTS and TERM-RECS. (Case 4)
5. If DESCRIPTOR2 is of the form (*OR td_1 .. td_n), then proceed symmetrically as above. (Case 4')

6. We will describe the case where either of the descriptors is a variable below. (Case 5)
7. If both DESCRIPTOR1 and DESCRIPTOR2 are *REC forms, we resort to a bag of tricks, also described below. (Case 6.1)
8. If DESCRIPTOR1 is a *REC form, then if the pair (DESCRIPTOR1 . DESCRIPTOR2) is a member of TERM-RECS, we return a list containing only (DESCRIPTOR2 . SUBSTS). Otherwise, we recur, with DESCRIPTOR1 opened up, and the pair (DESCRIPTOR1 . DESCRIPTOR2) pushed onto TERM-RECS. It would be possible, with *REC forms in both descriptors, for this algorithm to go into an infinite recursion, first unfolding one *REC and then the other, or by some similar but more elaborate scenario. But DUNIFY-DESCRIPTORS maintains a stack of pairs of operands, the TERM-RECS parameter, so that when it is recursively called with a pair which it has already seen, it can recognize that it is in an endless recursive chain. Any infinite unfolding in this manner must come back around to considering the original case, buried beneath a number of recursive calls. Almost always, this indicates that no value can satisfy both descriptors, and hence, almost always it would be sound to return *EMPTY rather than DESCRIPTOR2. An example of this would be:

```
(DUNIFY-DESCRIPTORS
  '(*REC DOUBLE
         (*OR $NIL (*CONS *UNIVERSAL
                          (*CONS *UNIVERSAL
                                 (*RECUR DOUBLE)))))
  '(*CONS *UNIVERSAL
          (*REC DOUBLE
                (*OR $NIL (*CONS *UNIVERSAL
                                 (*CONS *UNIVERSAL
                                        (*RECUR DOUBLE))))))
  SUBSTS TERM-RECS)
```

Here, proceeding without a stopper would result in first opening DESCRIPTOR1 and diving into the *CONS, then opening DESCRIPTOR2 and diving into the *CONS, *ad infinitum*. And there is no value which satisfies both these descriptors, since to do so would require a list whose length is both even and odd. Unfortunately, it is also possible that some value could satisfy both descriptors, and our algorithm simply is not strong enough to formulate the unifying descriptor. For this reason, we have to be conservative, so we return DESCRIPTOR2. We could return either DESCRIPTOR1 or DESCRIPTOR2 if we could figure out which were better in some sense, but in the current implementation we do not attempt to do so. (Case 6.2)

9. If DESCRIPTOR2 is a *REC form, proceed symmetrically as above. (Case 6.2')
10. If both DESCRIPTOR1 and DESCRIPTOR2 are *CONS descriptors, we do exactly as with *DLISTS below (since this is essentially a special case of length two), except that at the end we glue the *CONS operator back onto the descriptors instead of *DLIST. (Case 7)
11. If both DESCRIPTOR1 and DESCRIPTOR2 are *DLIST descriptors, first we make sure the lists are of the same length (otherwise indicates an error in the implementation, as does unifying a *DLIST with a single descriptor). Then we engage in a recursive algorithm for unifying lists of descriptors. First we unify the first descriptor in DESCRIPTOR1 with the first descriptor in DESCRIPTOR2. If this is the end of the list, we return this result (with the result descriptors wrapped in a list). Otherwise, we append the results of recurring, for each descriptor-subst pair returned for the CARs, on the CDRS. On each recursive call, we use the substs from the pair, which is a composition of the original SUBSTS and additional substitutions accumulated from the CAR. Each recursive call will return a list of pairs, and from this list we make another list, where we form the pairs by CONS-ing our CAR descriptor onto the descriptor list in each result pair, and using the substitutions from the result pair without modification. When we emerge from this recursive traversal, we stick the *DLIST operator back onto each complete list, so that the "descriptor" in each of our result pairs is a *DLIST of the same arity as the original. (Case 8)

12. Otherwise return NIL, since we have eliminated all possibilities except that one descriptor is a *CONS and the other is not, or that both descriptors are atomic simple descriptors like $INTEGER but are not equal. Thus, there can be no commonality in the values they represent. (Case 9)

So what do we do with variables? A basic notion is that when we unify a variable with something, the result is a list of descriptor-substitution pairs where the variable is the descriptor and the substitution is the input SUBSTS augmented with a new element which maps the variable to the thing with which the variable is being unified. But adding this new element can be a complex operation. For instance, if the variable is already mapped in SUBSTS, then the two mappings need to be reconciled. This is done by unifying the two right hand sides, since unification is the operation which forms a descriptor for the common ground between two descriptors. So with the complexity thus unfolding, let us look in detail at how a new element is added to the SUBST list.

The top level function in this operation is DMERGE-NEW-SUBST. It takes a VARIABLE, a DESCRIPTOR to which the variable will be mapped, and a substitution SUBSTS. DMERGE-NEW-SUBST returns a list of substitutions. (Keep in mind that a substitution is itself a list of substitution elements, each of which maps a variable to a descriptor. SUBSTS refers to a substitution, rather than a list of substitution.) It returns a list of substitutions rather than a single one because, if a unification is necessary to perform the merge, the result of the unification will be a list of pairs, each representing a different possible unifier, and each resulting in a different substitution.

The merge is performed as follows. If there is no current element mapping VARIABLE in SUBSTS, we simply insert (VARIABLE . DESCRIPTOR) as a new element. The SUBSTS are maintained as an ordered list, according to the lexical ordering of the bound variables. As discussed below, this ordering is one of several tricks that are used to help avoid potential problems with infinite loops. So inserting a new element means inserting it at the proper position in the list. We then return the singleton list containing this augmented SUBST.

If there is already an element mapping VARIABLE in SUBSTS, we get the right hand side and unify it with our DESCRIPTOR, using the original SUBSTS as the parameter on the recursive call to DUNIFY-DESCRIPTORS. We take each result from the unifier and create new substitution lists as follows, collecting all the resulting substitution lists into a great list of substitutions. First, we compare the substitution returned from the unifier with the substitution we started with, to see if the right hand sides for any variables other than the one we are concerned with are different (or not present) in the two substitutions. If not, then we just include in our result the original substitution with the right hand side for our variable replaced with the descriptor from the unifier result which we are considering.

If there are any other different right hand sides, however, we engage in a second level merging operation. This second level operation takes a list of substitution elements and a list of substitutions and returns a list of substitutions. The idea is that we return all substitutions obtainable by "merging" the elements into any of the given substitutions. The initial list of substitution elements contains all the elements in the substitution (returned by the unifier) being considered which are new or different relative to the original SUBSTS. The initial list of substitutions contains the single original substitution SUBSTS. We proceed as follows, in the order determined by the lexical ordering of the variables. We take the first substitution element in our first argument and use a recursive call of DMERGE-NEW-SUBST to merge the element into each substitution in the second argument. The result from each such merge is, of course, a list of substitutions, and all these lists are appended to form the results of merging the first new element. Then, we merge the next new element into all these substitution lists, and so forth, until each variable with a new right hand side has been merged.

After this second level merging finally returns a list of substitutions, we return to the variable whose unification began this whole process, and we replace its old right hand side in each of the substitution lists from the second level merge with the right hand side proposed from the original unification. This final step is the same as the final step for the case, described two paragraphs above, where no second level merge was required.

A well-formedness property must hold for substitutions.

Definition: A substitution is a *well-formed substitution*
if all its variable-to-variable mappings are downward directed according to a lexical ordering on variable names, and if the substitution contains no circularities, where by circularity we mean a circular path, formed by the transitive closure of a relation whereby the variable on the left hand side of a substitution element is linked to each of the variables on the right hand side.

When, at the very top of the unification algorithm in DUNIFY-DESCRIPTORS-INTERFACE, we use DAPPLY-SUBST-LIST to apply each substitution list to the descriptor with which it is paired, this application is repeated until the resulting descriptor stabilizes. This, in effect, unwinds the chain of references implicit in the SUBSTS. If there are circularities in the SUBSTS, this application will never stabilize, and the algorithm cannot terminate. A circularity exists if, for some variable in the substitution list, we can take the descriptor to which it is mapped and follow some chain of elements for the variables in the descriptor recursively back to the original variable.

The algorithm uses some safeguards against construction of ill-formed substitutions. For instance, the case analysis in DUNIFY-DESCRIPTORS ensures that circularities that consist only of substitution elements which map variables to variables are never constructed. Essentially, this is accomplished by mapping variables directly to other variables only in a lexically downward manner, and by never allowing an *OR as the top level form on the right hand side in a substitution element. (A variable could appear as one of the disjuncts, thus creating a potential loop if it is lexically greater than the variable on the left hand side.) Also, DUNIFY-DESCRIPTORS ensures that no substitution element can be created where the right hand side is a *REC descriptor which, if opened up, could be an *OR with a variable as a top-level disjunct.

But in the merging operation, it may still be possible to create a substitution in which a circularity exists where some variable in the loop is embedded within a *CONS. As originally constructed, the algorithm removed any such substitution from consideration, under the presumption that it represented an unrealizable situation where some Lisp CONS value would be required to be equal to some component of itself. For example,

```
((&1 . (*CONS &2 $NIL)) (&2 . &1))
```

The soundness assumption for any substitution which is integral to the proof of DUNIFY-DESCRIPTORS-OK is that there may exist some binding of its type variables to Lisp values such that no inconsistency arises. In this case, let us generically say that this binding is

```
((&1 . val1) (&2 . val2))
```

Thus, we would require

```
(AND (CONSP VAL1)                ; by the first substitution element
     (EQUAL (CAR VAL1) VAL2)
     (EQUAL (CDR VAL1) NIL)
     (EQUAL VAL1 VAL2)           ; by the second element
```

Clearly, there is a contradiction here, since VAL1 cannot be equal to its own CAR. But consider the substitution

```
((&1 . (*CONS (*OR $T &2) $NIL)) (&2 . &1))
```

This substitution contains a loop, but is perfectly consistent with the binding

```
((&1 . (CONS T NIL)) (&2 . (CONS T NIL)))
```

Therefore, we cannot remove a substitution from consideration simply because it contains a loop, since doing so may amount to removing the sole solution satisfying the conclusion of DUNIFY-DESCRIPTORS-OK.

One possible way out of this quandary would be to analyze the substitution to determine how it could be soundly pruned to remove the loop. The previous example could be soundly transformed to

```
((&1 . (*CONS $T $NIL)) (&2 . &1))
```

since the &2 disjunct in the substitution element for &1 did not contribute to the soundness argument with respect to the good binding. But in general, it may be difficult to determine just where to perform this kind of surgical pruning.

But it so happens that despite the measures taken, the proof we give for DUNIFY-DESCRIPTORS-OK is a partial correctness proof, in that termination is not guaranteed. So, rather than complicate both the algorithm and its proof of correctness with a difficult pruning operation, we chose to simply allow DUNIFY-DESCRIPTORS to return a looping substitution. Then, when DUNIFY-DESCRIPTORS-INTERFACE prepares to apply the substitution to construct its final result, it first checks that it is well-formed. If there are any circularities, the algorithm announces that an ill-formed substitution was constructed and punts by returning the descriptor argument whose form contains the fewer CONSes. Either argument descriptor could be returned, and any heuristic for returning one or the other would be trivially sound. We chose this heuristic because of its simplicity. The function which tests for well-formedness is WELL-FORMED-SUBSTS, and its definition is in Appendix G.8. It checks both the downward-directedness of all variable-to-variable substitution elements and the absence of circularity.

The saving grace is that, while we can construct, by hand, calls to the unifier which will result in the construction of looping substitutions, we have failed, despite some thought and effort, to construct an inference problem which led to such a call. So there is no apparent loss of functionality in this non-implemention of a termination safeguard.

Now let us pop back to the two outstanding original cases: unifying a variable with another variable, and unifying a variable with a non-variable descriptor. When we unify a variable with another variable, we first determine which is lexically greater. We merge the binding of the larger variable to the smaller one into SUBSTS, and for each resulting substitution, we form a descriptor-substitution pair where the descriptor is the smaller variable. So:

```
(dunify-descriptors '&1 '&2 nil nil)
 =
((&1 . ((&2 . &1))))
```

where the result is a single pair whose descriptor component is &1 and whose substitution list is ((&2 . &1)). Also,

```
(dunify-descriptors '&3 '&2 '((&2 . (*cons &1 $nil))) nil)
 =
```

```
((&2 . ((&2 . (*CONS &1 $NIL)) (&3 . &2))))
```

and

```
(dunify-descriptors '&3 '&2 '((&2 . &1)) nil)
 =
((&2 . ((&2 . &1) (&3 . &2))))
```

If we unify a variable with a descriptor other than a variable, we first check several special cases. For the sake of exposition, let us denote DESCRIPTOR1 by "&i". We have already considered the possibility that DESCRIPTOR2 is an *OR descriptor, and in the case where DESCRIPTOR1 is one of its disjuncts, we simply returned DESCRIPTOR1 paired with our original SUBSTS. But we have not yet considered the possibility that DESCRIPTOR2 is a *REC form. If it is, and the body of the *REC descriptor is an *OR form, and if &i is one of the possible disjuncts of that *OR, then we treat it as we would have with the simple *OR, returning the single pair ((&i . SUBSTS)).

If DESCRIPTOR2 is some other descriptor in which &i appears, we do a trick which may seem arcane, but which is easily justified and may result in more accurate results. We know from previous tests that DESCRIPTOR2 in no way represents an *OR form where &i is one of its disjuncts. We also know that DESCRIPTOR2 is not equal to &i. Therefore, if &i appears in DESCRIPTOR2, it must be because it appears within some *CONS structure. Since variables represent specific Lisp values, there is no way &i can unify with, or represent the same value as, a descriptor with &i buried inside a *CONS. This is not to say our unification will fail. A valid result of unifying &i with

```
(*CONS (*OR &i $INTEGER) $NIL)
```

is

```
((&i . ((&i . (*CONS $INTEGER $NIL)))))
```

So what we do is "screen" DESCRIPTOR2 for occurrences of &i, eliminating them where they appear as possible disjuncts in any *OR configuration. We can then return &i as our descriptor, paired with each substitution formed by merging the mapping of &i to the screened descriptor with the original SUBSTS. Of course, if the screened descriptor is effectively *EMPTY, we return an empty list. The code implementing this case is presented in Appendix G.10.

If &i does not appear in DESCRIPTOR2, we first consider the case where DESCRIPTOR2 is a *REC which has some variable as one of its disjuncts, for example where DESCRIPTOR2 is of the form

```
(*REC FOO (*OR $NIL &j (*CONS *UNIVERSAL (*RECUR FOO))))
```

If this is the case, we simply open the *REC using OPEN-REC-DESCRIPTOR-ABSOLUTE recursively unify it with &i. We open the *REC because we do not want to allow a substitution of the form:

```
(&i . (*REC FOO (*OR $NIL &j (*CONS *UNIVERSAL (*RECUR FOO)))))
```

This is because, in our notion of substitutions, we have meticulously avoided any substitution mapping a variable to an *OR, as this could profoundly complicate the issue of termination by allowing for the creation of ill-formed, but realizable substitutions. (See discussion above.) A substitution element like that above is a mapping to an *OR in thin disguise, and is likewise to be avoided. We accomplish this by exposing the *OR as the top level form (i.e., by opening the *REC), so that on our recursive call we unify with the *OR. As described in the original case analysis, this results in the *OR being split, and we wind up creating distinct substitutions within which we map the variable to each disjunct of the *OR.

This strategy itself could cause a non-termination problem resulting from the infinite unfolding of *REC descriptors. (Perhaps our &i originated from a similar position to that of &j in some other *REC descriptor.) This situation could only arise either from the unification of two *REC descriptors or from the unification of a *REC descriptor with another descriptor containing a *REC. In the first case, our special case *REC unification rules (described below) are such that no direct unification of *RECs can lead to this scenario. And the TERM-RECS mechanism guards against infinite recursion arising from the unification of a *REC with a non-*REC. In the absence of a termination proof, of course, we have to take these claims on faith.

We identified the previous special case to protect against it falling into our last course of action, which is simply to merge the substitution element (&i . DESCRIPTOR2) into SUBSTS, and to pair the descriptor &i with each resulting substitution. This, in fact, is the normal case.

That completes the discussion of unification where one of the descriptors is a variable.

The only case which has not yet been explained is that of unifying two *REC descriptors. As stated previously, there is no general algorithm for this case in the checker; it is handled by resorting to a bag of tricks. The tricks are a collection of rules which are triggered essentially by pattern matching against the arguments. A default rule handles all cases for which no other rule is eligible. Although a general procedure like the one employed in the inference algorithm would have been attractive and powerful, we were unable to imagine its proof against the semantic model. By using a rule-based approach, we reduce the proof of this case to a manageable proof of soundness for each of our collection of special-case rules.[30]

As an aid to pattern matching, we transform the descriptors to a slightly different canonical form. In the case where the body of the *REC is an *OR (the normal case), we gather all the non-recurring cases into a single disjunct. So for example,

```
(*REC FOO (*OR $INTEGER $NIL &1 (*CONS *UNIVERSAL (*RECUR FOO))
               (*CONS (*RECUR FOO) *UNIVERSAL)))
```

is transformed to:

```
(*REC FOO (*OR (*OR $INTEGER $NIL &1)
               (*CONS *UNIVERSAL (*RECUR FOO))
               (*CONS (*RECUR FOO) *UNIVERSAL)))
```

If the top level form is a *CONS, we similarly re-canonicalize both its CAR and CDR components. After application of the rules, any resulting descriptors are re-canonicalized to the more conventional original form.

Each rule has two parts, an enabling condition and an action. The enabling condition is basically a pattern match against the arguments. The action typically reduces the unification problem for the *RECs to some embedding of recursive calls to the unifier on selected components of the *RECs. In some cases the action is simply to note failure by returning NIL. In the default rule, employed when no other enabling condition is satisfied, the action is to return the list containing a single pair consisting of the argument with the lesser number of atoms in its form and the input substs. This is a conservative result, but after exhausting all the possibilities embodied in the rules, the most precise characterization of the unifying descriptor we could manage would be either DESCRIPTOR1 or DESCRIPTOR2, and we choose the

---

[30] In the implementation, the top level function for *REC unification is DUNIFY-RECS.

lexically smaller one arbitrarily.

The rules employed are as follows. The notation is in a generic pidgin descriptor dialect, meant for brevity rather than precision. Each rule is stated as an equality, meaning that the unification represented on the left hand side results in the answer on the right hand side. Names like "foo", "bar", and "bim" are generic *REC descriptor labels. Names like "d1", "d2", etc. represent arbitrary descriptors, except as qualified in the text. "s" represents the input SUBSTS. The TERM-RECS argument is not mentioned. The pattern embodied in the enabling condition is stated implicitly in the form of the arguments. In all the *REC rules, we assume that the only occurrences of *RECUR with the same label as the top level *REC are those explicitly shown. Rules whose names end with a "’" character are just slight modifications of the rules with the same, but unprimed, names. The rules are considered in the order presented (as opposed to the order of their numbers), hence with any rule we can assume that the enabling condition for all previously stated rules has failed.

On the right hand sides of the rules, a name like "d12" represents the result of a call to (DUNIFY-DESCRIPTORS-INTERFACE d1 d2), where d1 and d2 are variable-free descriptors. If we were to use a call to DUNIFY-DESCRIPTORS instead, the result would be a list of descriptor-substitution pairs where in each pair the substitution is our original substitution. All results with the same substitution list may be combined with *OR, therefore in this case we can return a singleton result, with the pair being the *OR of all the descriptor results. We are lazy. DUNIFY-DESCRIPTORS-INTERFACE performs this disjunction for us, and since the recursive arguments are variable-free, we need not be concerned about the impact of this recursive call on any substitution. So we simply use DUNIFY-DESCRIPTORS-INTERFACE and pair its result with the substitution argument.

```
Rule DUNIFY-*REC12
  Where the two *recs differ only in name, and where bar < foo in
  the lexical ordering DESCRIPTOR-ORDER,
  (dunify-descriptors (*rec foo ( .. (*recur foo) .. ))
                      (*rec bar ( .. (*recur bar) .. ))
                      s)
  = (((*rec bar ( .. (*recur bar) .. )) . s))

 Comment: There is no formal significance to this ordering decision.
  The choice could be arbitrary.  We do it this way because some
  *REC descriptors are named for recognizer functions, and all
  others have names of the form !RECn, where n is some natural
  number.  In the ordering, the !RECn names come higher than
  names beginning with an alphabetic character.  So our choice gives
  preference to named *RECs as opposed to gensymed *RECs, and lower
  numbered gensyms to higher ones.


Rule DUNIFY-*REC1
  (dunify-descriptors
    (*rec foo (*or $nil (*cons d1 (*recur foo))))
    (*rec bar (*or $nil (*cons d2 (*recur bar))))
    s)
   =
  (((*rec bim (*or $nil (*cons d12 (*recur bim)))) . s))


Rule DUNIFY-*REC1’
  (dunify-descriptors
    (*rec foo (*or $nil (*cons (*recur foo) d1)))
    (*rec bar (*or $nil (*cons (*recur bar) d2)))
    s)
   =
  (((*rec bim (*or $nil (*cons (*recur bim) d12))) . s))

 Comment:
  Rule DUNIFY-*REC1’ is identical to DUNIFY-*REC1, except that the
```

```
  order of the cons arguments is reversed.


Rule DUNIFY-*REC2
 Where d1 and d3 have no *CONS and no variables,

 (dunify-descriptors
   (*rec foo (*or d1 (*cons d2 (*recur foo))))
   (*rec bar (*or d3 (*cons d4 (*recur bar))))
   s)
  =
 (((*rec bim (*or d13 (*cons d24 (*recur bim)))) . s))

 Comment:
  d1 and d3 are variable-free by prescription, d2 and d4 by definition,
  since variables cannot appear in the replicating disjuncts of a *REC.


Rule DUNIFY-*REC2'
 Where d1 and d3 have no *CONS and no variables,

 (dunify-descriptors
   (*rec foo (*or d1 (*cons (*recur foo) d2)))
   (*rec bar (*or d3 (*cons (*recur bar) d4)))
   s)
  =
 (((*rec bim (*or d13 (*cons (*recur bim) d24))) . s))

 Comment:
  Rule DUNIFY-*REC2' is the same as Rule DUNIFY-*REC2, except that the
  order of the arguments in the cons is reversed.


Rule DUNIFY-*REC3
 Where d2, d3, and d4 contain no variables and d3 is either a
 primitive descriptor or a disjunction of primitive descriptors,
 and denoting
 (dunify-descriptors &j (*rec bar (*or d3 (*cons d4 (*recur bar)))) s)
   = ((&j . s1) .. (&j . sn)),

 (dunify-descriptors (*rec foo (*or &j (*cons d2 (*recur foo))))
                     (*rec bar (*or d3 (*cons d4 (*recur bar))))
                     s)
  =
 (((*rec bim (*or &j (*cons d24 (*recur bim)))) . s1)
   ..
  ((*rec bim (*or &j (*cons d24 (*recur bim)))) . sn) )


Rule DUNIFY-*REC3'
 Where d2, d3, and d4 contain no variables and d3 contains no *cons
 forms, and adopting the notation that
 (dunify-descriptors &j (*rec bar (*or d3 (*cons (*recur bar) d4))) s)
   = (((&j . s1) .. (&j . sn)),

 (dunify-descriptors (*rec foo (*or &j (*cons (*recur foo) d2)))
                     (*rec bar (*or d3 (*cons (*recur bar) d4)))
                     s)
  =
 (((*rec bim (*or &j (*cons (*recur bim) d24))) . s1)
   ..
  ((*rec bim (*or &j (*cons (*recur bim) d24))) . sn) )

 Comment:
  Rule DUNIFY-*REC3' is the same as Rule DUNIFY-*REC3, except that the
  order of the arguments in the cons is reversed.


Rule DUNIFY-*REC4
 Where neither descriptor contains any variables and d1 and d5 contain
```

```
 no *cons forms, and adopting the notation that
 (dunify-descriptors-interface
    d3 (*rec bar (*or d5 (*cons d6 d7) (*cons d8 (*recur bar)))))
   is denoted by d3bar

 (dunify-descriptors
    (*rec foo (*or d1 (*cons d2 d3) (*cons d4 (*recur foo))))
    (*rec bar (*or d5 (*cons d6 (*recur bar))))
    s)
  =
 (((*rec bim (*or d15 (*cons d26 d3bar) (*cons d46 (*recur bim)))) . s))


Rule DUNIFY-*REC4'
 Where neither descriptor contains any variables and d1 and d5 contain
 no *cons forms,

 (dunify-descriptors-interface
    d3 (*rec bar (*or d5 (*cons d6 d7) (*cons (*recur bar) d8))))
   = d3bar

 (dunify-descriptors
    (*rec foo (*or d1 (*cons d3 d2) (*cons (*recur foo) d4)))
    (*rec bar (*or d5 (*cons (*recur bar) d6)))
    s)
  =
 (((*rec bim (*or d15 (*cons d3bar d26) (*cons (*recur bim) d46))) . s))

 Comment:
  This is Rule DUNIFY-*REC4 with the arguments to the conses reversed.


Rule DUNIFY-*REC5
 With no variables appearing in either descriptor, and denoting
  (dunify-descriptors-interface
     d1 (*cons d4 (*rec bar (*or d3 (*cons d4 (*recur bar))))))
    = d14bar, and
  (dunify-descriptors-interface
     d3 (*cons d2 (*rec foo (*or d1 (*cons d2 (*recur foo))))))
    = d32foo

 (dunify-descriptors
   (*rec foo (*or d1 (*cons d2 (*recur foo))))
   (*rec bar (*or d3 (*cons d4 (*recur bar))))
   s)
  =
 (((*rec bim (*or d13 d14bar d32foo (*cons d24 (*recur bim)))) . s))

 Comment:
  This rule is very general, in that it does not require d1 or d3 to be
  cons-free.  Since all the non-replicating disjuncts in the *rec body
  are gathered into an *or, d1 and d3 represent all such terms, with
  the only other restriction being that they are variable-free.  Since
  there may be other rules which apply to more specific cases and result
  in less complex analysis than the computation of d13, d14bar, d32foo,
  and d24, eligibility for applying this rule should be checked later
  than the ones above.


Rule DUNIFY-*REC5'
With no variables in either descriptor,
 (dunify-descriptors
   (*rec foo (*or d1 (*cons (*recur foo) d2)))
   (*rec bar (*or d3 (*cons (*recur bar) d4)))
   s)
  =
 (((*rec bim (*or d13 d1bar4 d3foo2 (*cons (*recur bim) d24))) . s))

 Comment:
  Rule DUNIFY-*REC5' is the same as Rule DUNIFY-*REC5, except that the
```

```
  order of the arguments in the cons is reversed.

Rule DUNIFY-*REC6
 With no variables anywhere and no conses in d1 or d4,
 (dunify-descriptors
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*recur foo)))))
   (*rec bar (*or d4 (*cons d5 (*recur bar))))
   s)
  =
 (((*rec bim (*or d14 (*cons d25 (*cons d35 (*recur bim))))) . s))

Rule DUNIFY-*REC6' just reverses the order of the arguments.

Rule DUNIFY-*REC7
 No variables anywhere and no conses in d1 or d2,
 (dunify-descriptors
   (*rec foo (*or d1 (*cons (*recur foo) (*recur foo))))
   (*rec bar (*or d2 (*cons (*recur bar) (*recur bar))))
   s)
  =
 (((*rec bim (*or d12 (*cons (*recur bim) (*recur bim)))) . s))

Rule DUNIFY-*REC8
 No variables anywhere and no conses in d1 or d5,
 (dunify-descriptors
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*recur bar))))
   s)
  =
 (((*rec bim (*or d15 (*cons d26 (*cons d36 (*cons d46 (*recur bim))))))
   . s))

Rule DUNIFY-*REC8' just reverses the order of the arguments.

Rule DUNIFY-*REC9
 No variables anywhere and no conses in d1 or d5,
 (dunify-descriptors
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*cons d7 (*recur bar)))))
   s)
  =
 (((*rec bim
    (*or d15
         (*cons d26
                (*cons d37
                       (*cons d46
                              (*cons d27
                                     (*cons d36
                                            (*cons d47
                                                   (*recur bim))))))))))
   . s))

Rule DUNIFY-*REC9' just reverses the order of the arguments.

Rule DUNIFY-*REC13
 No variables anywhere and no conses in d2 or d4
 (dunify-descriptors (*rec foo (*cons d1 (*or d2 (*recur foo))))
                     (*rec bar (*cons d3 (*or d4 (*recur bar))))
                     s)
 =
 (((*rec bim (*cons d13 (*or d24 (*recur bim)))) . s))

Rule DUNIFY-*REC10
 No variables anywhere and no conses in d2 or d3,
 (dunify-descriptors
```

(*rec foo (*cons d1 (*or d2 (*recur foo))))
(*rec bar (*or d3 (*cons d4 (*recur bar))))
s)
=
(((*cons d14 (*rec bim (*or d23 (*cons d14 (*recur bim))))) . s))

Rule DUNIFY-*REC11
Where d2, d3, and d4 contain no variables, and adopting the notation
(dunify-descriptors &j (*rec bar (*or d3 (*cons d4 (*recur bar)))) s)
= ((&j . s_1) .. (&j . s_n))
(dunify-descriptors
d3 (*cons d2 (*rec foo (*or &j (*cons d2 (*recur foo))))) s)
= ((d32foo$_1$. sd3$_1$) .. (d32foo$_m$. sd3$_m$)),

(dunify-descriptors (*rec foo (*or &j (*cons d2 (*recur foo))))
(*rec bar (*or d3 (*cons d4 (*recur bar))))
s)
=
(((*rec bim (*or &j (*cons d24 (*recur bim)))) . s_1)
..
((*rec bim (*or &j (*cons d24 (*recur bim)))) . s_n)
((*rec bam1 (*or d32foo$_1$ (*cons d24 (*recur bam1)))) . sd3$_1$)
..
((*rec bamm (*or d32foo$_m$ (*cons d24 (*recur bamm)))) . sd3$_m$)
)

Rule DUNIFY-*REC11’
Where d2, d3, and d4 contain no variables, and adopting the notation
(dunify-descriptors &j (*rec bar (*or d3 (*cons (*recur bar) d4))) s)
= (((&j . s_1) .. (&j . s_n)) and
(dunify-descriptors
d3 (*cons d2 (*rec foo (*or &j (*cons (*recur foo) d2)))) s)
= (((d32foo$_1$. sd3$_1$) .. (d32foo$_m$. sd3$_m$)

(dunify-descriptors (*rec foo (*or &j (*cons (*recur foo) d2)))
(*rec bar (*or d3 (*cons (*recur bar) d4)))
s)
=
(((*rec bim (*or &j (*cons (*recur bim) d24))) . s_1)
..
((*rec bim (*or &j (*cons (*recur bim) d24))) . s_n)
((*rec bam1 (*or d32foo$_1$ (*cons (*recur bam1) d24))) . sd3$_1$)
..
((*rec bamm (*or d32foo$_m$ (*cons (*recur bamm) d24))) . sd3$_m$)
)

Rule DUNIFY-*REC11’ is the same as Rule DUNIFY-*REC11, except that the order of the arguments in each cons is reversed.

Rule DUNIFY-*REC14
No variables anywhere and no conses in d1, d3, d5, or d7,
(dunify-descriptors
(*rec foo (*or d1 (*cons d2 (*or d3 (*cons d4 (*recur foo))))))
(*rec bar (*or d5 (*cons d6 (*or d7 (*cons d8 (*recur bar))))))
s)
=
(((*rec bim (*or d15 (*cons d26 (*or d37 (*cons d48 (*recur bim))))))
. s))

Comment: Each descriptor characterizes lists which can be of either odd or even length. In foo, the odd elements are d2s, the even elements are d4s, and if the length is even, the tail is a d1, and if the length is odd, the tail is a d3. Similarly for bar. Thus, in bim, the odds are both d2 and d6, the evens are both d4 and d8, if the length is even, the tail is both d1 and d5, and if the

```
 length is even, the tail is both d3 and d7.

Rule DUNIFY-*REC0'
 Where either foo or bar contains some variable,
 (dunify-descriptors foo bar s)
  =
 (dunify-descriptors (replace-all-vars-with-universal foo)
                     (replace-all-vars-with-universal bar)
                     s)

Rule DUNIFY-*REC0
 (dunify-descriptors (*rec foo .. ) (*rec bar .. ) s)
  =
 (if (< (fringe-length (*rec bar .. )) (fringe-length (*rec foo .. )))
     (((*rec bar .. ) . s))
     (((*rec foo .. ) . s)))

 Comment: This is the default rule.  FRINGE-LENGTH just counts the
          number of atoms in the form.  Thus, we return the lexically
          smaller of the two forms.
```

More rules could certainly be added. The rules that we use have been sufficient to support all but a very, very few of the cases which have arisen in testing.

Despite all the steps taken to guarantee termination of the unifier algorithm, the termination argument is very difficult. Among many other factors, the opening of *REC descriptors causes problems. We were never able to find a correct measure with which to prove the unification algorithm terminates, though we sketched one possible approach. For an extended discussion of the argument as we left it, see Appendix F. The measure involved a vector of four distinct factors, each weighted differently and each rather vaguely defined. This effort, if nothing else, convinced us that any correct measure would be of great complexity, and further pursuing such a proof did not seem worthwhile. But even being aware of the structure of this failed effort to find a measure, we were unable to construct an example which failed to terminate, nor did we encounter any such example in any of the testing exercises. We believe the algorithm will always terminate, but in the absence of proof, we must take this on faith. In any case, a proof of termination is not necessary for the rest of this work.

6.7 Descriptor Canonicalization

Keeping the descriptors being manipulated by the checker in a canonical form helps minimize case propagation and clarify the algorithms. So, in many instances after new descriptors have been formulated, they are canonicalized. The canonicalization process is essentially the application of valid transformation rules appropriate for the descriptor being considered.

All the transformation rules are value-preserving, in that the canonicalized descriptor always represents the same set of values as the original, or rather, the same interpretation with respect to a binding under INTERP-SIMPLE. For soundness purposes, it is sufficient for the formal specification for each canonicalization to be of the form:

```
 For any descriptor td, value v, and binding b covering td,
  (I td v b) => (I (pcanonicalize-descriptor td) v b)
```

But in fact for all canonicalizations this relation is bi-directional.

PCANONICALIZE-DESCRIPTOR is the top-level function of the canonicalizer. To encapsulate the formal specifications of all the canonicalization rules, we have the following top-level lemma specifying the correctness of PCANONICALIZE-DESCRIPTOR. As with DUNIFY-DESCRIPTORS-INTERFACE, if the argument is a *DLIST, a *DLIST of the same length is returned, and in this case in the lemma below, the value v is considered to be a list of values the same length as the *DLIST, and the call to "I" denotes a call to INTERP-SIMPLE rather than INTERP-SIMPLE-1.

```
Lemma PCANONICALIZE-DESCRIPTOR-OK

 For any descriptor td, value v, and binding b,
  (where td is a *dlist containing n descriptors iff v is a
  value list of length n)

  (I td v b)
  =>
  (I (pcanonicalize-descriptor td) v b)
```

PCANONICALIZE-DESCRIPTOR directly calls PREAL-PCANONICALIZE-DESCRIPTOR, which performs some minimal management of the canonicalization process. PREAL-PCANONICALIZE-DESCRIPTOR invokes the principal recursive function PCANONICALIZE-DESCRIPTOR-1 and checks to see if the result differs from the original descriptor. If so, it calls PCANONICALIZE-DESCRIPTOR-1, after first attempting to fold up any expanded *REC descriptors and consolidate any expanded representations of *UNIVERSAL if possible. (The two transformations just described are in fact applications of canonicalization Rules 1 and 3 below.) When canonicalization stabilizes, it returns the result.

Below this level, the canonicalization code simply applies the transformation rules as appropriate. Each rule is bi-directional, in that the input and output descriptors are equivalent. The rules are stated below, with the direction of canonicalization being from left hand side to right hand side. Along with each is the name of the principal Lisp function within which it is implemented. Comments are supplied occasionally for clarity.

```
Rule 1:
  (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
       $NON-T-NIL-SYMBOL $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
  => *UNIVERSAL
From function PCONSOLIDATE-UNIVERSAL-DESCRIPTORS

Rule 2:
  (*REC FOO (... (*RECUR FOO) ...))
  => (... (*REC FOO (... (*RECUR FOO) ...)) ...)
From function OPEN-REC-DESCRIPTOR-ABSOLUTE

For example:
 (OPEN-REC-DESCRIPTOR-ABSOLUTE
   (*REC ILIST (*OR $NIL (*CONS $INTEGER (*RECUR ILIST)))))
 =
 (*OR $NIL
      (*CONS $INTEGER
             (*REC ILIST (*OR $NIL (*CONS $INTEGER
                                    (*RECUR ILIST)))))))

Rule 3:
  (... (*REC FOO (... (*RECUR FOO) ...)) ...)
  => (*REC FOO (... (*RECUR FOO) ...))
From function PFOLD-RECS-IF-POSSIBLE

For example:
 (PFOLD-RECS-IF-POSSIBLE
   (*OR $NIL
        (*CONS $INTEGER
```

```
               (*REC ILIST (*OR $NIL (*CONS $INTEGER
                                     (*RECUR ILIST)))))))
 =
 (*REC ILIST (*OR $NIL (*CONS $INTEGER (*RECUR ILIST)))))

Comment: Rules 2 and 3 are the inverses of each other.  They represent
  the validity of opening up and folding *REC descriptors.  Each is
  appropriate, depending on context.  It is more common for
  OPEN-REC-DESCRIPTOR-ABSOLUTE to be invoked as a stand-alone
  canonicalization from elsewhere in the system, whereas
  PFOLD-RECS-IF-POSSIBLE is invoked in the normal course of
  canonicalization.

Rule 4:
 (*OR .. (*OR d1 .. d2) ..) => (*OR .. d1 .. d2 ..)
From function PCANONICALIZE-OR-DESCRIPTOR

Comment: The nested *OR may appear in any argument position of the
  surrounding *OR.

Rule 5:
 (*OR .. d1 .. d1 ..) => (*OR .. d1 .. ..)
From function PCANONICALIZE-OR-DESCRIPTOR

Comment: This rule characterizes the treatment of any multiple
  occurrence of a descriptor as arguments of an *OR.

Rule 6.
 (*OR .. *EMPTY .. ) => (*OR .. .. )
From function PCANONICALIZE-OR-DESCRIPTOR

Comment: This rule characterizes the treatment of any occurrence of
  *EMPTY as an argument of an *OR.

Rule 7.
 Where D does not contain a (*RECUR FOO),
  (*OR D (*REC FOO (*OR D .. (*RECUR FOO))))
  => (*REC FOO (*OR D .. (*RECUR FOO)))
From function PREMOVE-RECURSIVE-EXPANSION-DUPLICATES

Rule 8:
 (*OR (*CONS d1 d2) (*CONS d3 d2)) => (*CONS (*OR d1 d3) d2)
From PMERGE-OR-DESCRIPTOR-CONSES

Rule 9:
 (*OR (*CONS d1 d2) (*CONS d1 d3)) => (*CONS d1 (*OR d2 d3))
From PMERGE-OR-DESCRIPTOR-CONSES

Rule 10:
 (*OR .. *UNIVERSAL ..) => *UNIVERSAL
From PCANONICALIZE-OR-DESCRIPTOR

Rule 11:
 (*OR d) => d
From PCANONICALIZE-OR-DESCRIPTOR

Rule 12:
 (*OR) => *EMPTY
From PCANONICALIZE-OR-DESCRIPTOR

Rule 13:
 (*OR .. d1 .. d2 ..) => (*OR .. d2 .. d1 ..)
From PCANONICALIZE-OR-DESCRIPTOR

Comment: This rule allows the placement of *OR disjuncts into a
  canonical order.

Rule 14:
 (*CONS *EMPTY d) => *EMPTY
```

```
From PCANONICALIZE-CONS-DESCRIPTOR

Rule 15:
 (*CONS d *EMPTY) => *EMPTY
From PCANONICALIZE-CONS-DESCRIPTOR

Rule 16:
 (*REC FOO (*OR .. (*RECUR FOO) ..)) => (*REC FOO (*OR .. ..))
From function PCANONICALIZE-REC-DESCRIPTOR

Comment: With this notation, we indicate that (*RECUR FOO) is a
  top-level disjunct of the top-level *OR in the *REC body.  This
  reduction signifies that the recursion in the *REC descriptor is
  ill-formed.  The recursion embodied in it does not terminate because
  it does not descend into the CONS structure of an object, but rather
  repeats the same predicate about the same portion of the data object.
  The recursive occurrences, then, are of no significance, since they
  represent no data object not already represented by the descriptor
  on the right hand side.

Rule 17:
Where within (.. (*RECUR FOO) ..) there is no *OR enclosing the
(*RECUR FOO), but some *CONS enclosing it,

 (*REC FOO (.. (*RECUR FOO) ..)) => *EMPTY
From function PCANONICALIZE-REC-DESCRIPTOR

Comment: With this notation, we indicate a *REC structure with no
  terminating disjunct.  Thus, it can represent only infinite objects,
  and since these are not allowed in our subset, we reduce the form
  to *EMPTY.

Rule 18:
  Where (*dlist d1 .. dfoo .. dn) and (*dlist d1 .. dbar .. dn)
  are identical except that they differ in one position, dfoo vs. dbar:
    (*or ... (*dlist d1 .. dfoo .. dn) ... (*dlist d1 .. dbar .. dn) ...)
  = (*or ... (*dlist d1 .. (*or dfoo dbar) .. dn) ...)
From function PCANONICALIZE-OR-DESCRIPTOR

Comment: An *OR of *DLISTS is not a conventional descriptor.  But it is
  formed on exit from DUNIFY-DESCRIPTORS-INTERFACE when we call it with
  *DLIST descriptors, whenever multiple results are returned from its
  call to DUNIFY-DESCRIPTORS.  Thus, rather than there being a semantic
  notion of an *OR of *DLISTs, this packaging is just a protocol with
  any function which calls DUNIFY-DESCRIPTORS-INTERFACE with *DLIST
  arguments, allowing DUNIFY-DESCRIPTORS-INTERFACE to behave
  polymorphically.  The caller has the obligation to unpack the *OR.
  and to treat the list of results as a disjunction of possibilities.
  Hence, this canonicalization allows a merging of results.
```

6.8 The Containment Algorithm

One may think of type descriptors as defining sets of Lisp values. The elementary notion of containment of a descriptor TD1 in a descriptor TD2 is that membership in TD1 implies membership in TD2. Membership is defined in terms of our interpreter function INTERP-SIMPLE and therefore in terms of a binding of type variables to values.

The main feature of INTERP-SIMPLE with respect to variables is the binding list, which binds type variables from the descriptor to Lisp values. When the interpreter encounters a variable &i, as in the call (INTERP-SIMPLE-1 &i V B), it returns T if the binding of &i in B is equal to V, and NIL otherwise. So one may think of the set of Lisp values defined by a descriptor TD as the set of all values V such that there exists a binding B such that (INTERP-SIMPLE TD V B) is true. For example, the set of Lisp values defined by the descriptor (*CONS &1 $NIL) is the set

```
{ (CONS X NIL) | X is any Lisp value }
```

I.e., we could imagine an infinite collection of bindings in which for any Lisp value X, there is some binding in which &1 is mapped to X.

Our notion of containment is that TD1 is contained in TD2 if for any value V where some binding B causes (INTERP-SIMPLE TD1 V B) to be true, there is some binding B' such that (INTERP-SIMPLE TD2 V B') is also true. For example, (*CONS &1 $NIL) is contained in (*CONS &2 (*OR $NIL $T)), because for any value V and binding B such

```
(INTERP-SIMPLE-1 (*CONS &1 $NIL) V B)
```

is true, there exists a binding B' such that

```
(INTERP-SIMPLE-1 (*CONS &2 (*OR $NIL $T)) V B')
```

will be true, namely any binding which binds &2 to the same value to which &1 was bound in B. Our algorithm for checking containment, then, has the task of demonstrating that B' can be generated. If no such binding can be found, the algorithm must fail.

The containment algorithm plays three roles:

1. When handling a function call, TC-INFER invokes the containment algorithm to determine if the descriptors corresponding to the actual parameters are contained in the descriptors characterizing the guard for the function. This is the fundamental guard verification operation.

2. After TC-INFER-SIGNATURE has produced its version of the guard descriptor and the segments for the function, if the checker's guard descriptor differs from that computed by the inference algorithm, TC-INFER-SIGNATURE invokes the containment algorithm on the two guard descriptors twice, once in each direction. As discussed in Section 6.3, if containment exists in both directions, we may soundly replace the checker's guard with the inference algorithm's guard, the latter likely being in a more nicely canonicalized form.

3. TC-INFER-SIGNATURE also invokes the containment algorithm to determine that each segment in the checker signature is contained in some segment from the original signature.

6.8.1 Overview

Not surprisingly, the problem is much simpler if both descriptors are variable-free, so much so that this is treated as a special case and solved with a distinct and much faster algorithm. Furthermore, even when variables appear in the descriptors, some effort is made to reduce the problem to the variable-free case. All these judgements are made in the function CONTAINED-IN-INTERFACE, which is the entry point for the containment algorithm.

To rid the problem of variables, CONTAINED-IN-INTERFACE tries the following tricks. First, any variable which occurs only once in a descriptor is replaced with *UNIVERSAL. Since the only information a variable carries is that it refers to some particular, unspecified value, the only practical use for a variable is to represent a multiple occurrence of that value within a descriptor or descriptor list. Thus, when we have singleton variables, we replace them with *UNIVERSAL and canonicalize. Second, after this replacement, if there are no variables in the second descriptor, the variables in the first descriptor are irrelevant, and they can be replaced with *UNIVERSAL. This is sound because it is never unsound to "increase" the value set of TD1. By doing so, we could never create containment where it did not already exist. Moreover, it has no impact on the result, since if TD1 is a variable and if TD2 is variable-free and is not *UNIVERSAL, then we always say TD1 is not contained in TD2.

If trying these optimizations produces variable-free descriptors, then we call the simple algorithm CONTAINED-IN. If there are still variables to contend with, we first standardize the variables in the two descriptors apart and then call a complicated heuristic algorithm VCONTAINED-IN, which produces a list of *mappings*, each of which can be interpreted as a schema for constructing bindings witnessing the containment requirement. Then, we check the validity of the result from VCONTAINED-IN with yet another, much simpler algorithm ICONTAINED-IN. The application of ICONTAINED-IN to the result of VCONTAINED-IN is managed by the function MAPPINGS-DEMONSTRATE-CONTAINMENT. All these subsidiary algorithms will be described below.

6.8.2 The Formal Specification

In our system proof, the optimizations in CONTAINED-IN-INTERFACE are proven sound as part of the proof of CONTAINED-IN-INTERFACE-OK. The soundness of the CONTAINED-IN algorithm is stated in Lemma CONTAINED-IN-OK. The VCONTAINED-IN algorithm is not proved, as it serves only as a heuristic witness, suggesting a possible solution, but the MAPPINGS-DEMONSTRATE-CONTAINMENT/ICONTAINED-IN algorithm which checks this solution is specified in the lemmas ALL-DESCRIPTOR1-DISJUNCTS-OK-OK and ICONTAINED-IN-OK. All these lemmas will be stated below and proved in Section 7.8.

The formal specifications of these algorithms are as follows. Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value. As with DUNIFY-DESCRIPTORS-INTERFACE, the TD1 and TD2 arguments can be *DLIST forms of the same length, and in this case in the lemmas below, the value V is considered to be a list of values the same length as the *DLIST, and the calls to "I" denote calls to INTERP-SIMPLE rather than INTERP-SIMPLE-1. The top level specification for CONTAINED-IN-INTERFACE is embodied in the lemma:

```
Lemma CONTAINED-IN-INTERFACE-OK
 For any descriptors td1 and td2, value v, and binding b,

  (and (contained-in-interface td1 td2)
       (I td1 v b))
  =>
  For some b', (I td2 v b')
```

The specification for the variable-free algorithm CONTAINED-IN is:

```
Lemma CONTAINED-IN-OK

 For any descriptors td1 and td2, Lisp value v, and binding b1,

  (and
H1  (null (gather-variables-in-descriptor td1))
H2  (null (gather-variables-in-descriptor td2))
H3  (contained-in td1 td2)
H4  (I td1 v b1))
  =>
  For some b2, (I td2 v b2)

Comment: The b2 used here is irrelevant, since the descriptors are
variable-free.
```

CONTAINED-IN and ICONTAINED-IN take an extra argument, TERM-RECS, which serves only as part of a termination mechanism and hence has no bearing on this soundness lemma. Therefore we eliminate reference to it to minimize notation. One may consider it to be universally quantified in the lemmas.

The soundness specification for MAPPINGS-DEMONSTRATE-CONTAINMENT is given in Lemma ALL-DESCRIPTOR1-DISJUNCTS-OK-OK. In it we introduce the notion of a value reference.

Definition: A *value reference* **is a symbolic reference to some hypothetical value V satisfying the descriptor TD1 on the top-level call of CONTAINED-IN-INTERFACE. A value reference may take the form of the atom V or some nest of CAR, CDR, DLIST-ELEM, or REC-TAIL function calls around V.**

```
Lemma ALL-DESCRIPTOR1-DISJUNCTS-OK-OK
 For any descriptor td1 of the form (*or td1_1 .. td1_n)
 and descriptor td2, value reference vref, value v, and binding
 b1 covering the variables of td1,
  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  For all i, for some simple mapping m,
      (and (well-formed-mapping m vref b1)
           (icontained-in td1_i (apply-subst m td2) vref))
H3  (I td1 v b1) )
   =>
   For some b2, (I td2 v b2)
```

The mappings in H2 are provided by the heuristic function VCONTAINED-IN. By definition, MAPPINGS-DEMONSTRATE-CONTAINMENT puts TD1 into the appropriate *OR form, and given the mappings from VCONTAINED-IN, is the direct implementation of H2.

Finally, the specification of the ICONTAINED-IN algorithm is Lemma ICONTAINED-IN-OK.

```
Lemma ICONTAINED-IN-OK:

 For any descriptors td1 and td2, simple mapping m, Lisp value v,
 symbolic value reference vref, and binding b1 covering the
 variables in td1,

  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  (well-formed-mapping m vref b1)
H3  (icontained-in td1 (dapply-subst-list-1 m td2) vref)
H4  (I td1 v b1) )
  =>
  For some b2, (I td2 v b2)
```

6.8.3 CONTAINED-IN

In the absence of variables, descriptors can be readily seen as representing sets of values, and the notion of containment can be easily grasped as a subset operation. This mindset may help with understanding the containment algorithm for variable-free descriptors.

The function CONTAINED-IN takes three parameters, the two descriptors TD1 and TD2, and a TERM-RECS argument which functions just as it does in DUNIFY-DESCRIPTORS to prevent an infinitely recursive unwinding of overlapping pairs of *REC descriptors. The recursive pattern of CONTAINED-

IN is quite similar to that of DUNIFY-DESCRIPTORS, including its employment of a collection of rules for handling the case where both TD1 and TD2 are *REC descriptors.

The following case analysis is used. The cases are considered in order, so each assumes the descriptors being considered do not suit any of the previous cases.

1. For any descriptor td, td is contained in td.
2. *EMPTY is contained in any descriptor.
3. No descriptor is contained in *EMPTY.
4. *UNIVERSAL is contained in no descriptor.
5. Any descriptor is contained in *UNIVERSAL.
6. An *OR descriptor is contained in TD2 if all its disjuncts are contained in TD2.
7. If TD1 is a *REC descriptor, then:
   a. If TD2 is an *OR descriptor, one of whose disjuncts is *name isomorphic* to TD1, then TD1 is contained in TD2. By name isomorphism, we mean either that the two *REC descriptors are equal or that one is exactly like the other except for bearing a different label.
   b. If TD2 is a *REC descriptor, we try each of the *REC rules described below. Each rule has an enabling condition and an action. The first rule whose enabling condition is satisfied provides the verdict.
   c. If TD2 is not a *REC, we first see if we have on previous recursion encountered the TD1-TD2 pair. If so, we return NIL. This is the TERM-RECS test, since the TERM-RECS parameter contains a list of all previously encountered pairs. Otherwise, we open TD1 and recur, CONS-ing the TD1-TD2 pair onto the TERM-RECS argument.
8. If TD2 is an *OR, then TD1 is contained in TD2 if it is contained in one of the disjuncts of TD2.
9. If TD2 is a *REC, we administer the TERM-RECS test as above. If we are not looping, we open TD2 and recur, with the augmented TERM-RECS list.
10. If both descriptors are CONS-es, we have containment if the respective CARs and CDRs both exhibit containment.
11. If both descriptors are *DLISTs, we have containment if we have pairwise containment on all the arguments.
12. Otherwise, containment fails.

Before trying the *REC rules, we canonicalize both descriptors in the same way as we did for the DUNIFY-DESCRIPTORS *REC rules, turning all the top level *ORs into binary forms, where the first disjunct is a disjunction of non-replicating forms, and the second is a disjunction of replicating forms (forms containing the *RECUR).

The notation used in the rules is like that used in the DUNIFY-DESCRIPTORS rules. For instance, *REC forms defined on the left hand side are referred to by their labels on the right hand side. A reference to "foo" or "bar" on the right hand side of a rule is a shorthand for the entire *REC form of that name on the left hand side. For a full explanation of the notation, see Section 6.6.3. Recall that the enabling condition is embodied in the form of the left hand side of the equality. Because the rules are considered in order, each enabling condition effectively includes the negation of all previous enabling conditions. The numbering of the rules is somewhat irregular, so that it will correspond to analogous ICONTAINED-IN *REC rules, to follow later. A rule whose name contains a "’" is a minor variant of its predecessor.

In reading each of these rules, remember the CONTAINED-IN algorithm is only invoked when both of its arguments are variable-free. Thus, all component descriptors are variable-free, and thus CONTAINED-IN may be called recursively.

```
Rule Contain-*REC1
 Where the two *recs differ only in name
 (contained-in (*rec foo ( .. (*recur foo) .. ))
               (*rec bar ( .. (*recur bar) .. )))
 = t

Rule Contain-*REC2
 (contained-in (*rec foo (*or $nil (*cons d1 (*recur foo))))
               (*rec bar (*or $nil (*cons d2 (*recur bar)))))
 =
 (contained-in d1 d2)

Rule Contain-*REC2'
 (contained-in (*rec foo (*or $nil (*cons (*recur foo) d1)))
               (*rec bar (*or $nil (*cons (*recur bar) d2))))
 =
 (contained-in d1 d2)

Rule Contain-*REC3
 Where d1 and d3 are either primitive non-cons descriptors or
 *ORs of primitive non-cons descriptors
 (contained-in (*rec foo (*or d1 (*cons d2 (*recur foo))))
               (*rec bar (*or d3 (*cons d4 (*recur bar)))))
 =
 (and (contained-in d1 d3) (contained-in d2 d4))

Rule Contain-*REC3'
 Where d1 and d3 are either primitive non-cons descriptors or
 *ORs of primitive non-cons descriptors
 (contained-in (*rec foo (*or d1 (*cons (*recur foo) d2)))
               (*rec bar (*or d3 (*cons (*recur bar) d4))))
 =
 (and (contained-in d1 d3) (contained-in d2 d4))

Rule Contain-*REC5
 (contained-in (*rec foo (*or d1 (*cons d2 (*recur foo))))
               (*rec bar (*or d3 (*cons d4 (*recur bar)))))
 =
 (or (and (contained-in d1 d3) (contained-in d2 d4))
     (contained-in foo d3)
     (and (contained-in d1 bar) (contained-in d2 d4)))

Rule Contain-*REC6
 No conses in d1 or d4.
 (contained-in (*rec foo (*or d1 (*cons d2 (*cons d3 (*recur foo)))))
               (*rec bar (*or d4 (*cons d5 (*recur bar)))))
 =
 (and (contained-in d1 d4) (contained-in d2 d5) (contained-in d3 d5))

Rule Contain-*REC6'
 No conses in d1 or d4.
 (contained-in (*rec bar (*or d4 (*cons d5 (*recur bar))))
               (*rec foo (*or d1 (*cons d2 (*cons d3 (*recur foo))))))
 =
 nil

Rule Contain-*REC7
 No conses in d1 or d4.
 (contained-in (*rec foo (*or d1 (*cons (*recur foo) (*recur foo))))
               (*rec bar (*or d2 (*cons (*recur bar) (*recur bar)))))
 =
 (contained-in d1 d2)

Rule Contain-*REC8
```

```
 No conses in d1 or d5.
 (contained-in
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*recur bar)))))
 =
 (and (contained-in d1 d5)
      (contained-in d2 d6)
      (contained-in d3 d6)
      (contained-in d4 d6))

Rule Contain-*REC8'
 No conses in d1 or d5.
 (contained-in
   (*rec bar (*or d5 (*cons d6 (*recur bar))))
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo)))))))
 =
 nil

Rule Contain-*REC9
 No conses in d1 or d5.
 (contained-in
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*cons d7 (*recur bar))))))
 =
 nil

Rule Contain-*REC9'
 No conses in d1 or d5.
 (contained-in
   (*rec bar (*or d5 (*cons d6 (*cons d7 (*recur bar)))))
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo)))))))
 =
 nil

Rule Contain-*REC10
 No conses in d2 or d4
 (contained-in (*rec foo (*cons d1 (*or d2 (*recur foo))))
               (*rec bar (*cons d3 (*or d4 (*recur bar)))))
 =
 (and (contained-in d1 d3) (contained-in d2 d4))

Rule Contain-*REC11
 No conses in d2 or d3.
 (contained-in (*rec foo (*cons d1 (*or d2 (*recur foo))))
               (*rec bar (*or d3 (*cons d4 (*recur bar)))))
 =
 (and (contained-in d1 d4) (contained-in d2 d3))

Rule Contain-*REC11'
 d3 is a primitive or a disjunction of primitives,
 (contained-in (*rec bar (*or d3 d4))
               (*rec foo (*cons d1 d2)))
 =
 nil

Rule Contain-*REC13
 No conses in d1, d3, d5, or d7
 (contained-in
   (*rec foo (*or d1 (*cons d2 (*or d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*or d7 (*cons d8 (*recur bar)))))))
 =
 (and (contained-in d1 d5)
      (contained-in d2 d6)
      (contained-in d3 d7)
      (contained-in d4 d8))

Rule Contain-*REC0
 (contained-in foo bar) =  nil
```

```
Comment: Rule Contain-*REC0 is a default, to be employed to administer
  failure when the preconditions for all the other rules have not
  been satisfied.
```

6.8.4 The VCONTAINED-IN Heuristic and Its Setting

Recall the top level specification for containment.

```
Lemma CONTAINED-IN-INTERFACE-OK
 For any descriptors td1 and td2, value v, and binding b,

  (and (contained-in-interface td1 td2)
       (I td1 v b))
  =>
  For some b’, (I td2 v b’)
```

In the case where one or both of the descriptors contains a variable which could not be removed by CONTAINED-IN-INTERFACE, to demonstrate containment we must demonstrate the existence of a binding B’ such that the containment relation defined in terms of INTERP-SIMPLE holds. Since the binding B’ will depend on information from B, as well as TD1 and TD2, without knowing B, we cannot generate specific bindings B’. So instead, we attempt to generate what we will call a *mapping* which shows precisely how B’ can be constructed, given some arbitrary B for which (I TD1 V B) is true. Finding this mapping is the job of the function VCONTAINED-IN.

This task is relatively difficult, and a proof of the algorithm embodied in VCONTAINED-IN would be daunting. But our soundness proof will be in the same spirit which led us to use a checker which is proven correct validate the function signatures generated by the inference system. Rather than proving the soundness of VCONTAINED-IN directly, we prove the correctness of an algorithm ICONTAINED-IN which checks that a mapping produced by the unverified algorithm VCONTAINED-IN does indeed demonstrate that the containment relation holds.

Both VCONTAINED-IN and ICONTAINED-IN are similar in structure to CONTAINED-IN. But VCONTAINED-IN produces a mapping structure, described below, which can be easily transformed into a list of simple mappings. We can then apply each mapping from this list in turn to TD2 and employ the algorithm ICONTAINED-IN, which is a fairly simple extension of CONTAINED-IN, to demonstrate that a binding B’ can be constructed which will make (INTERP-SIMPLE TD2 V B’) true whenever (INTERP-SIMPLE TD1 V B) is true.

Syntactically, simple mappings conform to the following quasi-BNF style grammar.

```
<mapping> ::= ( <elem>* )

<elem> ::= (<td2 variable name> . <target>)

<target> ::= <td1 variable name> | $nil | $t | <symbolic value>

<symbolic value> ::=
  V | (CAR <symbolic value> ) | (CDR <symbolic value> ) |
  (DLIST-ELEM <symbolic value list> <integer> ) |
  (REC-TAIL <symbolic-value> )

<symbolic value list> ::= ( <symbolic value>* )

<td1 variable name> :: a type variable appearing in td1

<td2 variable name> :: a type variable appearing in td2
```

REC-TAIL is a function which returns the atom which is the final CDR of a list.

```
(DEFUN REC-TAIL (L) (IF (ATOM L) L (REC-TAIL (CDR L))))
```

DLIST-ELEM is a function which, given a list L and a positive integer i less than the length of L, returns the ith element of L. This function is here only to index over the value list considered as V when TD1 and TD2 are *DLISTs.

```
(DEFUN DLIST-ELEM (L I)
  (IF (EQUAL I 1) (CAR L) (DLIST-ELEM (CDR L) (- I 1))))
```

A mapping directs the construction of B' in the following manner. If a <target> is a <td1 variable name> the right hand side of the corresponding binding would be the binding of the <target> in B. If a <target> is $NIL, the right hand side would be NIL. Similarly for $T and T. If the <target> is a symbolic value reference, such as V, (CAR V), or (REC-TAIL V), the right hand side would be the value of the corresponding component of the Lisp value under consideration by INTERP-SIMPLE. This value is the one represented symbolically by V in the statement of CONTAINED-IN-INTERFACE-OK. Note that we could have used a symbolic value in all cases, but, as we shall see in the description of the algorithms, this would not serve to handle multiple occurrences of variables within the descriptors.

ICONTAINED-IN determines whether the containment relationship exists between TD1 and TD2 under the mapping. If it does, given B and V, the mapping can be easily transformed into a binding B' which will satisfy the containment requirement in terms of INTERP-SIMPLE. If any of the mappings in the list formulated by VCONTAINED-IN can be validated, we have demonstrated containment. But as we shall later see, if there is disjunction in TD1, it may be the case that no single mapping will demonstrate containment. It is sufficient, however, to canonicalize TD1 so that every disjunction lying outside of any embedded *REC descriptor is raised to the top, i.e., so that TD1 is a disjunction of descriptors, each of which is *OR-free down to embedded *REC descriptors. Then, if for all disjuncts of TD1, there exists a simple mapping which demonstrates containment, containment is thereby demonstrated for the disjunction. We shall prove the correctness of ICONTAINED-IN, which as a checker will contribute to the proof of the correctness of CONTAINED-IN-INTERFACE.

VCONTAINED-IN is more complicated than ICONTAINED-IN, since it has to discover the appropriate mappings, rather than just validating them. To support this discovery requires a more elaborate notion of a mapping. VCONTAINED-IN takes as parameters two descriptors TD1 and TD2, an initial (elaborate) mapping M, a TERM-RECS argument (initially NIL), and a value reference VREF by which we will refer to some arbitrary value V, and it returns a list of (elaborate) mappings $M_1..M_n$.

```
(VCONTAINED-IN td1 td2 m nil vref) = (m1 .. mn)
```

Failure is signified by returning an empty list of mappings. The implicit assumption is that there exists a binding B such that (INTERP-SIMPLE TD1 V B) is true. (M_1 .. M_n) may be thought of as extensions of M. Again, the nature of the result is to show precisely how a binding B' can be constructed from B and V such that (INTERP-SIMPLE TD2 V B') is true. A list of mappings demonstrates containment if for any value satisfying TD1 under B, one of the mappings in the list creates a B' such that the value satisfies TD2 under B'.

To characterize elaborate mappings, we modify the preceding grammar for mappings by replacing

```
<mapping> ::= ( <elem>* )
```

with

```
<mapping> ::= ( <elem>* ) | (*CHOICE <mapping-list>* )

<mapping list> ::= ( <mapping>* )
```

VCONTAINED-IN returns a list of mappings. In a sense of validity to be explained below, a mapping list is a collection of mappings, all of which must be present in order to demonstrate containment. A *CHOICE encapsulates a collection of mapping lists, each of which individually demonstrates containment.

A list of mappings is necessary to characterize what happens when TD1 is an *OR. Consider the example:

```
(VCONTAINED-IN '(*OR $NIL $T) '&1 NIL NIL vref)
=
(((&1 . NIL)) ((&1 . T)))
```

Containment is guaranteed if when the value V is NIL, &1 is bound to NIL, and if when the value V is T, &1 is bound to T. Obviously, V cannot be simultaneously T and NIL, and the bindings to handle each situation are mutually exclusive. So we use a list of bindings to characterize what must be possible in order for containment to exist. The list of mappings enumerates the list of choices for B' which might be needed.

A *CHOICE indicates that, for all of the mapping lists which follow, one of its mappings will generate a binding which will be acceptable. *CHOICE arises when TD2 is an *OR. For example,

```
(VCONTAINED-IN '$INTEGER '(*OR &1 &2) NIL NIL 'V)
 =
((*CHOICE (((&1 . V))) (((&2 . V)))))
```

Containment is guaranteed either when &1 is bound to the integer in question, or &2 is similarly bound. In either case, it does not matter what the other variable is bound to. We can see this in the setting of our interpreter.

```
(INTERP-SIMPLE $INTEGER V B)
```

expands directly to (INTEGERP V).

```
(INTERP-SIMPLE (*OR &1 &2) V B')
```

expands to

```
(OR (EQUAL V (CDR (ASSOC &1 B'))) (EQUAL (CDR (ASSOC &2 B'))))
```

A B' constructed from this mapping can map either &1 or &2 to the integer V, and the predicate will be true.

When generating mapping lists witnessing containment for *CONS or *DLIST structures, we compute the mappings validating containment for the CARs. If there are no such mappings, we fail. Then we use each of the mappings as the base mapping for the recursive analysis of the CDRs. If a base mapping is a *CHOICE, we can succeed if any one of the choices leads to a positive result. But in order for containment to be demonstrated for the *CONS or *DLIST, every base mapping from the CAR must lead to a positive result.

Consider the following example:

```
(VCONTAINED-IN '(*CONS $INTEGER $NIL)
               '(*CONS (*OR &1 &2) &2)
               NIL NIL 'v)
 =
(((&1 . (CAR V)) (&2 . $NIL)))
```

To see how his result was derived, first notice that the preceding example gives us the result for the CARs (giving (CAR V) rather than V as the symbolic value reference):

```
((*CHOICE (((&1 . (CAR V)))) (((&2 . (CAR V))))))
```

This is a singleton list composed of a single *CHOICE mapping, thus we need only show that the *CHOICE will work in the CDR. So we proceed to the CDRs with our choices. There are two recursive calls of VCONTAINED-IN, one for each choice. Since each choice is a list of mappings, each mapping in one of the lists must be an effective base mapping for the CDR. But in this case, since each list is a singleton, our solution can be produced by one of the mappings from either:

```
(VCONTAINED-IN '$NIL '&2 '((&1 . (CAR V))) NIL (CDR V))
```

or

```
(VCONTAINED-IN '$NIL '&2 '((&2 . (CAR V))) NIL (CDR V))
```

In either case, we must map &2 to $NIL to get a solution. This poses no conflict with our first choice, which places no constraint on &2, and the first mapping is extended with the mapping element (&2 . $NIL) to produce the result:

```
(((&1 . (CAR V)) (&2 . $NIL)))
```

But the second choice gets ruled out, because &2 could not be mapped to both (CAR V) and $NIL. If (CAR V) is anything but NIL, there would be inconsistency. Thus, our ultimate result is the singleton list of mappings:

```
(((&1 . (CAR V)) (&2 . $NIL)))
```

On the other hand, when a disjunction is in TD1, we require that each of the disjuncts be contained under some mapping. The same mapping need not apply to all disjuncts. This generation of multiple mappings is the reason VCONTAINED-IN returns a list of mappings rather than a single one. Consider the example:

```
(VCONTAINED-IN '(*OR &1 $INTEGER) '&2 NIL NIL 'V)
 =
(((&2 . &1)) ((&2 . V)))
```

By the definition of INTERP-SIMPLE, for any value V for which

```
(INTERP-SIMPLE (*OR &1 $INTEGER) V B)
```

is true, either (INTERP-SIMPLE &1 V B) or (INTERP-SIMPLE $INTEGER V B) is true. If the &1 in (*OR &1 $INTEGER) is the operative descriptor, &2 maps to &1. If $INTEGER is operative, &2 maps to V. Any subsequent extension of the mappings must allow both of these cases, or else the proper containment relation will not hold in all cases. For example,

```
(VCONTAINED-IN '(*CONS (*OR &1 $INTEGER) &1)
               '(*CONS &2 &2)
               NIL NIL 'V)
```

```
  =
 (ALL-VCONTAINED-IN '&1 '&2 '(((&2 . &1)) ((&2 . (CAR V)))))
  =
 NIL
```

because in the case where the CAR, and thus &2, corresponds to $INTEGER and &2 maps to (CAR V), the CDR, and thus also &2, cannot necessarily correspond to an arbirtrary &1. An example which validates this failure is the value '(10 . NIL). The only binding b1 for which

```
(INTERP-SIMPLE (*CONS (*OR &1 $INTEGER) &1) (10 . NIL) b1)
```

is true is ((&1 . NIL)). But no binding B2 will make

```
(INTERP-SIMPLE (*CONS &2 &2) (10 . NIL) B2)
```

true. But consider:

```
 (VCONTAINED-IN (*CONS (*OR &1 $INTEGER) &1) (*CONS &2 &3) NIL NIL V)
  =
 (((&2 . &1) (&3 . &1)) ((&2 . (CAR V)) (&3 . &1)))
```

Containment exists in this case, since the mapping arising from the CDR containment is in both cases consistent with both of the cases from the CAR.

Moreover, the same phenomenon is the reason *CHOICE encompasses a list of mapping lists, rather than just a list of mappings. For example,

```
(VCONTAINED-IN-INTERFACE (*CONS &1 &1)
                         (*OR (*CONS &2 &3) (*CONS &2 &2)))
=
((*CHOICE (((&2 . &1) (&3 . &1))) (((&2 . &1)))))
```

Generating mapping lists for witnessing containment of one *REC descriptor within another *REC descriptor is handled as in DUNIFY-DESCRIPTORS and CONTAINED-IN. Rather than employing a general algorithm, we will employ a set of rules, each suited to descriptors of a particular form. An example of such a rule is:

```
 (VCONTAINED-IN (*REC FOO (*OR $NIL (*CONS D1 (*RECUR FOO))))
                (*REC BAR (*OR $NIL (*CONS D2 (*RECUR BAR))))
                M
                VREF)
  =
 (VCONTAINED-IN D1 D2 M *BOGUS-VREF*)
```

The FOO descriptor is contained in the BAR descriptor if D1 is contained in D2. Thus, the problem is reduced to one which does not involve the top level *REC descriptors. The VREF parameter on the right hand side is denoted bogus because there can be no variables in D1 or D2, and hence the VREF will never come into play. Each rule has a set of conditions which must be fulfilled in order for it to be invoked on a given problem. As a default, we have a final rule which will fire unconditionally when no other rule will, and which declares that containment does not exist. This action is not very satisfying, but is sound. (There being no formal notion of soundness for VCONTAINED, the reader can take this in the informal sense.) The *REC descriptors are put in a special canonical form prior to being matched against the rule enablers, as with DUNIFY-DESCRIPTORS and CONTAINED-IN. The rules are sufficiently similar in form to the CONTAINED-IN *REC rules that they do not merit stating here, but they may be found in the source code, stored in the constant REC-VCONTAINMENT-RULES. Since VCONTAINED-IN is just a heuristic function about which we make no claims of correctness, the details of the rules are not so important.

6.8.5 The ICONTAINED-IN Checker

As previously stated, the function MAPPINGS-DEMONSTRATE-CONTAINMENT and its subsidiaries manage the application of the ICONTAINED-IN algorithm. It raises all disjunction outside of *REC descriptors in TD1 to the top of the form. It massages the mapping list produced by VCONTAINED-IN into a list of simple mappings (without *CHOICE). This is done by simply appending the lists within a *CHOICE, forming a single list of simple mappings. Essentially, the "AND" represented by a *CHOICE is weakened to an "OR".[31]Then it organizes calls to ICONTAINED-IN to validate that for every disjunct of TD1, given any value V and binding B such that (INTERP-SIMPLE TD1 V B), some mapping provides a method for constructing a binding B’ such that (INTERP-SIMPLE TD2 V B’). This is sufficient to establish containment of TD1 in TD2.

In preparing each call to ICONTAINED-IN, MAPPINGS-DEMONSTRATE-CONTAINMENT actually substitutes the mapping under consideration into TD2 to form the second argument. A mapping maps TD2 variables to either a TD1 variable indicating that in B’ the variable from TD2 would be bound to the same value as the variable from TD1 is bound to in B, or it maps a TD2 variable to some CAR-CDR nest of the symbolic value VREF, indicating that in B’ the same variable would be bound to the same nest on the value V.

Thus, ICONTAINED-IN, which otherwise is quite like CONTAINED-IN, differs in that it must be prepared to handle very simple scenarios with variables and with our symbolic value references in TD2. It makes judgements on value references by constructing its own symbolic value reference as it recurs down through the *CONS structure of TD1 and TD2. This reference is a parameter of ICONTAINED-IN. On the initial call the actual parameter is ’V, which is the same symbolic root which was given to VCONTAINED-IN. Every time ICONTAINED-IN recurs into a *CONS, or a *DLIST, the proper destructor (CAR, CDR, or DLIST-ELEM) is wrapped around the value reference. The following rules for variables and symbolic value references augment the treatments of other forms, which are as in CONTAINED-IN.

1. If TD2 is a type variable, if TD1 is the identical type variable, TD1 is contained in TD2, otherwise it is not.
2. If TD2 is a symbolic value reference, if it is identical to the current value of the value reference parameter to ICONTAINED-IN, then TD1 is contained in TD2, otherwise it is not.

As with UNIFY-DESCRIPTORS, CONTAINED-IN, and VCONTAINED-IN, the case where both TD1 and TD2 are *REC descriptors is handled with a collection of special case rules. The method is identical, and the rules are similar to those used in CONTAINED-IN. The specific rules used in ICONTAINED-IN may be found in the Appendix B-I, which also includes the proofs of several of the rules. The notation is the same kind used for the CONTAINED-IN rules.

---

[31]In the implementation code, these simple mappings are referred to as *semibindings*.

Chapter 7

THE PROOF OF SOUNDNESS

This chapter presents the proof of soundness for the signature checker, i.e., of its validation of the function signatures produced by the inference system. We discuss the organization of the proof and each of its main ideas, state some key lemmas, and present some detailed proofs. In the case of more lengthy or subsidiary proofs, we point to their appearance in Appendix B. The order of presentation will be top-down, though this means, of course, that at almost every level the validity of the proof will rest on the proofs of lemmas presented later in the discussion. Most of these lemmas, however, have been previously stated in Chapter 6.

This is a proof of the correctness of an algorithm. The algorithm is implemented as a collection of applicative Common Lisp functions. Although the proof deals with fairly fine details of the code, it is not strictly a code proof. Subjective judgements were made about the points at which the implementation was involved with coding details which were not particularly relevant to the algorithm. For example, the precise structure of data objects and their abstraction and manipulation is not addressed in the proof. The objects are viewed at an abstract level which is fully satisfactory for complete comprehension of the algorithm. On the other hand, if a major transformation of these objects from one organization to another is relevant to the algorithm, the proof will deal with the transformation. An example is the cross product operation where the information computed about the types of the actual arguments in a function call is transformed prior to matching against the signature segments for the called function. (This was discussed in Section 6.5.2).

In the next section, we give a rough overview of the proof in terms of its main components, how they relate to one another, and which formal structures they manipulate. In the following section, we provide an informal sketch of the top level proof of soundness, thus providing a sense of direction for the sections which follow. Then we proceed with the proof itself, starting with a section containing the proofs of the top level lemmas. After a brief overview, we prove Lemma GUARD-COMPLETE, which is one of our top-level lemmas and the foundation for the treatment of guard verification. In the section "Validation of the Guard Descriptor", we prove Lemma TC-INFER-SIGNATURE-GUARD-OK, an important supporting lemma for the proof of Lemma TC-SIGNATURE-OK, our top level goal, which follows in Section 7.3.3. In the remaining sections, we prove important lemmas about subsidiary algorithms: the principal recursive function in the checker TC-INFER, the descriptor unifier DUNIFY-DESCRIPTORS-INTERFACE, the descriptor canonicalizer PCANONICALIZE-DESCRIPTOR, and the containment algorithm CONTAINED-IN-INTERFACE.

Fruitful reading of this chapter will require familiarity with the material presented in Chapter 5, which presents the formal semantics of the system, and Chapter 6, which gives detailed descriptions of the algorithms in the checker.

7.1 Structure of the Proof

There are four basic components to the proof, each corresponding to a subsystem of the signature checker. These are

- The descriptor canonicalizer, whose top level function is PCANONICALIZE-DESCRIPTOR and whose top level soundness lemma is PCANONICALIZE-DESCRIPTOR-OK,
- The descriptor unification algorithm, whose top level function is DUNIFY-DESCRIPTORS-INTERFACE and whose top level lemma is DUNIFY-DESCRIPTORS-INTERFACE-OK,
- The containment algorithm, whose top level function is CONTAINED-IN-INTERFACE and whose top level lemma is CONTAINED-IN-INTERFACE-OK, and
- The main checker algorithm, whose top level function is TC-SIGNATURE and whose top level lemmas are TC-SIGNATURE-OK and GUARD-COMPLETE.

Two interpreters are central to the formal semantics of the type system. One, composed of the functions INTERP-SIMPLE-1 and INTERP-SIMPLE, takes a type descriptor or a list of descriptors respectively, a Lisp value or list of values respectively, and a binding of type variables to values and determines whether the value is a member of the set of values represented by the descriptor under the binding. The Lisp evaluator E takes a Lisp form, an environment binding Lisp variables to values, a world of defined functions, and a non-negative integer clock and returns either a special value indicating the clock value was not adequately large to allow full evaluation, a special value indicating a guard violation occurred on some function call during the evaluation, or the Lisp value of the form in the environment if the evaluation succeeded.

The realm of discourse in the canonicalization proof consists of descriptors and descriptor lists, Lisp values, type variable bindings, and INTERP-SIMPLE. Bindings are present only because they are required by INTERP-SIMPLE. They are not manipulated in any way. E does not appear at all. In the global hierarchy of the proof, the canonicalization proof is self-contained. It relies on none of the other components.

The unification proof also involves descriptors and descriptor lists, values, bindings, and INTERP-SIMPLE. Also playing a role is another entity, a substitution, which maps type variables to other descriptors. A significant companion function is INTERP-SUBSTS, a predicate which determines whether a substitution is consistent with a binding. Substitutions appear only in the proof about the unification algorithm. Again, E makes no appearance. The unifier proof relies directly on the proof of the canonicalizer, but not on any other principal component.

As with the canonicalizer and the unifier, the containment proof involves descriptors and descriptor lists, values, bindings, and INTERP-SIMPLE. A significant local notion in the containment proof is that of a mapping, which embodies a method for constructing one type variable binding from another. Thus, the properties and the manipulation of bindings are points of interest in this proof. In the proof hierarchy, the containment proof uses the canonicalization lemmas, but does not rest on the unifier or the top level lemmas.

The top level proof about the checker uses almost everything previously mentioned. It manipulates function signatures, of which descriptors are components, values, type variable bindings, Lisp variable environments, the database of function signatures, Lisp forms, the world, and the clock. Both INTERP-SIMPLE and E are significantly explored. As both containment and unification are key concepts in the checker, their respective soundness lemmas are used heavily in the proof, along with the ubiquitous canonicalizer. Only in the top level proofs do Lisp forms, the evaluator E, the clock, and the world come into play.

7.2 The Top-Level Approach

Here we describe the very highest level of the formal setting of the inference algorithm and the checker, providing some semi-formal intuition for understanding why the lemmas we have proved give a proof of soundness for the system. In particular, we argue the soundness of using the untrusted signature generated by the inference algorithm as a basis for the checker's computation of a trusted signature, and how the containment relation can reflect that trust back to the original signature. This discussion is intended simply to motivate the ideas behind the top level proof. The rigorous proof is given later as the proof of Lemma TC-SIGNATURE-OK (Section 7.3.3).

Let f^* be the graph of a function f whose definition has been submitted to the inference system. The inference algorithm applied to f generates some function signature S. We want to ensure that S is a valid signature for f, i.e., that $f^* \subseteq S$, where by "$\subseteq$" we mean that the signature embodies a set of ordered tuples of which f^* is a subset. Embodied in the specification is the notion that any tuple of values [arg_1 .. arg_m (f arg_1 .. arg_m)] (where f is m-ary) *satisfies* S, meaning that if the values [arg_1 .. arg_m] satisfy the guard of the function, then the tuple satisfies some segment of S. Satisfaction is always determined by the function INTERP-SIMPLE. A segment (td_1 .. td_m) -> td of S is satisfied if there is some binding b of the type variables in the segment such that,

$$\texttt{(INTERP-SIMPLE (td}_1\texttt{ .. td}_m\texttt{ td) (arg}_1\texttt{ .. arg}_m\texttt{ (f arg}_1\texttt{ .. arg}_m\texttt{)) b)}$$

We demonstrate that $f^* \subseteq S$ by employing the algorithm TC-INFER-SIGNATURE to find a signature S' such that $f^* \subseteq S'$, and then by using the verified algorithm CONTAINED-IN-INTERFACE to check that $S' \subseteq S$. Thus, by the transitivity of inclusion, $f^* \subseteq S$. Given the success of the algorithm, we want to prove a lemma verifying that $f^* \subseteq S'$, and with the help of the containment algorithm that $S' \subseteq S$. This would be tantamount to a proof of Lemma TC-SIGNATURE-OK.

This line of reasoning is totally straightforward where f is a non-recursive function, i.e, a function for which the body of its definition does not contain a call to itself. But if f is recursive, we must recognize that TC-INFER-SIGNATURE assumes the correctness of S in deriving S' (i.e., it assumes that $f^* \subseteq S$). Why is it valid to allow this assumption? A simple inductive argument on the graph of the function will illustrate.

Imagine the graph of a recursive function as being the union of the graphs of an infinite number of non-recursive functions, with the first being the empty function, the second being the ordered pairs formed by cases where the function terminates without recursion, the next being cases where it terminates with at most one recursion, and so forth. For a given function f, call these non-recursively defined partial functions $\perp$, f^1, f^2, .. f^n. Each function definition f^i is non-recursive in the sense that it is formed from the definition of f by replacing the recursive calls with calls to f^{i-1}. Since at each level f^i, the function perhaps adds tuples to the graph of f^{i-1} without taking any away, the following relationship holds:

$$\perp \subseteq \mathbf{f}^1 \subseteq \;..\; \subseteq \mathbf{f}^n \;..\; \subseteq \mathbf{f}^*$$

where f^* is the complete graph of the function.

TC-INFER-SIGNATURE assumes the soundness of the database fs, which contains the signatures of all previously defined functions. In the case when the function being analyzed, f, is recursive, this includes the signature S, which has been ascribed to f (or more precisely, f^n).

We have already argued that for all non-recursive functions f,

```
f^n ⊆ (TC-INFER-SIGNATURE f fs)
```

holds. Now we will show by induction on n that it holds for recursive functions, too. The base case is trivial:

```
⊥ ⊆ (TC-INFER-SIGNATURE f fs)
```

For the inductive step, assume

```
f^n ⊆ (TC-INFER-SIGNATURE f fs)
```

We wish to prove

```
f^n+1 ⊆ (TC-INFER-SIGNATURE f fs)
```

Note that when we say

```
(TC-INFER-SIGNATURE f fs)
```

we really mean

```
(TC-INFER-SIGNATURE f^n+1 (cons f^n-sig fs))
```

since the idea behind the checker is to treat f as a non-recursive function where calls to f are treated as calls to f^n, whose signature was computed by the inference algorithm and is represented by f^n-sig. Since f^n is non-recursive, our assumption of the non-recursive case establishes the validity of f^n-sig. Given the casting of f^{n+1} as a non-recursive function which calls f^n rather than itself, assuming we have a valid signature for f^n, and given that we have established our lemma for non-recursive functions, we establish our goal.

Originally, we attacked this problem with an evaluator which assumed that all functions terminate normally and simply failed when there was a guard violation. The clock which now appears in the definition of E (Section 5.4) appeared in neither the original evaluator nor in any of the soundness lemmas. Normal termination is a basic requirement placed on all new functions submitted to the system, and under the assumption that all functions terminate for all values satisfying the guard, it is sufficient to guarantee that the evaluator will return normally unless it finds a guard violation. Hence, we can count on the use of our evaluator in our lemmas.

But the problem with this approach is that, since $\perp$, f^1, f^2, .. f^n are partial functions, the guard of f, which we have ignored to this point in this discussion, does not guarantee they will return a value for all inputs satisfying the guard. Specifically, if a call to f^n requires more than n "recursive" calls, the evaluation of the chain of partial functions will collide with $\perp$. If we attempt to patch the guards of each f^i, the fix will require turning the guards into forms which cannot result in complete descriptors, thus violating a crucial assumption for sound use of signatures.

To remedy the problem, we introduced a second kind of break in the E function, BREAK-OUT-OF-TIME, and the clock parameter which ticks down on every entry to a function body. We modified all the top level lemmas to be quantified over all non-negative integer clock values and to have a new assumption specifying that the clock is sufficiently large to allow full evaluation of the form under consideration without causing a BREAK-OUT-OF-TIME in E. Under this assumption, whenever f^n returns without a guard violation, it returns a value about which we can reason as before. The guard for each non-recursive function is the same as the guard for f, and hence the guard descriptors are complete. A nice side effect of this approach is that we do not rely on a termination requirement on our function definitions.

The top-level proof, then, becomes an induction on the value of the clock. In its base case, the clock is 0, and the hypothesis guaranteeing no BREAK-OUT-OF-TIME is violated, establishing the goal trivially. In the inductive step, where we reason about a call to f^{n+1}, the clock allows enough time to acquire a value from f^n, whose signature is the one we assume correct by our inductive assumption. Then we can apply the containment argument as before to achieve our goal.

The formal presentation of this argument appears in the proof of Lemma TC-SIGNATURE-OK-1, which is subsidiary to the proof of TC-SIGNATURE-OK in Section 7.3.3.

Credit for the suggestion of this treatment of recursive functions goes to Matt Kaufmann.

7.3 Validation of Signatures by the Checker

The lemmas which we will prove about the highest level of the checker establish the following with regard to a call to the function being analyzed. When

1. the guard descriptors for the function are complete and the guards of all functions hereditarily in the call tree of the function are complete, i.e., when each guard is formulated as a conjunction of calls to recognizer functions on distinct formal parameters,
2. the inference system does not return a *GUARD-VIOLATION result, and
3. the clock is sufficiently large to allow evaluation of the function call in question to finish,

then we will show

1. the evaluation of the guard expression on the actual parameters will not cause a guard violation;
2. any actual parameters satisfying the guard expression will satisfy the guard descriptor generated by the checker; and
3. if each segment generated by the checker is contained in some segment generated by the inference tool, then for any values satisfying the guard expression, no guard violation will occur in the course of evaluating the function call, and some segment generated by the checker will characterize both the argument values and the value of the function applied to them.

From the last claim in particular, following the approach outlined in Section 7.2, we will surmise that the segments generated by the inference tool are a correct typing for the function. The segments generated by the inference tool are correct, if not optimal, in that they represent a superset of the tuples defining the function.

Our top-level correctness lemma, TC-SIGNATURE-OK captures the notions just stated.

Lemma TC-SIGNATURE-OK

```
 For any n-ary function foo, whose definition is of the form
  (defun foo (a1 .. an)
    (declare (xargs :guard guard-form))
    body)
  where guard-form is a conjunction of recognizer calls on distinct
  formal parameters,
```

```
 for any world of Lisp functions world, including at least the above
  definition of foo and the definitions of all the functions in the
  call tree of foo,
 for any list of function signatures fs, including signatures for
  at least all the functions in the call tree of foo, except foo
  itself,
 for any non-negative integer clock,
 when (tc-signature foo fs) successfully validates a signature
  for foo,

H1  (and (valid-fs fs world clock)
H2       (and (not (equal (guard (tc-signature foo fs))
                          *guard-violation))
              (not (equal (segments (tc-signature foo fs))
                          *guard-violation)))
H3       (tc-all-called-functions-complete guard-form fs)
H4       (tc-all-called-functions-complete+ body fs foo t) )
    =>
    (valid-fs (cons (tc-signature foo fs) fs) world clock)
```

```
Note: Since on recursive calls to foo, we need to know if foo is
complete, and since foo is not in fs, we introduce in H4 the function
tc-all-called-functions-complete+, which is like
tc-all-called-functions-complete except that when it encounters a call
to foo, it looks to the fourth argument to see if foo's guard is
complete.  Our stated assumption about guard-form establishes that
it is.
```

Our other top-level result, Lemma GUARD-COMPLETE states that if a function's guard is formulated as a conjunction of calls to recognizer functions on distinct parameters, if descriptors characterizing the actual parameters in a call to the function are contained in the function's guard descriptors, and if the clock is sufficient to allow full evaluation of the guard on those parameters, then the parameters will satisfy the real guard.

Lemma GUARD-COMPLETE

```
 Given a function of arity n with argument list (a_1 .. a_n),
 and guard expression of the form
  (and (R_a_1 a_1) .. (R_a_n a_n))
 denoting a conjunction of calls to recognizer functions on distinct
 formal parameters, and where the recognizer function R_a_k
 has the segment (rtd_k) -> $t,
 for any values arg_1 .. arg_n, descriptors argtd_1 .. argtd_n,
 type variable binding b, non-negative integer clock, and a world of
 Lisp functions including all those in the call tree of the guard
 expression,

  (and
H1  (valid-fs fs world clock)
H2  (I (argtd_1 .. argtd_n) (arg_1 .. arg_n) b)
H3  (contained-in-interface (*dlist argtd_1 .. argtd_n)
                            (*dlist rtd_1 .. rtd_n))
H4  (not (break-out-of-timep
           (E (and (R_a_1 a_1) .. (R_a_n a_n))
              ((a_1 . arg_1) .. (a_n . arg_n))
              world
              clock))) )
  =>
  (equal (E (and (R_a_1 a_1) .. (R_a_n a_n))
            ((a_1 . arg_1) .. (a_n . arg_n))
            world
            clock)
```

```
        t)

 Note: For the sake of uniformity in notation, let us say that
 there is one recognizer call for each parameter, where for
 parameters which are unrestricted in the guard expression we
 use a recognizer (DEFUN UNIVERSALP (X) T) whose segments
 are ((*universal) -> $t) and ((*empty) -> $nil).  In
 any real guard, any such recognizer call may be omitted from
 the guard without the loss of generality of this lemma.
```

This lemma is critical to the proof of Lemma TC-SIGNATURE-OK, but because it makes possible the guard verification method discussed in Section 6.4 and because it is the primary specification of the guard descriptor component of a signature, we present it as a top-level result.

An alternate formulation of Lemma GUARD-COMPLETE might have omitted the notions of containment and argument descriptors and dealt strictly with some argument values satisfying the guard descriptor. Thus, it would have been stated:

```
  (and
H1  (valid-fs fs world clock)
H2  (I (rtd1 .. rtdn) (arg1 .. argn) b)
H3  (not (break-out-of-timep
           (E (and (Ra1 a1) .. (Ran an))
              ((a1 . arg1) .. (an . argn))
              world
              clock))) )
  =>
  (equal (E (and (Ra1 a1) .. (Ran an))
            ((a1 . arg1) .. (an . argn))
            world
            clock)
         t)
```

This would have more succinctly presented the notion that to determine some values satisfy a guard, it is sufficient to determine that they satisfy the guard descriptor. But by bundling the lemma as we did, we made it more convenient for every use in our proof.

Next, we will give the proof for GUARD-COMPLETE, which reveals it to be a corollary of another lemma, RECOGNIZER-SEGMENTS-COMPLETE. Then we will state and prove Lemma TC-INFER-SIGNATURE-GUARD-OK, which will support the proof of TC-SIGNATURE-OK, to which we return at the end of this section.

7.3.1 The Proof of Lemma GUARD-COMPLETE

The proof of this lemma will rest on the proof of an important property about the signatures of recognizer functions. This is not surprising, since a guard descriptor is complete if the guard is a conjunction of recognizer function calls on distinct formal parameters.

Recognizer functions are important in the type system because very precise signatures can be constructed for them, and because calls to recognizer functions (see Section 5.6) typically appear where the information they yield is important to the type inference problem. Recognizers are the linchpin of guard verification in the type system. If the guard of a function is expressed as a conjunction of recognizer calls on distinct formal parameters, we claim the containment test performed by the checker on function calls is sufficient to guarantee satisfaction of the function guard.

The formal property of recognizers which will support this proof is given below in Lemma RECOGNIZER-SEGMENTS-COMPLETE. It is that, given a function which conforms to the various syntactic requirements for recognizers, whose signature has two segments, both variable-free, one with a result type of $T and the other with result type $NIL, where DUNIFY-DESCRIPTORS-INTERFACE applied to the argument components of the two segments produces *EMPTY, if some descriptor characterizing the argument to the function is contained in the argument descriptor for the $T segment, and if the clock is sufficient to allow full evaluation of the function called with that argument, then the result of that evaluation is T.

```
Lemma RECOGNIZER-SEGMENTS-COMPLETE

 Given a recognizer function R of the form
  (defun R (x)
    (declare (xargs :guard t))
    body)
 with the signature:
  Guard: (*universal)
  Segments: (((targ) -> $t)
             ((nilarg) -> $nil))
 For any list of function signatures fs including R, Lisp world world
 including R, Lisp value v, non-negative integer clock, descriptor
 arg-td, and type variable binding b,

  (and
H1  (valid-fs fs world clock)
H2  (I arg-td v b)
H3  (contained-in-interface arg-td targ)
H4  (not (break-out-of-timep (R^{world,clock}(v)))) )
   =>
   (equal (R^{world,clock}(v)) t)
```

The proof of this lemma appears in Appendix B.2.

With this lemma as a basis, the proof of Lemma GUARD-COMPLETE is straightforward.

```
Proof of Lemma GUARD-COMPLETE

We will do this chiefly by repeated application of Lemma
RECOGNIZER-SEGMENTS-COMPLETE, found just above.

By expanding the definition of I, H2 becomes

H2' (and (I argtd_1 arg_1 b)
         ..
         (I argtd_n arg_n b))

By the definition of contained-in-interface, since each of
rtd_1 .. rtd_n is variable-free (by the definition of recognizer
functions, Section 5.6), H3 expands to

    (contained-in (universalize-all-vars[32]
                    (*dlist argtd_1 .. argtd_n))
                  (*dlist rtd_1 .. rtd_n))

which by the definitions of contained-in and universalize-all-vars,
expands further to

H3' (and (contained-in (universalize-all-vars argtd_1) rtd_1)
```

[32] UNIVERSALIZE-ALL-VARS just transforms all variables in its argument to *UNIVERSAL.

```
        ..
        (contained-in (universalize-all-vars argtdn) rtdn)
```

Now we instantiate Lemma RECOGNIZER-SEGMENTS-COMPLETE for each i in 1..n, using fs = fs, world = world, clock = clock, b = b, R = R_{a_i}, arg-td = $argtd_i$, targ = rtd_i, and v = arg_i. H1 equals H1. H2' guarantees its H2. H4 guarantees its H4, since if there is enough time to evaluate the conjunction of all the recognizer calls, there is enough time to evaluate each one individually. For each i, we still must relieve the hypothesis

```
  (contained-in-interface argtdi rtdi)
```

Because each rtd_i is variable-free, by the same reasoning explained above, this expands to

```
  (contained-in (universalize-all-vars arg-tdi) rtdi)),
```

which we know from H3'. Having relieved all the hypotheses, we can use the conclusion for each instantiation of the lemma, collected as:

$$\begin{array}{ll} \texttt{H5} & \texttt{(and (equal (}R_{a_1}{}^{world,clock}\texttt{(}arg_1\texttt{)) t)} \\ & \quad .. \\ & \quad \texttt{(equal (}R_{a_n}{}^{world,clock}\texttt{(}arg_n\texttt{)) t))} \end{array}$$

Since H4 allows enough clock time for the full evaluation of the form in the conclusion, expanding the evaluator E in the conclusion gives

$$\begin{array}{l} \texttt{(if (not (equal (}R_{a_1}{}^{world,clock}\texttt{(}arg_1\texttt{)) nil))} \\ \quad \texttt{(if (not (equal (}R_{a_2}{}^{world,clock}\texttt{(}arg_2\texttt{)) nil))} \\ \quad\quad .. \\ \quad\quad \texttt{(}R_{a_n}{}^{world,clock}\texttt{(}arg_n\texttt{)))} \\ \quad \texttt{nil)} \end{array}$$

which, using the equalities in H5, reduces to t. QED.

7.3.2 Validation of the Guard Descriptor

We claim that any values which satisfy the guard expression also satisfy the guard descriptor. This is significant in our proof that the segments generated by TC-INFER-SIGNATURE are correct, because the first step in the generation of those segments is the formulation of the type alists which will characterize the arguments. The concrete alist is extracted from the guard descriptor, which is itself extracted from the segments computed from the guard expression by TC-INFER. The guard descriptor is also the one returned by TC-INFER-SIGNATURE.

The lemma which captures this notion, TC-INFER-SIGNATURE-GUARD-OK, says that if TC-INFER-SIGNATURE does not return *GUARD-VIOLATION as its result, then for any binding environment of the formal variables, if there is sufficient clock for the guard to evaluate fully in that environment, and if the guard form evaluates to a non-NIL value, then that value does not signify a guard violation, and there exists some binding of type variables b such that running INTERP-SIMPLE with the vector of guard descriptors produced by TC-INFER-SIGNATURE and the actual parameters produces T.[33] (Here, as in

[33] In fact, the binding is immaterial, since no variables appear in the guard descriptors, but keeping the notion around allows seamless discourse in terms of INTERP-SIMPLE.

all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.)

```
Lemma TC-INFER-SIGNATURE-GUARD-OK

For any n-ary function foo, whose definition is denoted
 (defun foo (a1 .. an)
   (declare (xargs :guard guard-form))
   body),
for any Lisp world including foo and all functions hereditarily
 in the call tree of guard-form,
for any list of function signatures fs including all functions
 hereditarily in the call tree of guard-form (but not necessarily
 foo itself),
for any Lisp variable binding environment env of the form
 ((a1 . arg1) .. (an . argn)),
for any non-negative integer clock,

where we denote that if tc-infer-signature finds no guard
 violation, it returns a signature such that:
 (guard (tc-infer-signature foo (cons (infer-signature foo fs) fs)))
   = (gtd1 .. gtdn),

H1 (and (valid-fs fs world clock)
H2      (not (equal (guard
                      (tc-infer-signature
                        foo (cons (infer-signature foo fs) fs)))
                    *guard-violation))
H3      (tc-all-called-functions-complete guard-form fs)
H4      (not (break-out-of-timep (E guard-form env world clock)))
H5      (not (null (E guard-form env world clock))) )
   =>
   (and (not (break-guard-violationp (E guard-form env world clock)))
        For some binding b, (I (gtd1 .. gtdn) (arg1 .. argn) b) )

Note:
H1 establishes that fs is valid for the given clock.
H2 establishes that no guard violations are detected in the
   course of analyzing the guard.
H3 establishes that the guard descriptors for all the functions in the
   call tree of the guard are complete.
H4 establishes that the evaluation of the guard form under env terminates
   without exhausting the clock.
H5 establishes that the evaluation of the guard form under env returns
   a non-nil value.

In the conclusion, any binding will do, since (gtd1 .. gtdn)
contains no variables.
```

In fact, we will not need H5 to establish the first conjunct of this goal, but there is no harm in stating the lemma this way. The proof of this lemma utilizes two other significant lemmas, TC-INFER-OK and TC-PREPASS-OK, the latter of which is stated when its application arises in the proof. TC-INFER-OK is the soundness specification for TC-INFER, the recursive function which is the heart of the checker algorithm. TC-INFER is discussed at length in Section 6.5. Lemma TC-INFER-OK appears and is discussed further in Section 7.5 below, and its detailed proof is given in Appendix B.4. TC-PREPASS is the pre-processor for Lisp forms which normalizes IF expressions so that the test components always return Boolean values. It is discussed at length in Section 6.3. The detailed proof of Lemma TC-PREPASS-OK appears in Appendix B.1.

```
Proof of Lemma TC-INFER-SIGNATURE-GUARD-OK

 By definition, tc-infer-signature invokes
```

```
  (tc-infer (tc-prepass guard-form fs)
            ((a1 . &1) .. (an . &n))
            ((a1 . *universal) .. (an . *universal))
            fs)
 where &1 .. &n are distinct variables.

 Instantiate Lemma TC-INFER-OK with
  fs = fs,
  form = (tc-prepass guard-form fs),
  env = ((a1 . arg1) .. (an . argn)), and hence
   (arg1 .. argn) = (arg1 .. argn),
  abs-alist = ((a1 . &1) .. (an . &n)) and hence
   (tda1 .. tdan) = (&1 .. &n),
  conc-alist = ((a1 . *universal) .. (an . *universal)) and hence
   (tdc1 .. tdcn) = (*universal .. *universal),
  b = ((&1 . arg1) .. (&n . argn)),

 and also denote:
  (tc-infer (tc-prepass guard-form fs)
            ((a1 . &1) .. (an . &n))
            ((a1 . *universal) .. (an . *universal))
            fs)
 by:
  ((minseg1 conc-alist1 (maxtd1,1 .. maxtd1,n -> maxtd1))
   ..
   (minsegm conc-alistm (maxtdm,1 .. maxtdm,n -> maxtdm)))

 We will henceforth ignore the minimal segments and the concrete
 alists returned with each tuple returned by TC-INFER-OK,
 as they are irrelevant here.

 To use the conclusion of TC-INFER-OK, we need to relieve its
 hypotheses.  H1 is equal to H1'. (By H1' we mean the H1 from
 TC-INFER-OK.)  H2' is trivially true under the binding
 ((&1 . arg1) .. (&n . argn)).  H3' is trivially true.
 H2 guarantees H4'.  H3 and the fact that tc-prepass preserves
 tc-all-called-functions-complete (since the only function call
 it can add is null, whose guard descriptor is complete) guarantees
 H5'.  H4 in conjunction with tc-prepass-ok guarantees H6'.  Thus,
 we can use its conclusion.  Discarding the first conclusion about
 the minsegs and the third conclusion about the conc-alist for the
 result tuples, we have established:

H6 (and (not (break-guard-violationp
               (E (tc-prepass guard-form fs) env world clock)))
        for some i in 1..m,
         for some binding bi covering the descriptors below,
          (and
            (I (maxtdi,1 .. maxtdi,n tdi)
               (arg1 .. argn
                (E (tc-prepass guard-form fs) env world clock))
               bi)
          (extends-binding bi b)) )

 Now consider:

Lemma TC-PREPASS-OK
 For any list of function signatures fs, Lisp world world,
 non-negative integer clock, Lisp form form, and binding
 environment env,

  (valid-fs fs world clock)
  =>
  (equal (E form env world clock)
         (E (tc-prepass-form form fs) env world clock))
```

This lemma is proved in Appendix B.1.

**TC-PREPASS-OK allows us to substitute the value of
(E guard-form env world clock) for
(E (tc-prepass guard-form fs) env world clock), giving us**

**H6' (and (not (break-guard-violationp
(E guard-form env world clock)))
for some i in 1..m,
for some binding b_i covering the descriptors below,
(and
(I (maxtd$_{i,1}$ .. maxtd$_{i,n}$ td$_i$)
(arg$_1$ .. arg$_n$ (E guard-form env world clock))
b_i)
(extends-binding b_i b)))**

**The first conjunct of H6 establishes the first conjunct of
our goal. Note that we did not use H5 to establish this goal.**

**TC-INFER-SIGNATURE formulates its guard descriptor (gtd$_1$ .. gtd$_n$) by
first screening from
((maxtd$_{1,1}$ .. maxtd$_{1,n}$ td$_1$) .. (maxtd$_{1,1}$ .. maxtd$_{1,n}$ td$_1$))
all the segments where td$_i$ = $nil, i.e., thus preserving only
those segments where evaluating the guard expression can yield a
non-nil value. This is justified by H5, which says that the value of
the guard expression is non-nil under our binding environment env.
Thus, for any segment where tdi is $nil, the interpreter would yield
nil.**

**From the remaining segments it formulates the (gtd$_1$ .. gtd$_n$) by
combining the max segments "columnwise", using *or, giving:**

((*or maxtd$_{1,1}$ .. maxtd$_{m,1}$) .. (*or maxtd$_{1,n}$ .. maxtd$_{m,n}$))

This is conservative, i.e.,

**for some i in 1..m,
for some binding b_i covering the descriptors below,
(I (maxtd$_{i,1}$ .. maxtd$_{i,n}$ td$_i$)
(arg$_1$ .. arg$_n$ (E guard-form env world clock))
b_i)
=>
for some binding b' covering the descriptors below,
(I ((*or maxtd$_{1,1}$ .. maxtd$_{m,1}$) .. (*or maxtd$_{1,n}$ .. maxtd$_{m,n}$))
(arg$_1$ .. arg$_n$)
b')**

**where the b' is some b_i which sufficed for H6'. (To be
completely rigorous, consider that b' is extended as necessary to
bind all the variables in
((*or maxtd$_{1,1}$ .. maxtd$_{m,1}$) .. (*or maxtd$_{1,n}$ .. maxtd$_{m,n}$)),
but the right hand sides of the new bindings can be arbitrary,
since the bindings from the chosen b_i are sufficient to
establish our conclusion.) This implication holds, because for
whichever i and b_i
(I (maxtd$_{i,1}$ .. maxtd$_{i,n}$ td$_i$)
(arg$_1$ .. arg$_n$ (E guard-form env world clock))
b_i)
is true, the corresponding disjuncts in the expansion of I on
the combined form will also yield true.
So our hypothesis is now:**

**for some binding b' covering the descriptors below,
(and**

```
(I ((*or maxtd1,1 .. maxtdm,1) .. (*or maxtd1,n .. maxtdm,n))
   (arg1 .. argn)
   b')
(extends-binding b' b))
```

```
The (*or maxtd1,1 .. maxtdm,1) .. (*or maxtd1,n .. maxtdm,n)
are almost the (gtd1 .. gtdn) produced by tc-infer-signature.
The only remaining transformations are that singleton variables
are transformed to *universal, which is conservative (since for
any variable &i and value v, (I &i v b) => (I *universal v b)),
and the resulting descriptors are canonicalized, which by Lemma
PCANONICALIZE-DESCRIPTOR-OK also has no effect on the result
returned by I.  QED.
```

7.3.3 The Proof of Lemma TC-SIGNATURE-OK

Recall the discussion in Section 7.2, in which we noted that the graph of a recursive function can be represented as the union of the graphs of an infinite number of non-recursive functions, with the first being the empty function, the second being the ordered pairs formed by cases where the function terminates without recursion, the next being cases where it terminates with at most one recursion, and so forth. For a given function f, we called these non-recursive functions $\perp$, f^1, f^2, .. f^n. Each function definition f^i is non-recursive in the sense that it is formed by replacing the recursive calls of f with calls to f^{i-1}. Since each of these functions is partial, yet retains the guard of f to satisfy the guard completeness requirement, we adopted the clock semantics to support reasoning about them in the proof of Lemma TC-SIGNATURE-OK and its main supporting lemma, TC-SIGNATURE-OK-1.

Given Lemma TC-SIGNATURE-OK-1, the proof of Lemma TC-SIGNATURE-OK is trivial. It simply introduces as hypotheses some facts which TC-SIGNATURE establishes by direct computation, and then invokes Lemma TC-SIGNATURE-OK-1 to establish the conclusion. The proof of TC-SIGNATURE-OK-1 is the core of the argument. It establishes by induction on the clock that using a database of valid function signatures augmented with a signature for the function f in question which is shown to be valid for some clock value c, TC-INFER-SIGNATURE produces a signature which is valid for clock value c + 1. These signatures are what we are referring to as the signatures for f^n and f^{n+1} in the previous discussion of the top level proof of soundness. Then, we show that if the latter signature has the proper containment relation to the former, we are assured that the former is a valid signature for f. Specifically, this means that for any argument values which satisfy the guard of f, if the clock is sufficient to allow full evaluation of f applied to those arguments, we guarantee that the evaluation will not result in a guard violation, and there is some segment in the signature for f for which there exists a binding of type variables such that the argument values satisfy the left hand side of the segment and the result of f applied to those values satisfies the right hand side, or result type in the segment.

We will first give the simple proof for Lemma TC-SIGNATURE-OK, then present the statement and proof of Lemma TC-SIGNATURE-OK-1.

```
Proof of Lemma TC-SIGNATURE-OK

By definition, TC-SIGNATURE receives from the heuristic function
(infer-signature foo fs) the segments

 ((td1,1 .. td1,n td1) .. (tdm,1 .. tdm,n tdm))

Then it calls TC-INFER-SIGNATURE with fs augmented with the
segments above, and unless TC-INFER-SIGNATURE returns a
*guard-violation on either the guard form or the body, it
obtains the guard descriptor (gtd1 .. gtdn)
```

and the segments

((tc-td$_{1,1}$ .. tc-td$_{1,n}$ tc-td$_1$) .. (tc-td$_{l,1}$ .. tc-td$_{l,n}$ tc-td$_l$))

It then explicitly establishes by computation:

H3 (tc-all-called-functions-complete guard-form fs)
H4 (tc-all-called-functions-complete+ body fs foo t)
H5 (well-formed-signature-1
(gtd$_1$ .. gtd$_n$)
((tc-td$_{1,1}$ .. tc-td$_{1,n}$ tc-td$_1$)
..
(tc-td$_{l,1}$ .. tc-td$_{l,n}$ tc-td$_l$))
n)
H6 for all i in 1..l,
for some j in 1..m,
(contained-in-interface (*dlist tc-td$_{i,1}$ .. tc-td$_{i,n}$ tc-td$_i$)
(*dlist td$_{j,1}$ .. td$_{j,n}$ td$_j$))

H5 is required by the good-signaturep predicate for foo.
(valid-fs fs world clock) establishes the goal for all the functions represented in fs. As for foo, the definition of the valid-fs predicate requires that, for any Lisp values arg$_1$ .. arg$_n$,

(and
(not (break-out-of-timep (foo^{world,clock}(arg$_1$ .. arg$_n$))))
(not (null
(E guard-form ((a$_1$. arg$_1$) .. (a$_n$. arg$_n$)) world clock))))
=>
(and
(not (break-guard-violationp (foo^{world,clock}(arg$_1$ .. arg$_n$))))
for some k in [1..m]
for some binding b of type variables to Lisp values
covering td$_{k,1}$ .. td$_{k,n}$ and td$_k$
(I (td$_{k,1}$ .. td$_{k,n}$ td$_k$)
(arg$_1$ .. arg$_n$ (foo^{world,clock}(arg$_1$ .. arg$_n$)))
b))

Let two antecedents to this implication become H7 and H8.

Now instantiate Lemma TC-SIGNATURE-OK-1 (shown just below), using fndef = foo, clock = clock, world = world, fs = fs, and for each i, arg$_i$ = arg$_i$. H1 and H3 - H8 are identical to H1' and H3' - H8'. (By Hi', we denote Hi in TC-SIGNATURE-OK-1.) H2 guarantees H2', since if H2' were not true, by definition of TC-SIGNATURE, H2 would not be true. Thus, we obtain the conclusion of the lemma, which is equal to our goal.
QED.

Lemma TC-SIGNATURE-OK-1

For any n-ary function foo, whose definition is of the form
(defun foo (a$_1$ .. a$_n$)
(declare (xargs :guard guard-form))
body)
in which guard-form is a conjunction of recognizer calls on distinct formal parameters,
where (segments (infer-signature foo fs)) are denoted
((td$_{1,1}$ .. td$_{1,n}$ td$_1$) .. (td$_{m,1}$ .. td$_{m,n}$ td$_m$))
where (guard (tc-infer-signature foo (cons (infer-signature foo fs) fs))
is denoted (gtd$_1$ .. gtd$_n$)
and (segments
(tc-infer-signature foo (cons (infer-signature foo fs) fs)))

```
 are denoted
  ((tc-td$_{1,1}$ .. tc-td$_{1,n}$ tc-td$_1$) .. (tc-td$_{l,1}$ .. tc-td$_{l,n}$ tc-td$_l$))

for any world of Lisp functions world, including at least foo and
 all the functions in the call tree of foo,
for any list of function signatures fs, including signatures for
 at least all the functions in the call tree of foo, except foo
 itself,
for any non-negative integer clock, for any Lisp values arg$_1$ .. arg$_n$,

  (and
H1  (valid-fs fs world clock)
H2  (and (not (equal (guard
                      (tc-infer-signature
                        foo (cons (infer-signature foo fs) fs)))
                     *guard-violation))
         (not (equal (segments
                      (tc-infer-signature
                        foo (cons (infer-signature foo fs) fs)))
                     *guard-violation)))
H3  (tc-all-called-functions-complete guard-form fs)
H4  (tc-all-called-functions-complete+ body fs foo t)
H5  for all i in [1..l],
      for some j in [1..m],
        (contained-in-interface (*dlist tc-td$_{i,1}$ .. tc-td$_{i,n}$ tc-td$_i$)
                                (*dlist td$_{j,1}$ .. td$_{j,n}$ td$_j$))
H6  (well-formed-signature (gtd$_1$ .. gtd$_n$)
                           ((tc-td$_{1,1}$ .. tc-td$_{1,n}$ tc-td$_1$)
                            ..
                            (tc-td$_{l,1}$ .. tc-td$_{l,n}$ tc-td$_l$))
                           n)
H7  (not (break-out-of-timep (foo$^{world,clock}$(arg$_1$ .. arg$_n$))))
H8  (not (null (E guard-form ((a$_1$ . arg$_1$) .. (a$_n$ . arg$_n$)) world clock))))
  =>
  (and
C1  (not (break-guard-violationp (foo$^{world,clock}$(arg$_1$ .. arg$_n$))))
C2  for some k in [1..m]
      for some binding b of type variables to Lisp values
          covering td$_{k,1}$ .. td$_{k,n}$ and td$_k$
        (I (td$_{k,1}$ .. td$_{k,n}$ td$_k$)
           (arg$_1$ .. arg$_n$ (foo$^{world,clock}$(arg$_1$ .. arg$_n$)))
           b) )
```

The CONTAINED-IN-INTERFACE function is specified as follows:

```
Lemma CONTAINED-IN-INTERFACE-OK
 For any descriptors td1 and td2, value v, and binding b,

  (and (contained-in-interface td1 td2)
       (I td1 v b))
  =>
  For some b', (I td2 v b')
```

The containment algorithm is discussed at length in Section 6.8, and the proof of Lemma CONTAINED-IN-INTERFACE-OK is presented in Section 7.8. We will use this lemma, Lemma TC-INFER-SIGNATURE-GUARD-OK, and Lemma TC-INFER-OK in the proof of TC-SIGNATURE-OK-1.

Proof of Lemma TC-SIGNATURE-OK-1

We will prove this by induction on clock.

Base Case: clock = 0
This falsifies H7, since any call of E with clock < 1 returns a

break-out-of-time.

Inductive case:
Assuming the lemma for clock = c, we need to prove it holds for clock = c + 1.

Let us denote the Lisp variable binding environment
((a$_1$. arg$_1$) .. (a$_m$. arg$_m$))
by env.

By definition, tc-infer-signature first computes a guard descriptor for the function. Denote this descriptor (gtd$_1$ .. gtd$_n$). Instantiate Lemma TC-INFER-SIGNATURE-GUARD-OK, with foo = foo, fs = fs, world = world, clock = c + 1, and env = env. We need to relieve its hypotheses to use its conclusion. H1 is identical with H1'. (By Hi' we mean the Hi from TC-INFER-SIGNATURE-GUARD-OK.) H2' is the first conjunct of H2. H3 is identical with H3'. H7 establishes H4' by the following argument. The first step of (foo$^{world,c+1}$(arg$_1$ .. arg$_n$)) is to evaluate (E guard-form env world c+1). By definition of E, if (E guard-form env world c+1) returns an out-of-time break, then (foo$^{world,c+1}$(arg$_1$ .. arg$_n$)) does, too. But H7 establishes this does not happen, thus establishing H4'. H8 is equal to H5'. So we can use the conclusion of TC-INFER-SIGNATURE-GUARD-OK to obtain:

H9 (and
     (not (break-guard-violationp (E guard-form env world c+1)))
     For some binding b, (I (gtd$_1$ .. gtd$_n$) (arg$_1$ .. arg$_n$) b))

(The choice of binding b is immaterial, since gtd$_1$ .. gtd$_n$ are variable-free.)

Our inductive hypothesis is the lemma itself, with clock = c. We would like to use it to obtain the valid-fs predicate as it applies to foo, i.e., we would like to use the inductive hypothesis to establish (valid-fs fs' world c), where fs' is fs augmented with a signature with which we could reason about recursive calls to foo. The guard for this signature is (gtd$_1$ .. gtd$_n$), and the segments we want to use are
((td$_{1,1}$ .. td$_{1,n}$ td$_1$) .. (td$_{m,1}$ .. td$_{m,n}$ td$_m$)),
The predicate we need to establish in order for these segments to be valid on recursive calls to foo in body appears in the conclusion of the inductive hypothesis.

We need to relieve some of the antecedents of the inductive hypothesis. Our assumptions for the clock = c + 1 case will let us do that. Denote each hypothesis Hi of the inductive assumption as Hi'. First consider the following lemma:

Lemma VALID-FS-CLOCK

For any list of function signatures fs, world world, and non-negative integer clock,

```
(valid-fs fs world clock)
=>
(valid-fs fs world clock-1)
```

This lemma is proved in Appendix B.3. By VALID-FS-CLOCK, H1 implies H1'. H2 - H6 are identical to H2' - H6'. This reduces the inductive assumption to:

H10 (and

```
    (not (break-out-of-timep (foo^{world,c}(arg_1 .. arg_n))))
    (not (null (E guard-form ((a_1 . arg_1) .. (a_n . arg_n)) world c))))
   =>
   (and
     (not (break-guard-violationp (foo^{world,c}(arg_1 .. arg_n))))
     for some k in [1..m]
      for some binding b of type variables to Lisp values
        covering td_{k,1} .. td_{k,n} and td_k
       (I (td_{k,1} .. td_{k,n} td_k)
          (arg_1 .. arg_n (foo^{world,c}(arg_1 .. arg_n)))
          b) )
```

Combined with H6, we have established the good-signaturep predicate for (infer-signature foo fs) with clock c.

```
(good-signaturep foo
                 (gtd_1 .. gtd_n)
                 ((td_{1,1} .. td_{1,n} td_1)
                  ..
                  (td_{m,1} .. td_{m,n} td_m))
                 world
                 c)
```

Thus, we have established

H11 (valid-fs (cons ((gtd$_1$ .. gtd$_n$) (infer-signature foo fs)) fs) c).

By definition, tc-infer-signature generates the segments
((tc-td$_{1,1}$ .. tc-td$_{1,n}$ tc-td$_1$) .. (tc-td$_{l,1}$ .. tc-td$_{l,n}$ tc-td$_l$))
by invoking

```
 (tc-infer
   (tc-prepass
     body (cons ((gtd_1 .. gtd_n) (infer-signature foo fs)) fs))
   ((a_1 . &1) .. (a_n . &n))
   ((a_1 . (if gtd_1 = *universal then &1 else gtd_1))
    ..
    (a_n . (if gtd_n = *universal then &n else gtd_n)))
   (cons ((gtd_1 .. gtd_n) (infer-signature foo fs)) fs))
```

where &1 .. &n are new variables (i.e., variables not already used in the analysis of foo; we choose these names arbitrarily for the sake of exposition). So instantiate Lemma TC-INFER-OK, with form = the first argument to TC-INFER, abs-alist = the second, conc-alist = the third, fs = the fourth argument, world = world, clock = c, env = ((a$_1$. arg$_1$) .. (a$_n$. arg$_n$)), and b = ((&1 . arg$_1$) .. (&n . arg$_n$)). We relieve its hypotheses. H11 is equal to H1’. H2’ is trivially true under the binding b by simple expansion of I. H3’ follows directly by observing H2’ and the second conjunct of H9, and expanding I. H4’ is a conjunct of H2. H5’ is an immediate consequence of H4. To establish H6’, expand the call to E in H7:

```
(not (break-out-of-timep (foo^{world,c+1}(arg_1 .. arg_n))))
=
(not (break-out-of-timep
       (if (break-out-of-timep (E guard-form env world c+1))
           (E guard-form env world c+1)
           (if (break-guard-violationp (E guard-form env world c+1))
               (E guard-form env world c+1)
               (if (not (null (E guard-form env world c+1)))
                   (E body env world c)
                   (break-guard-violation))))))
```

If (break-out-of-timep (E guard-form env world c+1)) were true, H7 would simplify to false, so it must not be true. If (break-guard-violationp (E guard-form env world c+1)) were true, it would contradict the first conjunct of H9. H8 establishes (not (null (E guard-form env world c+1))). Thus, we have established

H12 (equal (foo$^{world,c+1}$**(arg**$_1$ **.. arg**$_n$**))**
(E body env world c))

and reduced the above form to:

(not (break-out-of-timep (E body env world c)))

which is precisely H6', thus relieving the final hypothesis. So we establish the conclusion of the lemma, and we ignore the conjuncts regarding the mintd's and tdconc's. By definition of tc-infer-signature, the maxtd's are the ones which are returned as the segments

((tc-td$_{1,1}$ **.. tc-td**$_{1,n}$ **tc-td**$_1$**) .. (tc-td**$_{l,1}$ **.. tc-td**$_{l,n}$ **tc-td**$_l$**))**

Thus, the conclusion of the instantiated Lemma TC-INFER-OK gives us

(and
(not (break-guard-violationp
(E (tc-prepass
body (cons ((gtd$_1$ **.. gtd**$_n$**) (infer-signature foo fs)) fs))**
env world c)))
for some i in 1..l**,**
for some binding b covering the descriptors below,
(and
(I (tc-td$_{i,1}$ **.. tc-td**$_{i,n}$ **tc-td**$_i$**)**
(arg$_1$ **.. arg**$_n$
(E (tc-prepass
body (cons ((gtd$_1$ **.. gtd**$_n$**) (infer-signature foo fs)) fs))**
env world c))
b)
(extends-binding b ((&1 . arg$_1$**) .. (&n . arg**$_n$**)))))**

Now use Lemma TC-PREPASS-OK, instantiated with clock = c, env = env, fs = (cons ((gtd$_1$ **.. gtd**$_n$**) (infer-signature foo fs)) fs), world = world, and form = body. H11 relieves its hypothesis, and the conclusion gives us an equality which allows us to transform the form above to**

(and
(not (break-guard-violationp (E body env world c)))
for some i in 1..l**,**
for some binding b covering the descriptors below,
(and (I (tc-td$_{i,1}$ **.. tc-td**$_{i,n}$ **tc-td**$_i$**)**
(arg$_1$ **.. arg**$_n$ **(E body env world c))**
b)
(extends-binding b ((&1 . arg$_1$**) .. (&n . arg**$_n$**)))))**

We do not need the (extends-binding b ((&1 . arg$_1$**) .. (&n . arg**$_n$**))) result, so henceforth we disregard it. Finally, use the equality substitution from H12 to get**

H13 (and
(not (break-guard-violationp
(foo$^{world,c+1}$**(arg**$_1$ **.. arg**$_n$**))))**
for some i in 1..l**,**
for some binding b,
(I (tc-td$_{i,1}$ **.. tc-td**$_{i,n}$ **tc-td**$_i$**)**
(arg$_1$ **.. arg**$_n$ **(foo**$^{world,c+1}$**(arg**$_1$ **.. arg**$_n$**)))**

```
           b) )

The first conjunct of H13 establishes our goal C1.

Now we use our containment hypothesis, H5.  Consider any i and b
which satisfy the second conjunct of H13.  H5 ensures that for
this i,

 for some j in [1..m]
  (contained-in-interface (*dlist tc-td_{i,1} .. tc-td_{i,n} tc-td_i)
                          (*dlist td_{j,1} .. td_{j,n} td_j))

Select the j which satisfies this condition and instantiate
Lemma CONTAINED-IN-INTERFACE-OK with
td1 = (*dlist tc-td_{i,1} .. tc-td_{i,n} tc-td_i),
td2 = (*dlist td_{j,1} .. td_{j,n} td_j),
v = (arg_1 .. arg_m (foo^{world,c+1}(arg_1 .. arg_n))),
and b = b.  H5 for this i and j satisfies the first hypothesis
of the lemma, and the second conjunct of H13 for our particular
i satisfies the second hypothesis.  This yields the conclusion,

 for some b',
  (I (td_{j,1} .. td_{j,n} td_j)
     (arg_1 .. arg_m (foo^{world,c+1}(arg_1 .. arg_n)))
     b')

This establishes the second conjunct of our conclusion.  QED.
```

Just to be complete, we make the following observation about the use of type variables. In the proof of TC-INFER-OK below, and implicitly wherever we use this lemma (as we did in the previous proof), we use the fact that whenever we utilize the VALID-FS predicate to obtain the segments for some function, those segments are imported into the checker's problem state with all fresh type variables. We claim that these variables have never been introduced before and will never be introduced again. As a matter of programming efficacy, the inference tool in fact does "reset" the counter on its variable generator whenever a new function is introduced for analysis. This does not cause a problem, because no variables from any previous problem state appear in the new problem state unless they are renamed on importation of signature segments. From a formal viewpoint, though, we can consider that a type variable name is actually a pair whose first component is the kind of name (&2, for instance) by which we have been referring to them all along, and the second component is the name of the function being analyzed. Thus, no variable is ever introduced twice as a "fresh" variable for the life of the operation of the system.

A similar trick is used to ensure that *REC descriptor labels are not duplicated. The labels are generated by a pseudo-gensym function which forms them from the characters "!REC" followed by a counter. Since *REC labels are not renamed when a signature is extracted from FS, this is not enough. So another string of characters, unique for each function being processed, is attached to the end of the *REC label, ensuring the uniqueness of labels for the life of the system operation.

7.3.4 Replacing the Guard

As mentioned in Section 6.3, if the guard descriptor computed by the inference tool is different from the one computed by the checker, the checker determines whether the inference tool descriptor may be used instead of its own. This is done after the checker has generated its segments, so in any case it uses its own guard descriptor for that purpose. The checker's guard descriptor may be replaced by the inference tool guard if the two are equivalent. This is determined by administering the containment test in both directions. If each is contained in the other, the two are equivalent under interpretation by INTERP-SIMPLE. The following lemma provides the formal justification.

```
Lemma GUARD-REPLACEMENT-OK

 For any descriptors td1_1.. td1_n and td2_1.. td2_n,
 Lisp values v_1.. v_n, and binding b,

  (and
H1  (null (gather-vars-in-descriptor (*dlist td1_1.. td1_n)))
H2  (null (gather-vars-in-descriptor (*dlist td2_1.. td2_n)))
H3  (contained-in-interface (*dlist td1_1.. td1_n)
                            (*dlist td2_1.. td2_n))
H4  (contained-in-interface (*dlist td2_1.. td2_n)
                            (*dlist td1_1.. td1_n))
  =>
  (equal (I (td1_1.. td1_n) (v_1.. v_n) b)
         (I (td2_1.. td2_n) (v_1.. v_n) b))

Proof of Lemma GUARD-REPLACEMENT-OK

 Instantiate Lemma CONTAINED-IN-INTERFACE-OK with v = (v_1.. v_n),
 td1 = (*dlist td1_1.. td1_n), td2 = (*dlist td2_1.. td2_n),
 and b = b.  Since the descriptors are variable-free, any binding
 will suffice, so we choose b in the conclusion, giving

H5  (I (td1_1.. td1_n) (v_1.. v_n) b)
    =>
    (I (td2_1.. td2_n) (v_1.. v_n) b)

 Instantiating with the descriptors reversed gives us

H6  (I (td2_1.. td2_n) (v_1.. v_n) b)
    =>
    (I (td1_1.. td1_n) (v_1.. v_n) b)

 Clearly, two hypotheses establish the conclusion, since if
 either call to I is true, the other is, and if either is false,
 the other must be, else its being true would enable a
 contradition to be trivially derived.
 QED
```

7.4 The Initial State

The following lemma, which we state without proof, establishes that fs, the initial state of function signatures, is correct for all the subrs it represents. The initial fs appears in Appendix A.

```
Lemma INITIAL-FS-VALID
 Where world is a world containing the initial set of primitive subrs,
 and where initial-function-signatures is the list of function
 signatures for the primitives,

 For any non-negative integer clock,
  (valid-fs initial-function-signatures world clock)
```

The purpose of all the proofs in the preceding section was to illustrate that if the system database is a VALID-FS initially, we obtain a VALID-FS by repeated applications of the tool. Lemma INITIAL-FS-VALID establishes the base case for this line of reasoning, and Lemma TC-SIGNATURE-OK provides the inductive step.

7.5 The Central Checker Algorithm -- TC-INFER

The recursive function which is at the heart of the checker is TC-INFER. TC-INFER takes four arguments, a Lisp FORM being analyzed, an abstract type alist ABS-ALIST, a concrete type alist CONC-ALIST, and the database of FUNCTION-SIGNATURES. It returns a list of 3-tuples, where each tuple is composed of a minimal segment, a concrete type alist, and a maximal segment. This function is discussed at length in Section 6.5. Its formal specification is given by Lemma TC-INFER-OK, which states that if FORM is evaluated in a context where the variables are bound to values which satisfy the descriptors in both ABS-ALIST and CONC-ALIST, if TC-INFER does not return *GUARD-VIOLATION, if all the guards in the call tree of form are complete, and if clock is sufficient to allow full evaluation of the form in a given environment, then no guard violation will occur in the course of the evaluation, and some tuple in the result is such that the variable values and the result of the evaluation satisfy both the minimal and maximal segments in the tuple, and the variable values satisfy the descriptors in the concrete alist. The type variable binding under which this is achieved is an extension (see Appendix B.3) of the one under which the context values satisfied the two alist parameters.

```
Lemma TC-INFER-OK

For any Lisp form form, function signature list fs, Lisp world world
  including all functions hereditarily in the call tree of form,
for any non-negative integer clock, type variable bindings b,
 Lisp values arg1 .. argm,
 Lisp variables a1 .. am,
 binding environment env of the form ((a1 . arg1) .. (am . argm))
  where a1 .. am include all the free variables in form,
 ABS-ALIST of the form ((a1 . tda1) .. (am . tdam)),
 CONC-ALIST of the form ((a1 . tdc1) .. (am . tdcm)),
and denoting
  (tc-infer form abs-alist conc-alist fs)
 by
  (((mintd1,1 .. mintd1,m mintd1)
    ((a1 . tdconc1,1) .. (am . tdconc1,m))
    (maxtd1,1 .. maxtd1,m maxtd1))
   ..
   ((mintdn,1 .. mintdn,m mintdn)
    ((a1 . tdconcn,1) .. (am . tdconcn,m))
    (maxtdn,1 .. maxtdn,m maxtdn)))

H1 (and (valid-fs fs world clock)
H2      (I (tda1 .. tdam) (arg1 .. argm) b)
H3      (I (tdc1 .. tdcm) (arg1 .. argm) b)
H4      (not (equal (tc-infer form abs-alist conc-alist fs)
                    *guard-violation))
H5      (tc-all-called-functions-complete form fs)
H6      (not (break-out-of-timep (E form env world clock))) )
   =>
   (and
C1   (not (break-guard-violationp (E form env world clock)))
C2   for some i,
      for some binding b' covering the descriptors below,
       (and (I (mintdi,1 .. mintdi,m mintdi)
               (arg1 .. argm (E form env world clock))
               b')
            (I (maxtdi,1 .. maxtdi,m maxtdi)
               (arg1 .. argm (E form env world clock))
               b')
            (I (tdconci,1 .. tdconci,m) (arg1 .. argm) b')
            (extends-binding b' b)) )

Note:
```

```
H1 establishes that the signatures in the system state fs are valid.
H2 establishes that the abs-arglist is valid.
H3 establishes that the conc-arglist is valid.
H4 establishes that no guard violations are detected in the
   course of analyzing form.
H5 establishes that the guards of all functions in the call tree
   of form are complete.
H6 establishes that the evaluation of form terminates without
   exhausting the clock.
```

The proof of the lemma is in Appendix B.4.

7.6 The Unifier -- DUNIFY-DESCRIPTORS

The top level conjecture for descriptor unification in the checker is characterized by the following lemma. Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK
 For any descriptors tda and tdb, Lisp value v and fully
 instantiating binding of type variables to Lisp values b,

 (and (I tda v b)
      (I tdb v b))
 =>
 (I (dunify-descriptors-interface tda tdb) v b)
```

The definition of DUNIFY-DESCRIPTORS-INTERFACE is:

```
(DEFUN DUNIFY-DESCRIPTORS-INTERFACE (DESCRIPTOR1 DESCRIPTOR2)
  ;; DESCRIPTOR1 and DESCRIPTOR2 should both be well-formed descriptors.
  (PCANONICALIZE-DESCRIPTOR
    (CONS '*OR
          (MAP-DAPPLY-SUBST-LIST
            (DUNIFY-DESCRIPTORS DESCRIPTOR1 DESCRIPTOR2 NIL NIL)))))

(DEFUN MAP-DAPPLY-SUBST-LIST (DUNIFY-RESULT)
  (IF (NULL DUNIFY-RESULT)
      NIL
      (CONS (DAPPLY-SUBST-LIST
              (DUNIFIED-FORM-SUBSTS (CAR DUNIFY-RESULT))
              (DUNIFIED-FORM-DESCRIPTOR (CAR DUNIFY-RESULT)))
            (MAP-DAPPLY-SUBST-LIST (CDR DUNIFY-RESULT)))))
```

DUNIFY-DESCRIPTORS-INTERFACE is a wrapper for the key recursive unification algorithm, DUNIFY-DESCRIPTORS, which it calls with its two arguments and an empty substitution list. DUNIFY-DESCRIPTORS returns a list of (descriptor . substs) pairs. Simply stated, unification represents an effort to find the common ground between the input descriptors, and each element of this result list characterizes some of that common ground. The descriptor part of the pair characterizes the result down to the point where type variables appear, and the substs further characterize the types of the objects represented by the variables. From this list of pairs, DUNIFY-DESCRIPTORS-INTERFACE constructs an *OR whose disjuncts are the results of applying each substitution to its associated descriptor. After canonicalizing, it returns this result.

So two lemmas, one characterizing the result of DUNIFY-DESCRIPTORS and the other for DAPPLY-SUBST-LIST, will provide the foundation for the proof of DUNIFY-DESCRIPTORS-INTERFACE-OK.

The lemmas are as follows.

```
Lemma DUNIFY-DESCRIPTORS-OK
 Denoting (dunify-descriptors tda tdb substs)
  by ((td1 . substs1) .. (tdn . substsn)),
 for all v and fully instantiating b,

  (and
H1  (interp-substs substs b)
H2  (I tda v b)
H3  (I tdb v b))
  =>
  for some i,
    (and (interp-substs substsi b)
         (I tdi v b))
```

Note: Recall that DUNIFY-DESCRIPTORS also takes a fourth argument, TERM-RECS, which is employed solely as a mechanism for terminating the computation in certain cases. As such, it has no effect on the soundness argument, which does not rely on termination, and to avoid clutter in the lemmas and proofs, we simply omit mention of it. One could also think of it as being a universally quantified variable in the lemma.

The INTERP-SUBSTS predicate characterizes the information which is carried by SUBSTS and by each SUBSTS$_i$. The substitution lists map variables to descriptors characterizing what we know about the type of the value corresponding to the variable. INTERP-SUBSTS guarantees that this information is consistent with the value bound to the variable in BINDINGS.

```
(DEFUN INTERP-SUBSTS (SUBSTS BINDINGS)
  ;; SUBSTS should be an alist whose keys are type variables and
  ;;  whose associated values are well-formed descriptors.
  ;; BINDINGS should be an alist associating type variables to
  ;;  well-formed descriptors.
  (IF (NULL SUBSTS)
      T
      (AND (INTERP-SIMPLE-1 (CDR (CAR SUBSTS))
                            (CDR (ASSOC (CAR (CAR SUBSTS)) BINDINGS))
                            BINDINGS)
           (INTERP-SUBSTS (CDR SUBSTS) BINDINGS))))
```

The proof of DUNIFY-DESCRIPTORS-OK appears in Appendix B.5.

Note that the reverse implication in DUNIFY-DESCRIPTORS-OK does not hold. Consider an example where the unifier loses information:

```
 (DUNIFY-DESCRIPTORS
   '(*REC FOO (*OR &1 (*CONS (*OR $INTEGER $T) (*RECUR FOO))))
   '(*REC BAR (*OR &1 (*CONS (*OR $INTEGER $NIL) (*RECUR BAR))))
    NIL)
  = (by *REC Rule 0')
 (((*REC BIM (*OR *UNIVERSAL (*CONS $INTEGER (*RECUR BIM)))) . NIL))
```

Neither of the conclusions, (INTERP-SUBSTS NIL B) nor

```
(INTERP-SIMPLE-1
  (*REC BIM (*OR *UNIVERSAL (*CONS $INTEGER (*RECUR BIM)))) V B)
```

carry the information necessary to relate the last tail of v to the binding of &1 in b.

The second lemma is DAPPLY-SUBST-LIST-OK.

```
Lemma DAPPLY-SUBST-LIST-OK
 For any well-formed substitution s, descriptor td, binding b,
 and value v, (where td can be a *dlist containing n descriptors
 iff v is a value list of length n, in which case the call to I
 is on the list of n descriptors)

 (and (interp-substs s b) (I td v b))
 =>
 (I (dapply-subst-list s td) v b)
```

Lemma DAPPLY-SUBST-LIST-OK says that if the substitution list S is consistent with the binding list B, as determined by INTERP-SUBSTS, and if the value V satisfies a descriptor TD with respect to B, then with respect to B, V also satisfies the descriptor formed by applying the substitution list S to the descriptor TD.

Lemma DAPPLY-SUBST-LIST-OK allows us to apply the substitution generated by DUNIFY-DESCRIPTORS to the accompanying descriptor result. In terms of our code, it means we can soundly call DUNIFY-DESCRIPTORS-INTERFACE rather than DUNIFY-DESCRIPTORS. The proof of DAPPLY-SUBST-LIST-OK appears in Appendix B.4.

Now we can return to the proof of DUNIFY-DESCRIPTORS-INTERFACE-OK, which employs Lemmas DAPPLY-SUBST-LIST-OK and DUNIFY-DESCRIPTORS-OK.

The proof of Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK is actually the "simultaneous" proof of a rather large collection of lemmas which are invoked in an inductive nest composing the whole. The grand induction schema is a computational induction on the length of the computation required to return a result from DUNIFY-DESCRIPTORS-INTERFACE. The induction, then, follows the structure of the computation defined by the functions of concern, which form a large, mutually recursive nest. The lemmas within this collection, whose proofs are "mutually inductive", include DUNIFY-DESCRIPTORS-INTERFACE-OK, DUNIFY-DESCRIPTORS-OK, DMERGE-OK, DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST-OK, DMERGE-SECOND-ORDER-OK, DMERGE-DUNIFIED-FORMS-INTO-SUBSTS-OK, and the various lemmas which are associated with the special case rules for unifying pairs of *REC descriptors. Each of these lemmas characterizes the result of some function (or invocation of a *REC rule). For any call of any of these functions subsidiary to the main call of DUNIFY-DESCRIPTORS-INTERFACE within the computation, we can invoke the associated lemma as an inductive hypothesis.

Note that since we are not demonstrating a proof of termination, this is a partial correctness proof, since we are not guaranteeing that the length of the computation is finite. (See the discussion in Section 6.6.)

Proof of Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK

```
 (dunify-descriptors-interface tda tdb) first calls
 (dunify-descriptors tda tdb nil).[34]  Let us denote the result
 returned from that call as
  ((td_1 . substs_1) .. (td_n . substs_n))
 So we would like to use Lemma DUNIFY-DESCRIPTORS-OK.
 (interp-substs nil b) is trivially true for any binding b,
 satisfying H1 of DUNIFY-DESCRIPTORS-OK, and its other hypotheses
 are guaranteed by hypotheses in our main conjecture.  So we use
 the conclusion of DUNIFY-DESCRIPTORS-OK in order to obtain:

H3  for some i in 1..n,
```

34 The fourth argument, TERM-RECS, is omitted, as previously mentioned.

(and (interp-substs substs$_i$ b)
(I td$_i$ v b))

Next, by definition dunify-descriptors-interface calls well-formed-substs on each substs$_i$. We split into cases, based on whether any substs$_i$ was found to be ill-formed.

Case 1. for some i, (not (well-formed-substs substs$_i$))

By definition dunify-descriptors-interface returns either tda or tdb, depending on which has the smaller representation. Either one trivially satisfies our conclusion, which is equated to the hypothesis corresponding to one or the other.

Case 2. for all i, (well-formed-substs substs$_i$)

Using Lemma DAPPLY-SUBST-LIST-OK on each (td$_i$. substs$_i$) pair where substs$_i$ is non-looping (discovery of a looping subst causes the tool to break) from the result of (dunify-descriptors tda tdb nil), we can transform H3 to

for some i, (I (dapply-subst-list substs$_i$ td$_i$) v b)

or, stated equivalently

H3 (or (I (dapply-subst-list substs$_1$ td$_1$) v b)
..
(I (dapply-subst-list substs$_n$ td$_n$) v b))

This is equal to the definition of I applied to v, b, and

(*or (dapply-subst-list substs$_1$ td$_1$)
..
(dapply-subst-list substs$_n$ td$_n$))

giving us

H3' (I (*or (dapply-subst-list substs$_1$ td$_1$)
..
(dapply-subst-list substs$_n$ td$_n$))
v
b)

By definition, dunify-descriptors-interface returns the descriptor

(pcanonicalize-descriptor
(*or (dapply-subst-list substs$_1$ td$_1$)
..
(dapply-subst-list substs$_n$ td$_n$)))

Thus our goal is

(I (pcanonicalize-descriptor
(*or (dapply-subst-list substs$_1$ td$_1$)
..
(dapply-subst-list substs$_n$ td$_n$)))
v b)

PCANONICALIZE-DESCRIPTOR-OK (See Section 7.7) is the lemma which says that canonicalization preserves interpretation under I. Applying it to H3, we satisfy the goal.

QED.

7.7 Descriptor Canonicalization

Eighteen canonicalization rules are implemented within PCANONICALIZE-DESCRIPTOR and its subsidiary functions. The rules are stated in the form "td1 ==> td2", where td1 is a generic representation of descriptors eligible for the canonicalization, and td2 is the representation of the form into which those descriptors are canonicalized. By "==>" we mean only that

```
(PCANONICALIZE-DESCRIPTOR TD1) = TD2
```

Although the canonicalization is directed, each rule happens to be an equality, in the sense that td1 and td2 represent the same set of values under any interpretation by INTERP-SIMPLE. All that is required for soundness of each rule "td1 ==> td2" is proof of an associated lemma of the form[35]:

```
For every value v and binding b,
 (I td1 v b)
 =>
 (I td2 v b)
```

Since it would be tedious to restate each rule in this form, we simply provide the rule notation, but the proof follows the INTERP-SIMPLE model. The rules are stated in Section 6.7, and the proofs are in Appendix B.6. Each rule is annotated with the name of the function in which it is implemented.

It will be convenient to have a top-level lemma to which to appeal when encountering canonicalization.

```
Lemma PCANONICALIZE-DESCRIPTOR-OK

 For any descriptor td, value v, and binding b,
  (where td is a *dlist containing n descriptors iff v is a
  value list of length n)

  (I td v b)
  =>
  (I (pcanonicalize-descriptor td) v b)


Proof of Lemma PCANONICALIZE-DESCRIPTOR-OK

 By definition,
 (PCANONICALIZE-DESCRIPTOR td) = (PREAL-CANONICALIZE-DESCRIPTOR td).
 By definition, PREAL-CANONICALIZE-DESCRIPTOR calls
 (PCANONICALIZE-DESCRIPTOR-1 td), checks to see if the
 result = td, and, if so, returns it.  If not, it calls itself
 recursively, by way of two individual canonicalizations,

 (PREAL-CANONICALIZE-DESCRIPTOR
   (PFOLD-RECS-IF-POSSIBLE
     (PCONSOLIDATE-UNIVERSAL-DESCRIPTORS result)))

 PCONSOLIDATE-UNIVERSAL-DESCRIPTORS and PFOLD-RECS-IF-POSSIBLE
 either do nothing or invoke canonicalization Rules 1 and 3,
 respectively, and by the lemmas for these rules, each function
 preserves (I td v b).

 PCANONICALIZE-DESCRIPTOR-1 calls PREAL-CANONICALIZE-DESCRIPTOR
 recursively on components of td, and otherwise invokes individual
 canonicalization rules as appropriate.

 Thus, with these mutually recursive functions, we present this proof
```

[35] Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
as a computational induction on the number of computational steps
in the evaluation of the top level call of the function under
consideration.  This is a partial correctness proof, in that we
are not proving termination.  Thus, we do not rule out the
possibility that n is infinitely large.

The inductive assumption is that for all calls to
PREAL-CANONICALIZE-DESCRIPTOR subsidiary to the top level call,
for all values v and bindings b,

 (I td v b)
 =>
 (I (PREAL-CANONICALIZE-DESCRIPTOR td) v b)

Case 1: (PCANONICALIZE-DESCRIPTOR-1 td) = td,

 Thus, PREAL-CANONICALIZE-DESCRIPTOR and PCANONICALIZE-DESCRIPTOR
 each return td, and thus we establish the goal trivially.

Case 2: (PCANONICALIZE-DESCRIPTOR-1 td) = td', where td ≠ td'

 Some number of computational steps have occurred.  For each
 such step, either no transformation was performed on td, in
 which case (I td v b) continues to hold, or the step performed
 a canonicalization by invoking some canonicalization rule.  For
 each such rule, we have a lemma which says (I td v b) implies
 (I canonicalized-td v b), where canonicalized-td is the result
 of the canonicalization.  By chaining these results, we see that

  (I td v b)
  =>
  (I (PCANONICALIZE-DESCRIPTOR-1 td) v b)

 By definition, PREAL-CANONICALIZE-DESCRIPTOR now calls
 PCONSOLIDATE-UNIVERSAL-DESCRIPTORS and PFOLD-RECS-IF-POSSIBLE
 in turn on the result.  If either function transforms td, its
 associated lemma ensures that (I td v b) is preserved on the
 result, and our counter is further reduced.  Then
 PREAL-CANONICALIZE-DESCRIPTOR calls itself recursively, and
 we invoke our inductive assumption to establish the goal.

 Note that PCANONICALIZE-DESCRIPTOR-1 may have called
 PREAL-CANONICALIZE-DESCRIPTOR recursively.  Any such recursive
 call reduces the computational count and allows the use of the
 inductive assumption to characterize the result.  QED.
```

7.8 The Containment Algorithm

Recall that our specification of containment is:

```
Lemma CONTAINED-IN-INTERFACE-OK
 For any descriptors td1 and td2, value v, and binding b,

  (and (contained-in-interface td1 td2)
       (I td1 v b))
  =>
  For some b', (I td2 v b')
```

The containment algorithm has the task of demonstrating that b' can be generated. If the algorithm cannot determine how to do so, it returns a NIL result.

In the special case where TD1 and TD2 are variable-free, the binding is irrelevant. This is the case handled by the subsidiary algorithm CONTAINED-IN. The task is the relatively simple one of determining whether a value's membership in the set characterized by TD1 guarantees its membership in the set corresponding to TD2.

As described in Section 6.8, if variables appear in the descriptors, the top level function CONTAINED-IN-INTERFACE attempts to transform the arguments to variable-free forms and to employ the CONTAINED-IN-INTERFACE algorithm. If it cannot do this, it employs the heuristic algorithm VCONTAINED-IN to produce a mapping which represents a method for producing a B' from any satisfactory B. Then it validates that the mapping will work by calling a verified routine MAPPINGS-DEMONSTRATE-CONTAINMENT, which manages the application of a checker algorithm ICONTAINED-IN. Essentially, MAPPINGS-DEMONSTRATE-CONTAINMENT raises disjunctions in TD1 to the top, as previously described, and then determines whether for each disjunct, the mapping provides a method for constructing a binding which will establish containment. If so, CONTAINED-IN-INTERFACE returns T, otherwise it returns NIL.

Let us leave for the moment further discussion of CONTAINED-IN-INTERFACE, except to note that it makes a decision as to whether the problem can be handled by the variable-free containment function CONTAINED-IN. We will state and prove lemmas about CONTAINED-IN and ICONTAINED-IN, then return to the proof of CONTAINED-IN-INTERFACE. Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

Our specification for the simpler, variable-free containment algorithm is:

```
Lemma CONTAINED-IN-OK

 For any descriptors td1 and td2, Lisp value v, and binding b1,

  (and
H1  (null (gather-variables-in-descriptor td1))
H2  (null (gather-variables-in-descriptor td2))
H3  (contained-in td1 td2)
H4  (I td1 v b1))
  =>
  For some b2, (I td2 v b2)

Comment: The b2 used here is irrelevant, since the descriptors are
variable-free.
```

Note: Recall that CONTAINED-IN also takes a third argument, TERM-RECS, which is employed solely as a mechanism for terminating the computation in certain cases. As such, it has no effect on the soundness argument, which does not rely on termination, and to avoid clutter in the lemmas and proofs, we simply omit mention of it. One could also think of TERM-RECS as being a universally quantified variable in the lemma. This lemma is proved in Appendix B.8.

For the case where we still have variables in our descriptors, VCONTAINED-IN is simply a heuristic function serving as a witness for ICONTAINED-IN. Supplied with TD1 and TD2, VCONTAINED-IN returns a mapping structure which is subsequently flattened into a list of simple mappings. (See the discussion of mapping structures in Section 6.8.4.) Using equivalence-preserving canonicalization rules, which have been proven correct, we lift *ORs in TD1. Furthermore, imagine a symbolic value reference VREF which represents, on entry, the value to which we refer as V in Lemma CONTAINED-IN-INTERFACE-OK. If for each disjunct $TD1_i$, there exists some simple mapping M_j from our list which can be applied to TD2 so that

(ICONTAINED-IN $td1_i$ (DAPPLY-SUBST-LIST m_j td2) vref)

holds, then we have shown containment, as we shall now prove, and CONTAINED-IN-INTERFACE returns true. To show the soundness of this approach, we need to prove the following lemma:

```
Lemma ALL-DESCRIPTOR1-DISJUNCTS-OK-OK
 For any descriptor td1 of the form (*or td1_1 .. td1_n)
 and descriptor td2, value reference vref, value v, and binding
 b1 covering the variables of td1,
  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  For all i, for some simple mapping m,
      (and (well-formed-mapping m vref b1)
           (icontained-in td1_i (apply-subst m td2) vref))
H3  (I td1 v b1) )
   =>
   For some b2, (I td2 v b2)
```

This proof, of course, will require a statement of correctness of ICONTAINED-IN, which follows. ICONTAINED-IN must be given a simple mapping which is well-formed in the context of the problem. The predicate which determines well-formedness is WFF-MAPPING. The WFF-MAPPING predicate on mappings is only strong enough to guarantee a mapping is syntactically consistent with the binding B and the value reference being passed around.

```
(DEFUN WELL-FORMED-MAPPING (M B VREF)
  ;; M is assumed to be an alist whose keys are type variables.
  (IF (NULL M)
      T
      (IF (OR (EQUAL (CDR (CAR M)) '$NIL)
              (EQUAL (CDR (CAR M)) '$T)
              (ASSOC (CDR (CAR M)) B)
              (AND (VAR-REFP (CDR (CAR M)))
                   (SAME-ROOT (CDR (CAR M)) VREF)))
          (WELL-FORMED-MAPPING (CDR M) B VREF)
          NIL)))

(DEFUN SAME-ROOT (VREF1 VREF2)
  (EQUAL (ROOT-OF-VAR-REF VREF1) (ROOT-OF-VAR-REF VREF2)))

(DEFUN ROOT-OF-VAR-REF (VREF)
  (IF (ATOM VREF)
      VREF
      (IF (MEMBER (CAR VREF) '(CAR CDR REC-TAIL DLIST-ELEM))
          (ROOT-OF-VAR-REF (CADR VREF))
          (INTERNAL-ERROR
            "ROOT-OF-VAR-REF -- Unexpected symbolic value" VREF))))
```

The soundness specification for ICONTAINED-IN is given in the lemma:

```
Lemma ICONTAINED-IN-OK:

 For any descriptors td1 and td2, simple mapping m, Lisp value v,
 symbolic value reference vref, and binding b1 covering the
 variables in td1,

  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  (well-formed-mapping m vref b1)
H3  (icontained-in td1 (dapply-subst-list-1 m td2) vref)
H4  (I td1 v b1) )
  =>
  For some b2, (I td2 v b2)
```

The proof of this lemma is given in Appendix B-G. With this lemma in hand, the proof of ALL-

DESCRIPTOR1-DISJUNCTS-OK-OK is straightforward.

Proof of Lemma ALL-DESCRIPTOR1-DISJUNCTS-OK-OK

For each i, employ Lemma ICONTAINED-IN-OK as a hypothesis, instantiated with td1 = td1$_i$, td2 = td2, and m = the m satisfying H2 for that td1$_i$. We can use the conclusion of each of these hypotheses because H1 of the main theorem equals the H1 for the instantiated lemma, H2 of the main theorem establishes its H2 and H3, and H3 equals its H4.

(I (*or td1$_1$ .. td1$_n$) v b1) expands by definition of I to (or (I td1$_1$ v b1) .. (I td1$_n$ v b1)). Since we are assuming this predicate is true, for any disjunct which is true, utilize the corresponding hypothesis, which yields the conclusion directly. QED.

Now let us return to the top level lemma.

Lemma CONTAINED-IN-INTERFACE-OK
For any descriptors td1 and td2, value v, and binding b,

```
(and (contained-in-interface td1 td2)
     (I td1 v b))
=>
For some b’, (I td2 v b’)
```

Proof of Lemma CONTAINED-IN-INTERFACE-OK

Suppose that CONTAINED-IN-INTERFACE were defined as follows:

```
(DEFUN CONTAINED-IN-INTERFACE (TD1 TD2)
  ;; TD1 and TD2 are assumed to be well-formed type descriptors.
  (IF (AND (NULL (GATHER-VARS-IN-DESCRIPTOR TD1))
           (NULL (GATHER-VARS-IN-DESCRIPTOR TD2)))
      (CONTAINED-IN TD1 TD2)
      (ALL-DESCRIPTOR1-DISJUNCTS-OK
        (MAKE-SIMPLE-MAPPING-LIST (VCONTAINED-IN TD1 TD2 NIL ’V))
        (LET ((LIFTED-TD1 (LIFT-ORS TD1)))
          (IF (OR-DESCRIPTORP LIFTED-TD1)
              (CDR LIFTED-TD1)
              (LIST LIFTED-TD1)))
        TD2
        ’V)))
```

where ALL-DESCRIPTOR1-DISJUNCTS-OK returns T if for every disjunct td1$_i$ of td1, (i.e., every descriptor in its second argument) some simple mapping m$_j$ from VCONTAINED-IN causes (ICONTAINED-IN td1$_i$ td2 m$_j$) to return true. TD1 was put into an *OR-lifted form via repeated application of Canonicalization Rules 4 - 13, with Rules 8 and 9 applied in reverse. (The proof that these rules are applicable in reverse is obtained simply by executing the proofs of Rules 8 and 9 in reverse order. See Appendix B.6.) LIFT-ORS, however, is a no-op when applied to a *DLIST. The lemmas CONTAINED-IN-OK and ALL-DESCRIPTOR1-DISJUNCTS-OK-OK trivially prove CONTAINED-IN-INTERFACE-OK. This is all that would be necessary to validate the containment algorithm.

But by definition, as previously described, CONTAINED-IN-INTERFACE tries some optimizations when there are variables in td1 or td2 in an attempt to reduce the problem to the variable-free case, for which the algorithm is simpler and the code much faster. First, if td2 contains any single instances of variables, those variables are changed to *UNIVERSAL. This is justified by the following lemma.

Lemma TC-UNIVERSALIZE-SINGLETON-VARS-1-OK

For all descriptors td, values v, type variable bindings b, and lists vlist containing all the type variables which appear only once within td,

```
(I (tc-universalize-singleton-vars-1 td vlist) v b)
=>
For some b' (I td v b')
```

This lemma is proved in Appendix B.10. Essentially, b' need differ from b only by ensuring that &i is bound to the correct component of the value v. By this lemma, then, if we demonstrate containment in our transformed td2, we are also demonstrating containment in td2.

After making this modification, if there are no variables remaining in td2, we replace all variables in td1 with *UNIVERSAL. This is justified, because clearly for any descriptor td, value v, and binding b, if some variable in a descriptor is satisfied by some component of v, then *UNIVERSAL will be satisfied. In this, we are only dropping the requirement that equal values must occupy the positions of each occurrence of a given variable.

```
(I td v b) => (I (universalize-all-vars td) v b)
```

Thus, by performing this replacement we are enlarging the set of values represented by td1. This is conservative replacement, as the problem is to determine if td1 represents a subset of the values represented by td2, and if a superset of td1's values is a subset of td2's values, then clearly td1's values form a subset of td2's values.

If after attempting these optimizations, we have rid the descriptors of all variables, CONTAINED-IN-INTERFACE calls CONTAINED-IN on the transformed descriptors, and we invoke Lemma CONTAINED-IN-OK to establish our goal. On the other hand, if there are still variables in td1 or td2, CONTAINED-IN-INTERFACE standardizes the variables apart, to satisfy the precondition for ICONTAINED-IN-OK. By this we mean that for each variable &i which appears in both td1 and td2, we create a fresh variable &j and replace all occurrences of &i in td2 with &j. Thus, our containment test establishes that

```
(I td1 v b) => for some b' (I transformed-td2 v b')
```

But it is easy to see that

```
for some b' (I transformed-td2 v b')
=>
for some b'' (I td2 v b'')
```

Just construct b'' from b' by replacing, for each substitution performed when we standardized apart, all instances of &j with &i, thus reversing the original substitution. td2 and b'' are thus name isomorphic to transformed-td2 and b', and the interpreter I will evaluate (I transformed-td2 v b') and (I td2 v b'') identically, modulo handling these different names. The formal justification for each replacement of a variable is in the Lemma RENAMING-PRESERVES-I, which is stated and proved just below. These arguments suffice to justify the operations performed in CONTAINED-IN-INTERFACE, down to the calls of CONTAINED-IN and ICONTAINED-IN. CONTAINED-IN-INTERFACE-OK therefore still holds. QED.

Lemma RENAMING-PRESERVES-I

For any descriptor td containing a reference to a variable &i, Lisp value v, type variable binding b, and type variable &j

```
 not appearing in td,

(I td v b)
=>
for some b' (I (subst &j &i td) v b')
```

Proof of Lemma RENAMING-PRESERVES-I

A fully rigorous proof of this lemma would involve an induction on the structure of descriptors, with all the cases other than the one where we consider variables either disposed of as trivial base cases or with totally straightforward employment of the inductive assumption of this lemma. The variable case would proceed as below. Since this approach would simply bury the only interesting case among a host of uninteresting ones, we choose to present the argument in the following operational manner.

Construct b' by replacing the occurrence of &i in b with &j. Then note that (I td v b) will behave identically to (I (subst &j &i td) v b') down to the point where, whether on some recursive call or on the top level call, the descriptor parameter in the first case is &i. Thus, we are referring to a call of the form

```
(I &i <component of v> b)
```

By the definition of I, this is equal to

```
(equal <component of v> (cdr (assoc &i b)))
```

The analogous call, from (I (subst &j &i td) v b'), is

```
(I &j <component of v> (subst &j &i b))
```

for the same component of v as before. This expands to

```
(equal <component of v> (cdr (assoc &j (subst &j &i b))))
```

which is, by the definition of subst,

```
(equal <component of v> (cdr (assoc &i b)))
```

QED.

Chapter 8

TESTING THE SYSTEM

Not surprisingly, extensive testing of the system revealed its strengths and weaknesses very well. In this chapter, we will present the test suite used to demonstrate the system, give an overview of the results produced, and examine certain examples which illustrate the successes and shortcomings of the tool on some representative functions. Along the way we will discuss how these results might suggest future work.

8.1 The Test Suite

The system was tested by placing it in its bare, initial state, described with the handful of signatures in Appendix A, and then submitting a collection of almost 400 functions to the tool for analysis.

These functions are in three basic groups. The first is a group of functions which bootstrap the database into a more useful collection of Lisp functions. These include some equality and membership functions, some Boolean connectives, a family of recognizer functions which compose into increasingly complex structures, and the family of CAR/CDR compositions (CADR, CDDR, CADDR, etc.). On this basis, we introduce the second group, the non-mutually recursive functions from the axioms.lisp file of an early developmental version of the Acl2 theorem prover of Boyer and Moore [Boyer 90]. The last group consists mainly of functions which were conceived during the development process to test various aspects of the system.

Many of the functions from axioms.lisp were modified so that the inference tool could produce better signatures. This was done so that the signature base would be in better condition for evaluating later results based on those signatures. In some cases, the original definition was submitted to show how the tool performed, and then a modified definition was submitted as the basis for future reference. In a few cases, a signature was composed by hand and summarily stuffed into the database, either because the function could not be modified so that the tool could produce a satisfactory result, or because the author lost patience with the process.

A number of other modifications to the axioms.lisp functions were made, and for a variety of reasons. Almost all, if not all, of the changes fell under one of the following categories:

- Macros, such as AND, OR, LIST were expanded by hand, since macros are not part of the accepted subset. This usually led to unattractive, verbose, and gratuitously inefficient code.
- Guards and IF tests were inserted, enhanced, or weakened. In some cases, this is because the original function definitions actually had inadequate guards. In other cases, guards which specified more than could be captured in the type system were weakened into type predicates. For example, a guard like (EQ (CAR X) ’FOO) may have been transformed to

(SYMBOLP (CAR X)). We attempted to comment all these cases in the examples.

- The above techniques and others were used to transform some functions frequently used as guards or IF predicates into recognizers. This was to benefit from the increased accuracy of the signature produced.
- Calls to NTH and UPDATE-NTH where the first argument is a constant were modified into the appropriate CAR/CDR nests, since NTH's signature yields no information about its result. Examples of such modifications occur in functions like GLOBAL-TABLE and UPDATE-GLOBAL-TABLE, which manipulate an object of a large state type employed in the axioms.lisp code. There is no reason why the tool could not give NTH and UPDATE-NTH special treatment and make these modifications in a prepass phase.
- Symbols with package names were changed to strings, since packaged atoms are not handled. An example is in *INITIAL-GLOBAL-TABLE*.
- Global constants were changed to parameterless functions. Support for constants would be trivial to implement, utilizing existing functionality in the tool.
- LET forms were transformed by dereferencing LET-bound variables, replacing each occurrence with the terms to which they were bound. Thus, a form like

  ```
  (LET ((X (FOO Y))) (CONS X X))
  ```

  would be transformed to

  ```
  (CONS (FOO Y) (FOO Y))
  ```

  This led to great inefficiency, since types were computed repeatedly, particularly in cases, like the ROUND function, where there are deeply nested LET constructs. This conversion, though necessary to bring the function into the accepted subset, was extremely explosive.
- Signatures for COERCE and INTERN were introduced into the initial FUNCTION-SIGNATURES. These signatures do not handle all the cases necessary to correspond to the full Common Lisp definitions of the functions, but they were satisfactory for the arguments with which those functions were invoked in the axioms code.

None of the functions from MUTUAL-RECURSION nests were included, since this definitional capability is not supported in the inference system.

A simple driver program was used to submit this test suite to the inference tool. For each function, it invokes a function NEW-FUNCTION, which simply calls the inference algorithm, then the signature checker, and reports the result, including any accounts of failure. On the Symbolics, it also measures the total time consumed on each invocation. Almost invariably, the larger portion of the time was spent in the signature checker. Invoked at the end of the testbed run, the function CHECK-FUNCTION-SIGNATURES reports on the condition of the signature database. For each function in the database, it checks the well-formedness of the signature and reports the following results from the inference tool:

- Cases where the inference tool and/or the checker reported an incomplete guard descriptor or when some function in the hereditary call tree of a function has an incomplete guard descriptor,
- Cases where the checker's guard descriptor was replaced by the inference tool guard descriptor (where the descriptors are different, but containment holds in both directions).
- Cases where the checker's signature segments are not contained in the inference algorithm segments.
- Cases where the inference algorithm decreed a function to be a recognizer and the checker disagreed.

The full results produced by the test suite are available via anonymous ftp. See Appendix H for instructions on how to retrieve them. In the suite, some examples are preceded by comments, including functions from axioms.lisp which were modified from the original. The results are not annotated with descriptive labels as they are in this chapter, but the signature components are in the same order as those presented in annotated form in the next section.

The "Signature is certified sound" flag is provided as a quick reference for the reader. It signifies that the checker has determined the guard is complete, that the guards of all the functions in the call tree are complete, that the segment containment relation holds, and if the inference algorithm claims a function is a recognizer, the checker agrees.

8.2 Summary of Results

373 functions were submitted to the tool. For 231 of these functions, the inference tool generated signatures which the checker subsequently validated to be sound.

The following things were noted for the 142 remaining functions:

```
26 functions were discarded by the inference algorithm because of
   perceived guard violations.
   Names: CASE-LIST CASE-LIST-CHECK REVERSE INTEGER-ABS-FOOBA
          GETPROPS1 HAS-PROPSP1 DIMENSIONS MAXIMUM-LENGTH DEFAULT
          ARRAY2P AREF1 COMPRESS1 AREF2 COMPRESS2 ASET2 OPEN-CHANNEL1
          STATE-P1 OPEN-INPUT-CHANNEL-P1 OPEN-OUTPUT-CHANNEL-P1
          MATCHFN FOO1 GOO1 GOO2 GOO4 GETPROPS
          TRANSLATE-DECLARATION-TO-GUARD1

2 functions were discarded by the inference algorithm because the
  signature did not become stable within the required number of
  iterations (6).
  Names: REVAPPEND COMPRESS21

31 functions had incomplete guards, according to the inference tool.
   Names: EQ EQL XXXJOIN ASSOC-EQ NTH CHAR MEMBER PAIRLIS
          NONNEGATIVE-INTEGER-QUOTIENT FLOOR CEILING TRUNCATE ROUND
          MOD REM EXPT LOGBITP NTHCDR COMPRESS11 ASET1 COMPRESS211
          ASCII-CODE DIGIT-TO-CHAR EXPLODE-NONNEGATIVE-INTEGER
          MAKE-LIST-AC AREF-T-STACK AREF-32-BIT-INTEGER-STACK
          ASET-32-BIT-INTEGER-STACK DUMMY1 GOO3 REMOVE

50 functions had incomplete guards, according to the checker (the
   difference reflecting its more stringent criteria).
   Names: all those listed immediately above, plus
          ORIG-CADR ORIG-CDDR ORIG-CADDR LENGTH LIST*-MACRO
          ASSOC-KEYWORD EXPLODE-ATOM PUSH-UNTOUCHABLE W
          CURRENT-PACKAGE KNOWN-PACKAGE-ALIST GET-UNTOUCHABLES
          LOAD-MODE SKIP-PROOFS-LOAD-MODEP MATCHFN-2 DUMMY2
          FOO5 FOO7 BAR

104 functions either had an incomplete guard or had a function
    within the hereditary call tree which had an incomplete guard,
    according to the checker.
    Names: all those listed in the two categories above, plus
           ORIG-NULL ORIG-ALISTP ORIG-SYMBOL-ALISTP ORIG-SYMBOL-LISTP
           ACL2-COUNT COND-CLAUSESP MEMBER-EQUAL-EQ MEMBER-EQ
           ORIG-ASSOC-EQUAL STANDARD-CHAR-P SYMBOLP-LISTP
           MEMBER-SYMBOL-NAME LEGAL-CASE-CLAUSESP ZEROP LOGAND ASH
           MUTUAL-RECURSION-GUARDP STANDARD-CHAR-MEMBER ASCII-<-L
           ASCII-<=-L SUBSETP ASSOC SUBST WORLDP GETPROP ADD-PAIR
           HAS-PROPSP FUNCTION-SYMBOLP *MAXIMUM-POSITIVE-32-BIT-INTEGER*
           BOUNDED-INTEGER-ALISTP ARRAY1P BOUNDED-INTEGER-ALISTP2
           ASSOC2 TYPED-IO-LST OPEN-CHANNEL1 READABLE-FILE WRITTEN-FILE
           READ-FILE-LST1 WRITEABLE-FILE-LST1 STATE-P1
           OPEN-INPUT-CHANNEL-P OPEN-OUTPUT-CHANNEL-P
```

```
          OPEN-OUTPUT-CHANNEL-ANY-P1 OPEN-INPUT-CHANNEL-ANY-P1
          DELETE-PAIR SOME-SLASHABLE T-STACK-LENGTH1
          32-BIT-INTEGER-STACK-LENGTH1 PUT-ASSOC-EQ FOO SYM-INP
          APPLY-SUBST1 COND-MACRO
          TRANSLATE-DECLARATION-TO-GUARD/INTEGER

17 functions had signatures which failed the segment containment
   test of the checker.
   Names: XXXJOIN-2 SUBST ARRAY1P UPDATE-NTH OPEN-CHANNEL-LST-2
          OPEN-CHANNELS-P-2 STATE-P1 STATE-P MAKE-LIST-AC
          ASET-32-BIT-INTEGER-STACK DECREMENT-BIG-CLOCK READ-IDATE
          READ-RUN-TIME GOO6 FOO7 APPLY-SUBST1 FOOP

5 functions designated as recognizers by the inference algorithm
  were rejected by the checker.
  Names: PROPER-CONSP-2 NON-T-NIL-SYMBOL-PLISTP
         OPEN-CHANNEL-LST-2 OPEN-CHANNELS-P-2 STATE-P

97 functions reported only incomplete guards (as opposed to segment
   containment failure or recognizer rejection) as a cause for
   soundness rejection.

2 functions had their checker guard replaced by the inference tool
  guard.
  Names: MEMBER REMOVE

25 functions were submitted and processed, and then minor variants
   were resubmitted, causing the previous results to be overwritten
   in the database (but preserved in the log)
   Names: OPEN-INPUT-CHANNELS UPDATE-OPEN-INPUT-CHANNELS
          OPEN-OUTPUT-CHANNELS UPDATE-OPEN-OUTPUT-CHANNELS
          GLOBAL-TABLE UPDATE-GLOBAL-TABLE T-STACK UPDATE-T-STACK
          32-BIT-INTEGER-STACK UPDATE-32-BIT-INTEGER-STACK
          BIG-CLOCK-ENTRY UPDATE-BIG-CLOCK-ENTRY IDATES
          UPDATE-IDATES RUN-TIMES UPDATE-RUN-TIMES FILE-CLOCK
          UPDATE-FILE-CLOCK READABLE-FILES WRITTEN-FILES
          UPDATE-WRITTEN-FILES READ-FILES UPDATE-READ-FILES
          WRITEABLE-FILES OPEN-CHANNEL1

21 functions had their original result replaced in the
   database with a new result which would allow the process to
   continue reasonably for subsequently submitted functions.
   Names: W PRINT-RATIONAL-AS-DECIMAL GET-TIMER PUT-GLOBAL
          GET-GLOBAL MAKUNBOUND-GLOBAL BOUNDP-GLOBAL1 PRINC$
          OPEN-OUTPUT-CHANNEL-P1 OPEN-INPUT-CHANNEL-P1
          EXPLODE-NONNEGATIVE-INTEGER STATE-P1
          WORLD-KNOWN-PACKAGE-ALISTP COMPRESS21 AREF2 COMPRESS1
          ARRAY2P REVAPPEND DIMENSIONS MAXIMUM-LENGTH DEFAULT
```

Among the examples in which the inference algorithm detected a guard violation, in some, like INTEGER-ABS-FOOBA, GETPROPS1 (presented in the next section), HASPROPS1, MATCHFN, FOO1, GOO1, GOO2, and GOO4, there is a real guard violation in the function. In a number of other cases, a problem arises because the tool cannot derive strong enough descriptors for forms used as guards or IF tests to govern the violating call. For example, in CASE-LIST and CASE-LIST-CHECK, a call to LEGAL-CASE-CLAUSESP is used as a guard. But because the definition of LEGAL-CASE-CLAUSESP employs a test (EQ ’OTHERWISE (CAR (CAR TL))), it is not treated as a recognizer, and its signature is therefore not sufficiently strong to define a safe type context for the operations within CASE-LIST and CASE-LIST-CHECK. Another class of problem is exhibited by the guard violations in DIMENSIONS, DEFAULT, ARRAY2P, AREF1, COMPRESS11, AREF2, ASET2, and COMPRESS2. In these functions, a relation is used as a governor, either within the guard or in an IF test, and relations yield no information as type predicates. A third class of guard violations is exhibited in STATE-P1, OPEN-INPUT-CHANNEL-P1, and OPEN-OUTPUT-CHANNEL-P1, where the violation is caused by the failure of the system to treat abstraction functions whose definitions are calls to destructor functions (like CADDR) in the same way as it treats the destructor function CADDR. As discussed in Section 4.4.4, the TYPE-PREDICATE-P algorithm within the inference algorithm gives special treatement to certain destructor functions. If this treatment were extended to user-defined destructors like GLOBAL-TABLE, the tool would be able to verify the guards in STATE-P1, *et al*.

Among the functions whose guard descriptors were judged incomplete by the checker but complete by the inference algorithm, only the LENGTH function appears to get much use in subsequently defined functions. Among the functions for which the inference algorithm and the checker agree on incompleteness are many, like EQ, EQL (presented below), NTH, MEMBER, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM, EXPT, DIMENSIONS, and a number of others which receive subsequent use. The functions just named all have guards which either contain relations between formal parameters, which have some value dependency, like (NOT (EQUAL X 0)), or which have a disjunction on the types of distinct parameters, for example

```
(OR (EQLABLEP X) (EQLABLE-LISTP Y))
```

In none of these cases is there a way to fully capture the guard with a vector of descriptors, so the problem does not reflect on the checker's stringent requirement that a guard must be a conjunction of recognizer calls on distinct parameters. Thus, of the 102 functions whose signatures the checker refused to acknowledge for soundness, it is almost certain that the failure was due to guards which could not legitimately be captured by the type system, regardless of the strength of the heuristics used.

A number of different reasons accounted for the failure of the segment containment check in 17 cases. Perhaps the most common cause of failure was dissonance in the use of type variables between the inference algorithm and the checker. If multiple occurrences of variables in the original signature do not lead to checker segments with similar multiple occurrences, containment is almost sure to fail. A simple, but characteristic example is GOO6, which we will display below, but a similar problem is shared by XXXJOIN-2, ASET-32-BIT-INTEGER-STACK, DECREMENT-BIG-CLOCK, READ-IDATE, and READ-RUN-TIME. Another common problem, which afflicted SUBST, APPLY-SUBST1, FOOP, and UPDATE-NTH was the absence of an applicable *REC rule in the heuristic VCONTAINED-IN algorithm. In any such case, it is likely a sound rule could be supplied. Similarly, the lack of a good DUNIFY-DESCRIPTORS *REC rule caused a problem with OPEN-CHANNEL-LST-2 and OPEN-CHANNELS-P-2. Infinite loops involving *REC descriptors in the DUNIFY-DESCRIPTORS were resolved by the TERM-RECS mechanism while processing OPEN-CHANNEL-LST-2, OPEN-CHANNELS-P-2, STATE-P, FOO7, likely leading to creation by default of overly inclusive descriptors which contributed to containment failure. Some sort of bug in the inference algorithm, probably related to the closure of *FIX forms, led to an incorrect signature being generated for MAKE-LIST-AC which the checker correctly refused to validate. Finally, the ARRAY1P, given below, example illustrates some sort of problem with the factoring of *OR, where a checker segment had a result type of (*OR $NIL $T), but the original segments all had result types of either $T or $NIL, causing containment to fail.

For three of the five functions whose recognizer status was rejected by the checker, this failure was due to the failure of the segment containment test. For the other two, PROPER-CONSP-2 and NON-T-NIL-SYMBOL-PLISTP, the lack of a suitable *REC rule for unification or containment seemed to play a role. For these two, their signatures were otherwise valid, and thus could be used for subsequent sound inferences.

Of the functions which were resubmitted, most were functions from axioms.lisp which invoked either the functions NTH or UPDATE-NTH, where the numeric argument was a constant. Since it was known that the signatures for NTH or UPDATE-NTH yield no information about the result, the original functions were submitted for illustration and then modified to have the calls to NTH or UPDATE-NTH replaced with appropriate CAR/CDR/CONS operations. The results make for interesting comparison and highlight that NTH and UPDATE-NTH would be good candidates for special treatment in a prepass.

Of the functions for which hand-coded signatures were jammed into the database, nine were functions for which the inference tool had signalled guard violations, two were functions whose signatures never stabilized, and one for a function whose signature had an incomplete guard. For the remainder, we judged by inspection of the function that similar problems would befall, and chose simply to not bother with the original submission.

We believe these results are quite encouraging. In spite of the known inadequacies of the inference algorithm, for almost two-thirds of the functions submitted, the tool generated sound signatures. Some of these examples were designed to fail, via guard violation, for instance, or by use of guards known not to be complete. A user who attempted to use the complete guard regimen wherever possible would likely receive significant assistance from the tool in the task of guard verification. Moreover, for those functions where guards would need to express more than could be captured in the type system, a highly symbiotic relationship is easily imagined for the inference system and a general-purpose, automatic theorem prover capable of handling those cases.

8.3 Selected Examples

When displayed, the timings are from execution on a Symbolics 3645 machine. The system runs much faster on a Sun Sparc platform. For some functions, notably large and complex functions manipulating the huge state types of axioms.lisp, the amount of time consumed by the tool was significant, occasionally exceeding a half hour.

The tool performed quite well on recognizer functions. Some examples follow.

```
(DEFUN TRUE-LISTP (X)
  (IF (NULL X) (NULL X) (IF (CONSP X) (TRUE-LISTP (CDR X)) NIL)))

Function: TRUE-LISTP
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   -> $T)
 (((*REC !REC1
        (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
             $NON-T-NIL-SYMBOL $STRING $T
             (*CONS *UNIVERSAL (*RECUR !REC1)))))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor:
  (*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
TC validates recognizer: T
Signature is certified sound: T

Time: 2. seconds


(DEFUN EQLABLEP (X)
```

```
  (IF (RATIONALP X)
      (RATIONALP X)
      (IF (SYMBOLP X) (SYMBOLP X) (CHARACTERP X))))

Function: EQLABLEP
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $T))
   -> $T)
 (((*OR $STRING (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor:
  (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
       $NON-T-NIL-SYMBOL $T)
TC validates recognizer: T
Signature is certified sound: T

Time: 3. seconds


(DEFUN EQLABLE-ALISTP (X)
  (IF (ATOM X)
      (EQUAL X NIL)
      (IF (CONSP (CAR X))
          (IF (EQLABLEP (CAR (CAR X))) (EQLABLE-ALISTP (CDR X)) NIL)
          NIL)))

Function: EQLABLE-ALISTP
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*REC EQLABLE-ALISTP
         (*OR $NIL
              (*CONS (*CONS (*OR $CHARACTER $INTEGER $NIL
                                 $NON-INTEGER-RATIONAL
                                 $NON-T-NIL-SYMBOL $T)
                            *UNIVERSAL)
                     (*RECUR EQLABLE-ALISTP)))))
   -> $T)
 (((*REC !REC1
         (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
              $NON-T-NIL-SYMBOL $STRING $T
              (*CONS *UNIVERSAL (*RECUR !REC1))
              (*CONS (*OR $CHARACTER $INTEGER $NIL
                          $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
                          $STRING $T
                          (*CONS (*OR $STRING
                                      (*CONS *UNIVERSAL *UNIVERSAL))
                                 *UNIVERSAL))
                     *UNIVERSAL))))
   -> $NIL)
TC segments contained in Segments: T
```

```
Recognizer descriptor:
  (*REC EQLABLE-ALISTP
        (*OR $NIL
             (*CONS (*CONS (*OR $CHARACTER $INTEGER $NIL
                                $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
                                $T)
                           *UNIVERSAL)
                    (*RECUR EQLABLE-ALISTP))))
TC validates recognizer: T
Signature is certified sound: T

Time: 10. seconds


(DEFUN COND-CLAUSESP-1 (CLAUSES)
  (IF (CONSP CLAUSES)
      (IF (CONSP (CAR CLAUSES))
          (IF (TRUE-LISTP (CAR CLAUSES))
              (IF (NOT (CONSP (CDDR (CAR CLAUSES))))
                  (COND-CLAUSESP-1 (CDR CLAUSES))
                  NIL)
              NIL)
          NIL)
      (NULL CLAUSES)))

Function: COND-CLAUSESP-1
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*REC COND-CLAUSESP-1
         (*OR $NIL
              (*CONS (*CONS *UNIVERSAL
                            (*OR $NIL (*CONS *UNIVERSAL $NIL)))
                     (*RECUR COND-CLAUSESP-1)))))
   -> $T)
 (((*REC !REC2
         (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
              $NON-T-NIL-SYMBOL $STRING $T
              (*CONS *UNIVERSAL (*RECUR !REC2))
              (*CONS (*OR $CHARACTER $INTEGER $NIL
                          $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
                          $STRING $T
                          (*CONS
                            *UNIVERSAL
                            (*OR $CHARACTER $INTEGER
                                 $NON-INTEGER-RATIONAL
                                 $NON-T-NIL-SYMBOL $STRING $T
                                 (*CONS
                                   *UNIVERSAL
                                   (*OR $CHARACTER $INTEGER
                                        $NON-INTEGER-RATIONAL
                                        $NON-T-NIL-SYMBOL $STRING $T
                                        (*CONS *UNIVERSAL *UNIVERSAL)
                                        )))))
                     *UNIVERSAL))))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor:
  (*REC COND-CLAUSESP-1
        (*OR $NIL
             (*CONS (*CONS *UNIVERSAL
                           (*OR $NIL (*CONS *UNIVERSAL $NIL)))
```

```
                    (*RECUR COND-CLAUSESP-1))))
TC validates recognizer: T
Signature is certified sound: T

Time: 8. seconds
```

This kind of result continued to scale up to enormous structures, as witnessed by the examples WORLDP, KNOWN-PACKAGE-ALISTP, TIMER-ALISTP, OPEN-CHANNEL1-2, OPEN-CHANNEL-LST-2, READABLE-FILE-2, READABLE-FILES-LST-2, WRITTEN-FILE-2, WRITTEN-FILE-LST-2, READ-FILE-2, READ-FILE-LST-2, WRITEABLE-FILE-2, WRITEABLE-FILE-LST-2. (Most of these functions were modified, for instance by replacing calls to NTH to the proper CAR/CDR nests.)

```
(DEFUN PROPER-CONSP (X)
  (IF (CONSP X)
      (IF (CONSP (CDR X)) (PROPER-CONSP (CDR X)) (EQUAL (CDR X) NIL))
      NIL))

Function: PROPER-CONSP
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*REC !REC1 (*CONS *UNIVERSAL (*OR $NIL (*RECUR !REC1)))))
   -> $T)
 (((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $STRING $T
        (*REC !REC3
              (*CONS *UNIVERSAL
                     (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                          $NON-T-NIL-SYMBOL $STRING $T
                          (*RECUR !REC3))))))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor:
  (*REC !REC1 (*CONS *UNIVERSAL (*OR $NIL (*RECUR !REC1))))
TC validates recognizer: T
Signature is certified sound: T

Time: 8. seconds
```

IMPROPER-CONSP illustrates the use of negation in formulating a recognizer.

```
(DEFUN IMPROPER-CONSP (X)
  (IF (CONSP X) (NOT (PROPER-CONSP X)) NIL))

Function: IMPROPER-CONSP
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*REC !REC2
         (*CONS *UNIVERSAL
                (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                     $NON-T-NIL-SYMBOL $STRING $T (*RECUR !REC2)))))
```

```
   -> $T)
 (((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $STRING $T
        (*REC !REC4
              (*CONS *UNIVERSAL (*OR $NIL (*RECUR !REC4))))))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor:
  (*REC !REC2
        (*CONS *UNIVERSAL
               (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                    $NON-T-NIL-SYMBOL $STRING $T (*RECUR !REC2))))
TC validates recognizer: T
Signature is certified sound: T

Time: 6. seconds
```

The Boolean connectives produced the desired signatures, for example:

```
(DEFUN IMPLIES (P Q)
  " IMPLIES is the ACL2 implication function.  (implies P Q) means that
 either P is false or Q is true."
  (IF P (IF Q T NIL) T))

Function: IMPLIES
Guard computed by the tool:
 (*UNIVERSAL *UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL *UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
   (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $T)
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
   $NIL)
   -> $NIL)
 (($NIL *UNIVERSAL) -> $T)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T

Time: 7. seconds
```

Unfortunately, the guard for EQL is not an acceptable type predicate, since it is a disjunction, rather than a conjunction, of recognizer calls on distinct arguments. (The guard is the macro expansion of (OR (EQLABLEP X) (EQLABLEP Y)).) So the guard descriptors default to *UNIVERSAL, and the guard is not complete. This shortcoming will frequently percolate through applications as an inability to guarantee the guard verification

Notice the form (*FREE-TYPE-VAR 1). This is just the form in which type variables are stored in the system database. Each distinct variable will have a different integer within this representation.

```
(DEFUN EQL (X Y)
  (DECLARE (XARGS :GUARD (IF (EQLABLEP X) (EQLABLEP X) (EQLABLEP Y))))
  (EQUAL X Y))
```

```
Function: EQL
Guard computed by the tool:
 (*UNIVERSAL *UNIVERSAL)
Guard complete: NIL
All called functions complete: T
TC Guard:
 (*UNIVERSAL *UNIVERSAL)
TC Guard complete: NIL
TC All called functions complete: NIL
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($NIL $NIL) -> $T)
 (($T $T) -> $T)
 (((*FREE-TYPE-VAR 1) (*FREE-TYPE-VAR 1)) -> $T)
 (((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $STRING (*CONS *UNIVERSAL *UNIVERSAL))
   (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $NIL)
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
   $NIL)
   -> $NIL)
 (($T (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
           $STRING (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: NIL

Time: 8. seconds
```

TWICE-GUARDED is a lovely example. In the (CONSP X) case, the tool deduces that the inner recursive call can return only NIL, and likewise for the outer one. Then, seeing that the result type for the CONSP case is the same as for the (ATOM X) case (which because of the (TRUE-LISTP X) guard has been narrowed to (NULL X)), and since there is only one argument, it merges the (NULL X) and (CONSP X) segments by disjoining their arguments with *OR. When it canonicalizes this result, it notices that it can fold the *OR back into the *REC descriptor for TRUE-LISTP.

```
(DEFUN TWICE-GUARDED (X)
  (DECLARE (XARGS :GUARD (TRUE-LISTP X)))
  (IF (ATOM X) X (TWICE-GUARDED (TWICE-GUARDED (CDR X))))))

Function: TWICE-GUARDED
Guard computed by the tool:
 ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
Guard complete: T
All called functions complete: T
TC Guard:
 ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T
```

The CADR function illustrates how one must coddle the tool to help it establish the formal soundness claim. First we show the function as normally defined (but renamed to ORIG-CADR). The guard is the form

```
(OR (NULL X) (AND (CONSP X) (OR (NULL (CDR X)) (CONSP (CDR X)))))
```

with the AND and OR macros expanded.

```
(DEFUN ORIG-CADR (X)
  (DECLARE (XARGS :GUARD
                  (IF (NULL X) (NULL X)
                      (IF (CONSP X)
                          (IF (NULL (CDR X)) (NULL (CDR X))
                              (CONSP (CDR X)))
                          NIL))))
  (CAR (CDR X)))

Function: ORIG-CADR
Guard computed by the tool:
 ((*OR $NIL
       (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))))
Guard complete: T
All called functions complete: T
TC Guard:
 ((*OR $NIL
       (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))))
TC Guard complete: NIL
TC All called functions complete: NIL
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $NIL (*CONS *UNIVERSAL $NIL)))
   -> $NIL)
 (((*CONS *UNIVERSAL (*CONS (*FREE-TYPE-VAR 1) *UNIVERSAL)))
   -> (*FREE-TYPE-VAR 1))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: NIL
```

The inference algorithm is clever enough to realize that the guard is complete. But the guard expression does not conform to the requirement, imposed by the checker, that a guard must be a conjunction of recognizer calls on distinct arguments in order to generate complete descriptors. Thus, the "TC Guard complete" flag is NIL, and likewise for the "TC All called functions complete" flag, which can only be T if the "TC Guard complete" flag is T.

However, if we take the guard expression and make it the body of a function, CADR-GUARD, the tool will determine that the function is a recognizer and generate a signature which it certifies to be sound.

```
(DEFUN CADR-GUARD (X)
  (IF (NULL X) (NULL X)
      (IF (CONSP X)
          (IF (NULL (CDR X)) (NULL (CDR X)) (CONSP (CDR X))) NIL)))

Function: CADR-GUARD
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $NIL
        (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))))
   -> $T)
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
```

```
        $STRING $T
        (*CONS *UNIVERSAL
               (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                    $NON-T-NIL-SYMBOL $STRING $T))))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor:
  (*OR $NIL
       (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL))))
TC validates recognizer: T
Signature is certified sound: T
```

Then, replacing the old guard with a call to this function enables the checker to validate that the signature, which is identical to the one generated for ORIG-CADR, is sound. Certainly, relaxing the checker's complete guard criteria would make the system a bit more usable. But modulo a little inconvenience, no functionality is lost with the current implementation.

```
(DEFUN CADR (X)
  (DECLARE (XARGS :GUARD (CADR-GUARD X)))
  (CAR (CDR X)))

Function: CADR
Guard computed by the tool:
 ((*OR $NIL
       (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))))
Guard complete: T
All called functions complete: T
TC Guard:
 ((*OR $NIL
       (*CONS *UNIVERSAL (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $NIL (*CONS *UNIVERSAL $NIL)))
   -> $NIL)
 (((*CONS *UNIVERSAL (*CONS (*FREE-TYPE-VAR 1) *UNIVERSAL)))
   -> (*FREE-TYPE-VAR 1))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T
```

There are a number of cases in the testbed where examples were modified and then resubmitted with guards in a form amenable to completeness. Look for functions whose names end with "-GUARD".

As previously discussed, the lack of a type-instantiable variable within *REC descriptors leaves something to be desired in the signature for the APPEND function.

```
(DEFUN APPEND (X Y)
  (DECLARE (XARGS :GUARD (TRUE-LISTP X)))
  (IF (NULL X) Y (CONS (CAR X) (APPEND (CDR X) Y))))

Function: APPEND
Guard computed by the tool:
 ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
  *UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 ((*REC TRUE-LISTP (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP))))
  *UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
```

```
Segments:
 (($NIL (*FREE-TYPE-VAR 1)) -> (*FREE-TYPE-VAR 1))
 (((*CONS (*FREE-TYPE-VAR 2)
          (*REC TRUE-LISTP
                (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
   (*FREE-TYPE-VAR 1))
   -> (*CONS (*FREE-TYPE-VAR 2)
             (*REC !REC4
                   (*OR (*FREE-TYPE-VAR 1)
                        (*CONS *UNIVERSAL (*RECUR !REC4))))))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T
```

Notice that the *UNIVERSAL within !REC4 represents the element types of the list X. If the second segment could employ a different type of variable, for example:

```
 (((*CONS (*NEW-TYPE-VAR 2)
          (*REC TRUE-LISTP
                (*OR $NIL
                     (*CONS (*NEW-TYPE-VAR 2) (*RECUR TRUE-LISTP)))))
   (*FREE-TYPE-VAR 1))
   -> (*CONS (*NEW-TYPE-VAR 2)
             (*REC !REC4
                   (*OR (*FREE-TYPE-VAR 1)
                        (*CONS (*NEW-TYPE-VAR 2) (*RECUR !REC4))))))
```

then the signature for APPEND-INTLISTS below could carry into the body of the *REC descriptor in its result type the information that the elements from the first parameter were integers. This would be a worthy upgrade to the system. As it is, we must be content with:

```
(DEFUN APPEND-INTLISTS (X Y)
  (DECLARE (XARGS :GUARD
                  (IF (INTEGER-LISTP X) (INTEGER-LISTP Y) NIL)))
  (APPEND X Y))

Function: APPEND-INTLISTS
Guard computed by the tool:
 ((*REC INTEGER-LISTP
        (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
  (*REC INTEGER-LISTP
        (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
Guard complete: T
All called functions complete: T
TC Guard:
 ((*REC INTEGER-LISTP
        (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP))))
  (*REC INTEGER-LISTP
        (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($NIL (*REC INTEGER-LISTP
              (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> (*REC INTEGER-LISTP
            (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
 (((*CONS $INTEGER
          (*REC INTEGER-LISTP
                (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   (*REC INTEGER-LISTP
         (*OR $NIL (*CONS $INTEGER (*RECUR INTEGER-LISTP)))))
   -> (*CONS $INTEGER
             (*REC !REC4-G1568
                   (*OR (*REC INTEGER-LISTP
```

```
                              (*OR $NIL
                                   (*CONS $INTEGER
                                    (*RECUR INTEGER-LISTP))))
                        (*CONS *UNIVERSAL (*RECUR !REC4-G1568))))))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T
```

As mentioned in the discussion of the SAD-BUT-TRUE-LISTP example in Section 4.4.7, the tool can do a less than adequate job when a function which is not a recognizer decomposes its argument recursively. A good example of this phenomenon is the function STANDARD-CHAR-LISTP, below. Ideally, the signature would have included a segment mapping a list of characters to $T or $NIL, and another segment mapping everything else to $NIL. But since we do not construct *FIX forms on the left hand side of our segments, we wind up with a signature which suggests $T as a possible result when the argument is, for example, the CONS of a character onto an integer. There are several other examples in the test suite illustrating this phenomenon, among them LAST, FLIP, and COND-CLAUSESP (which also shows the futility of using a LENGTH test as a structure constraint in restricting TRUE-LISTP types).

```
(DEFUN STANDARD-CHAR-LISTP (L)
       (IF (CONSP L)
           (IF (CHARACTERP (CAR L))
               (IF (STANDARD-CHAR-P (CAR L))
                   (STANDARD-CHAR-LISTP (CDR L)) NIL)
               NIL)
           (EQUAL L NIL)))

Function: STANDARD-CHAR-LISTP
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*CONS $CHARACTER *UNIVERSAL)) -> (*OR $NIL $T))
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $NIL)
 (($NIL) -> $T)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: NIL
```

The signature for STANDARD-CHAR-LISTP is not certified sound because the signature for MEMBER, which has an incomplete guard, is not certified.

Next is an example from axioms.lisp where the inference algorithm discovers a guard violation. Other examples in the test suite where guard violations are detected include the following.

```
(DEFUN GETPROPS1 (ALIST)
  (IF (NULL ALIST) NIL
      (IF (NULL (CDR (CAR ALIST))) (GETPROPS1 (CDR ALIST))
          (CONS (CONS (CAR (CAR ALIST)) (CAR (CDR (CAR ALIST))))
                (GETPROPS1 (CDR ALIST))))))

******* Type-checking error detected *******

Form: (CAR ALIST)
```

```
There is a guard violation.  When in the context of the function
call, the variables are of type:
 ALIST: $CHARACTER
The actual arguments to the function are of type:
 Arg: $CHARACTER
The guard of the called function is:
 Arg: (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL))

Analysis terminated

Could not type this function.  No action taken.

Result:
(*UNABLE-TO-TYPE*
    ((*UNIVERSAL) T T
     (*GUARD-VIOLATION
         ((CAR ALIST) *MARKER-6
          ((ALIST *OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                  $NON-T-NIL-SYMBOL $STRING $T
                  (*CONS *UNIVERSAL *UNIVERSAL)))
          ((*MARKER-5) &7))
         (($CHARACTER) ($CHARACTER)))
     NIL))
```

Next we show two examples where the checker could not confirm segment containment and therefore refused to validate the soundness of a signature. The first suggests a problem in the checker algorithm with carrying variables into its final result. The checker's strategy of carrying both abstract and concrete type alists and of generating both minimal and maximal segments enabled it to handle almost all the cases with which we tested it. But this case is evidence that there is still some price being paid for not having a descriptor form, like (*BOTH &1 $CHARACTER), which provides annotation of a type variable with a descriptor characterizing a type constraint on the variable.

```
(DEFUN GOO6 (X)
       (DECLARE (XARGS :GUARD (CONSP X)))
       (IF (EQUAL (CAR X) 0) 0 (CONS (CAR X) NIL))))

Segment containment failed
 TC-SEGMENT =
   (((*CONS $CHARACTER *UNIVERSAL)) -> (*CONS $CHARACTER $NIL))
 IMP-SEGMENTS = (((*CONS $INTEGER *UNIVERSAL) -> $INTEGER)
                 ((*CONS &1 *UNIVERSAL) -> (*CONS &1 $NIL)))

Function: GOO6
Guard computed by the tool:
 ((*CONS *UNIVERSAL *UNIVERSAL))
Guard complete: T
All called functions complete: T
TC Guard:
 ((*CONS *UNIVERSAL *UNIVERSAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*CONS $INTEGER *UNIVERSAL))
   -> $INTEGER)
 (((*CONS (*FREE-TYPE-VAR 1) *UNIVERSAL))
   -> (*CONS (*FREE-TYPE-VAR 1) $NIL))
TC segments contained in Segments: NIL
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: NIL
```

The second suggests a problem with the factorization of an *OR descriptor. If the troublesome segment were factored into two segments, one with a result type of $NIL and the other $T, each would have been contained in one of the original signature segments. This factorization would be a relatively easy upgrade, and its formal justification would be straightforward. But there might be a significant performance penalty, and we would have to weigh whether it is justified by the rare occasion where benefit is derived.

```
(DEFUN ARRAY1P (NAME L)
  (IF (SYMBOLP NAME)
      (IF (ALISTP L)
          (IF (KEYWORD-LISTP (CDR (HEADER NAME L)))
              (IF (TRUE-LISTP (DIMENSIONS NAME L))
                  (IF (EQUAL (LENGTH (DIMENSIONS NAME L)) 1)
                      (IF (INTEGERP (CAR (DIMENSIONS NAME L)))
                          (IF (INTEGERP (MAXIMUM-LENGTH NAME L))
                              (IF (< 0 (CAR (DIMENSIONS NAME L)))
                               (IF
                                (< (CAR (DIMENSIONS NAME L))
                                   (MAXIMUM-LENGTH NAME L))
                                (IF
                                 (<= (MAXIMUM-LENGTH NAME L)
                                  (*MAXIMUM-POSITIVE-32-BIT-INTEGER*))
                                 (BOUNDED-INTEGER-ALISTP L
                                  (CAR (DIMENSIONS NAME L)))
                                 NIL)
                                NIL)
                               NIL)
                              NIL)
                          NIL)
                      NIL)
                  NIL)
              NIL)
          NIL)
      NIL))

Segment containment failed
 TC-SEGMENT = (($NIL $NIL) -> (*OR $NIL $T))
 IMP-SEGMENTS as below

Function: ARRAY1P
Guard computed by the tool:
 (*UNIVERSAL *UNIVERSAL)
Guard complete: T
All called functions complete: NIL
TC Guard:
 (*UNIVERSAL *UNIVERSAL)
TC Guard complete: T
TC All called functions complete: NIL
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $NIL $NON-T-NIL-SYMBOL $T)
   (*OR (*CONS (*CONS *UNIVERSAL *UNIVERSAL)
               (*REC ALISTP
                     (*OR $NIL
                          (*CONS (*CONS *UNIVERSAL *UNIVERSAL)
                                 (*RECUR ALISTP)))))
        (*REC !REC1
              (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL
                   $NON-T-NIL-SYMBOL $STRING $T
                   (*CONS *UNIVERSAL (*RECUR !REC1))
                   (*CONS (*OR $CHARACTER $INTEGER $NIL
                               $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
                               $STRING $T)
                          *UNIVERSAL)))
        (*REC ALISTP
              (*OR $NIL
                   (*CONS (*CONS *UNIVERSAL *UNIVERSAL)
                          (*RECUR ALISTP))))))
   -> $NIL)
 (((*OR $NIL $NON-T-NIL-SYMBOL $T) $NIL)
   -> $T)
```

```
 (((*OR $NIL $NON-T-NIL-SYMBOL $T)
   (*CONS (*CONS (*OR $INTEGER $NON-T-NIL-SYMBOL) *UNIVERSAL)
          (*REC ALISTP
                (*OR $NIL
                     (*CONS (*CONS *UNIVERSAL *UNIVERSAL)
                            (*RECUR ALISTP))))))
   -> (*OR $NIL $T))
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $STRING
        (*CONS *UNIVERSAL *UNIVERSAL))
   *UNIVERSAL)
   -> $NIL)
TC segments contained in Segments: NIL
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: NIL
```

Chapter 9
FUTURE WORK AND CONCLUSION

In this chapter we summarize and comment upon the significance of this research and then discuss several avenues for its extension.

9.1 Summary

In this report we have presented a system for inferring type signatures for functions defined in a purely applicative subset of Common Lisp. We defined a mechanism which allows a user to annotate his function with a guard declaration, which can be an arbitrary predicate on the function's parameters, but which is most fruitfully used here for stating the type requirements on his parameters. The guard mechanism has an advantage over the SATISFIES predicate of Common Lisp that it can be used to state required relations among several parameters, but this usage is not well supported by our type system.

Starting from a very small library of pre-defined function signatures, the system accepts new function definitions, checks that the type constraints embodied in guards are observed on function calls, and assigns a type signature to any function for which no guard violation is diagnosed.

The type inference algorithm is heuristic and has no formal model, but the signatures it produces have a formal semantics. We have defined a checker algorithm which validates that a signature produced by the inference algorithm is correct with respect to our evaluation model for Lisp and an interpreter semantics for our type descriptors. For the case where the guards of a function and all its subsidiary functions are completely captured by type descriptors, we presented a proof establishing that any signature certified by the checker is correct. By submitting a significant collection of typical (and some atypical) functions to the system, we have subjectively observed that the signatures produced are "good", in addition to being sound.

This system differs significantly from previous type inference efforts. Other type inference systems for Lisp have generally focussed on automatically annotating a program to supply compiler directives, with the goal being to reduce the need for run-time type checking. They have operated only with respect to a known set of previously declared types, mainly the native types for atomic Lisp objects, though some recent work supports record types declared with DEFSTRUCT and an abstraction for simple lists. Though some of these systems offer a more extensive treatment of primitive types and operate on a larger subset of the Common Lisp language, judging from the literature, our system represents several significant steps over any previous work:

- Rather than producing annotations on forms, it generates complete function signatures which capture *ad hoc* polymorphism and modularize results at function boundaries.
- It successfully infers and manipulates arbitrary recursive types which have not been

previously declared.

- It performs the static diagnosis of type errors and, under certain constraints on the style of expression used in guards, can certify that a function is free from type errors.
- Its results are validated by a rigorous mathematical proof with respect to a formal evaluation model and a formal semantics for type signatures.

This system also differs significantly, both in style and substance, from other type systems which do satisfy the above criteria (along with many other marvelous formal properties) but operate on languages other than Lisp. Truly formidable type systems have been created for languages carefully designed with type inference in mind. But to our knowledge, no other such system works on a language which treats data structures as freely as Common Lisp. In particular, our type system supports the creation and recognition of objects of virtually arbitrary new and undeclared structure, such as can be freely constructed with Lisp CONS, while simultaneously supporting the *ad hoc* polymorphism inherent in Lisp's IF.

Given that the type inference problem for our Common Lisp subset is undecidable, this system has a strongly heuristic flavor. The separation of the heuristics in the type inference algorithm and the checker from the formal validation of its results allows for incremental improvement or complete revision in its heuristic techniques without imposing any new proof requirement, so long as the semantic model remains stable.

The formal underpinnings of this system and its ability to deal with recursive functions suggest a potentially symbiotic relationship with other formal, mechanical tools for analysis of Lisp functions. In particular, powerful theorem provers which utilize Lisp-like logics are available to support mathematical modelling of computing systems, and this type system could play the role of a specialized proof assistant for type-related problems. Alternatively, this system could conceivably support a Lisp compiler or Lisp system development tools in ways other Lisp type inference systems could not, by providing reliable type annotations for complex, freely constructed structures.

9.2 Future Work

This work could be extended in a number of ways. Three threads of work are apparent:

- re-implementing or adding certain features and heuristics to improve its performance on the task as currently defined,
- extending the system to handle an enlarged language subset, and
- finding fruitful ways to utilize the system in a larger system development context.

We will discuss each thread separately.

9.2.1 Improvements to the System

Some of these possibilities have been previously mentioned in the report, but we collect them here and add a few new suggestions. Some amount to little more than minor enhancements to the system, while others suggest new approaches to particular problems which might be more fruitful than those currently implemented.

9.2.1-A A New Class of Variables

Probably the single most important feature which could be added to the system would be a second class of type variable, instantiable with type descriptors rather than singleton values. This has been previously discussed with respect to the APPEND example. Such a variable could appear within a replicating component of a *REC descriptor, for instance. Denoting it by %1 in the following example, a signature segment for APPEND might be:

```
(((*CONS %1 (*REC TRUE-LISTP
                  (*OR $NIL (*CONS %1 (*RECUR TRUE-LISTP)))))
  &2)
  -> (*CONS %1 (*REC !REC4 (*OR &2 (*CONS %1 (*RECUR !REC4))))))
```

rather than what is currently generated:

```
(((*CONS *UNIVERSAL
         (*REC TRUE-LISTP
               (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP)))))
  &2)
  -> (*CONS *UNIVERSAL
            (*REC !REC4 (*OR &2 (*CONS *UNIVERSAL (*RECUR !REC4))))))
```

Thus, with the help of a little canonicalization, if we called APPEND with two lists of integers, we could infer that the result would be a list of integers rather than a true list terminating in a list of integers.

9.2.1-B A Combining Descriptor for Variables

Another significant effort would be to extend the descriptor language with a *BOTH form, whose first component would be a (normal) type variable and whose second component would be a descriptor. Any value satisfying the form would need to satisfy both the variable (i.e., the binding of the type variable would need to equal the value) and the descriptor. Such a construct would obviate the need for the checker's schizophrenic strategy of carrying both abstract and concrete type alists and generating both minimal and maximal segments. It would also remove the need for the *SUBST form and the restrictions result component in the inference algorithm's unification procedure. But we should not underestimate the possible difficulty of implementing such a construct, even though its semantics in an INTERP-SIMPLE setting are straightforward. This construct was considered during the development of the inference algorithm, and the difficulties it presented were simply too challenging to face simultaneously with the other problems confronted in the developmental prototype.

9.2.1-C Support for Relations in Guards

The inability to use disjuntive forms like

```
(OR (SYMBOLP X) (SYMBOL-ALISTP Y))
```

in type predicates resulted in a limitation in the power of the inference algorithm. The fundamental problem is that the linear representation of a type alist or a guard descriptor does not allow for the expression of a relation like the one above. A potentially worthwhile direction of inquiry would be to study a different representation of type alists and guard descriptors, perhaps employing a clausal form (disjunctive normal form, for instance). This would allow for freer use of negation of a type alist and enlarge the class of forms which could be used to generate complete guard descriptors. Because of the pervasiveness of guard descriptors and type alists, it is also an issue which could have root and branch impact on the entire system, and careful study would be required to measure the cost-benefit tradeoffs.

A potential benefit to the system is that this alternate representation might allow a guard expression or type predicate to be cleanly factored into type-specific components and other residual predicates, so that type-specific problems (like type inference) could be cleanly handled by the type system, and other problems by some other mechanism. This suggests an approach to guard verification, blending the capabilities of the type system and other methods, so that the support for guard verification could be generalized to arbitrary predicates on parameters.

9.2.1-D Generating Guards

The system as designed seeks to verify guards supplied by the user. It would be interesting to investigate how the tool could be put in a mode in which it *generates* guards instead. That is, when a new function is submitted (without a guard), the system could attempt to combine context information in the function body with the guard requirements from all the functions called in the body to generate a guard expression which, when holding for the parameters, would guarantee the absence of guard violations in the body. This would involve only the addition of a new heuristic algorithm, since the process of signature validation would be identical to that used now.

9.2.1-E Unification Involving Variables

The unification algorithm in the checker, with its merging of substitutions and <descriptor>/<subst> result form, is a much more satisfying and tractable treatment of variables than that of the inference algorithm's unification routine, which combines *SUBST annotations and an extra "restrictions" component in its result. Short of implementing the *BOTH descriptor described above, it would be worthwhile to determine if unification in the inference algorithm could be re-implemented to use the checker's strategy. The challenge would be to see if the checker's strategy could support the extra descriptor forms manipulated in the inference algorithm.

9.2.1-F Discovering Recursive Structures in Arguments

As was mentioned with the STANDARD-CHAR-LISTP example in the previous chapter, as well as in the discussion of the SAD-BUT-TRUE-LISTP example in Section 4.4.7, the *FIX heuristic is only constructed on the right hand, or result, side of a segment, when it would sometimes be appropriate to construct it on the left hand, or argument, component as well. We believe the approach could be generalized, so that *FIX descriptors would come into play when functions decompose their arguments recursively, as well as when they construct their results recursively. It is conceivable that this approach could render the special algorithm for the treatment of recognizer functions unnecessary. But in any case it is likely that for functions whose guard does not specify a recursively typed argument, but whose body nevertheless unwinds an argument recursively, we could see a vast improvement in the quality of signatures produced by the inference algorithm.

9.2.1-G Employing Pure Recursive Descent

The tabular infrastructure approach used in the DERIVE-EQUATIONS/SOLVE-EQUATIONS components of the inference algorithm was an artifact of an early and ill-conceived notion that reasonably accurate signatures could be computed without performing an iterative approximation, given the right supporting data structure. In retrospect, a straightforward recursive descent functional approach would be the cleanest formulation of the algorithm, and re-coding in this style would yield a dividend of clarity and likely performance improvement.

9.2.1-H Tuning the Checker Algorithm

The run-time performance of the checker leaves much to be desired. Given the consistent choice of conceptual simplicity over computational efficiency, this is to be expected. But if computational overhead becomes a serious deterrent to the employment of the type system, a good place to look for improvement would be in the checker. It is quite possible that some simple steps could be taken to contain the explosion of cases without involving extensive formal complication, perhaps by merging segments produced in intermediate results. This would likely require a pragmatic approach, utilizing extensive testing to see if such steps would result in a higher rejection rate by the checker.

9.2.1-I Relaxing the Style Constraint Guards

There is some inconvenience to the user in the checker's insistence that a guard must be a conjunction of recognizer calls on distinct parameters in order to result in complete descriptors. That the inference algorithm has more liberal criteria is an indication that the checker could be more lenient. Again, some extra pain in the formal analysis might result in a relaxation of the checker's requirement. Though the CADR/CADR-GUARD example in the preceding chapter illustrates there is likely no true loss of functionality with the current state of affairs, there is obviously some extra overhead for the user to conform to the checker's requirement which could be avoided.

9.2.1-J Miscellany

This section represents a potpourri of minor enhancements which have been previously mentioned and which would probably be worthwhile.

Just as the system notices when a recognizer is being defined, it could also notice when a destructor function is introduced. Thus, user abstraction functions whose bodies are essentially nests of CAR and CDR calls could be treated uniformly with CAR and CDR, which receive special treatment in some corners of the algorithm. (See the discussion of "feedback" in Section 4.4.7.)

NTH is a Lisp function commonly used to access elements of lengthy lists. Unfortunately, the signature derived for NTH by the inference algorithm allows virtually no useful information to appear in the result type. This suggests a need for special treatment, as mentioned in the previous chapter. Calls to NTH, where the numeric argument is a constant, could be expanded in a prepass algorithm to the proper nest of calls to CAR and CDR, thus allowing the type variable mechanism to handle the transferrence of information from argument to result. UPDATE-NTH could be treated similarly.

Occasionally, one would like to define a recognizer function which, for example, specifies a list of some specific length N, but using the form (EQUAL (LENGTH FOO) N) disqualifies the function from treatment as a recognizer. To preserve the functionality, the user must construct an ugly CONSP nest to convey the same idea. This is another form which could perhaps be transformed in a prepass so that the ugly nest could be constructed reliably and without imposition on the user.

The ARRAY1P test suite example in the previous chapter illustrated a case where the checker failed to factor a segment with an *OR result type into separate segments for each disjunct. This resulted in a failure of the segment containment step. A trivial patch would be to perform this factorization. To do so with minimal loss of efficiency, we might perform this factorization only after containment actually failed. This kind of factorization could possibly mitigate against the loss of functionality hypothesized above as a result of efficiency improvements in the checker.

As *REC descriptors are currently defined, the top level form in the body may be a *CONS, as in the descriptor:

```
(*REC PROPER-CONSP
      (*CONS *UNIVERSAL (*OR $NIL (*RECUR PROPER-CONSP))))
```

In numerous places, extra case analysis is required to consider this kind of form, and clarity in the discussion of *REC descriptors can be reduced. It might be worthwhile to require that the top level form in a *REC descriptor be an *OR. All that would likely be necessary to accomplish this is a canonicalization applied to newly formed *REC descriptors which would bring an outer *CONS out of the *REC body. The PROPER-CONSP descriptor above would thus be transformed to:

```
(*CONS *UNIVERSAL
       (*REC !REC1 (*OR $NIL (*CONS *UNIVERSAL (*RECUR !REC1)))))
```

9.2.2 Extensions to the Supported Lisp Subset

A number of extensions to the base language would be relatively easy to implement using the techniques in the current system.

9.2.2-A Global Constants

As we have mentioned, support for global constants would be trivial to implement. In fact, constants are currently supported in the implementation of the inference algorithm. As a matter of expediency, and because there seemed to be little of formal interest in them, constants were not supported in the checker. Support of constants was simply disabled at the top level loop.

9.2.2-B LET

Though not quite so simple, support for LET was initially provided in the inference algorithm, for the case where the LET bindings contained no recursive calls to the function in which they appeared. The treatment was simply to compute the type for each LET-bound variable and push it onto the type-alist for consideration in the LET body. To simplify the task presented to the initial implementation of the checker, we again suppressed this capability. This seemed in keeping with the notion that macros are not supported. Nevertheless, LET seems like an entirely feasible endeavor, if restricted as above. A general treatment of LET in the presence of recursion gets into the realm of mutually recursive functions, which are discussed below.

9.2.2-C Multiple-Value Functions

Multiple-value functions can be quite useful in a purely applicative language, and there seems to be no good reason why they could not be easily supported within the inference system. One way of doing so would be to interpret VALUES as we would LIST and MULTIPLE-VALUE-BIND as we would a destructuring LET. This would likely require little extension to the system other than to treat these forms in a prepass.

9.2.2-D Mutually Recursive Functions

A very interesting problem would be to handle type inference for functions whose definitions are mutually recursive. This would require the nest of all such functions to be submitted as a collection. Intuitively, we can imagine the fundamental approach being to extend the iterative stabilization strategy in the following manner. Currently, we iterate over the body of a new function, with each iteration establishing a new set of working segments, until the collection of working segments stabilizes. To handle mutual recursion, we imagine starting the analysis by assigning an empty set of segments to each function, making one pass over all the functions, replacing its segments with the new ones, and then repeating this process until the working segments for all the functions have stabilized. The checker could then treat the functions one at a time. But the proof of Lemma TC-SIGNATURE-OK would need to be modified to support the inductive step of adding a whole collection of new signatures at once. Without having devoted any serious thought to this prospect, we can only say it would require very careful formal attention.

9.2.2-E Arrays and Structures

Extension to arrays and structures defined by defstruct may be problematic, since we assume our Lisp subset is purely applicative. Common Lisp's standard update mechanism for arrays and structures is SETF, which is a destructive operation. Tracking the side effects of destructive operations is beyond the current scope and spirit of this effort. A purely applicative model of arrays and structures, however, would lend itself well to treatment within the inference system.

9.3 Symbiotic Relation to Other Tools

One purpose for which type inference has been employed in other systems is to supply inferred type declarations to the compiler. The improvement of run-time speed of code generated with prolific type declarations is the apparent *raison d'etre* for the TICL type inference system [Ma 90] and the Nimble Type Inferencer [Baker 90].

9.3.1 Assisting a Compiler

The system described in this report could be easily modified to generate declarations in this style. A simple mapping could be maintained in the database associating descriptors with their recognizer functions. Whenever the type of any form is determined to correspond to some variable-free type descriptor, a THE declaration [Steele 90] could be wrapped around the form. Similarly, DECLARE forms could be inserted to correspond to information derived from guard expressions. Thus, a function

```
(DEFUN RAW-FN (X Y)
  (DECLARE (XARGS :GUARD (IF (INTEGER-LISTP X) (INTEGERP Y) NIL)))
  (IF (EQUAL Y 0)
      (CAR X)
      (RAW-FN (CDR X) (MINUS Y 1))))
```

could be annotated by the tool as follows:

```
(DEFUN RAW-FN (X Y)
  (DECLARE (XARGS :GUARD (IF (INTEGER-LISTP X) (INTEGERP Y) NIL)))
  (DECLARE (TYPE INTEGER-LISTP X))
  (DECLARE (TYPE INTEGER Y))
  (IF (EQUAL Y 0)
      (THE (OR INTEGERP NULL) (CAR (THE INTEGER-LISTP X)))
      (THE (OR INTEGERP NULL)
           (RAW-FN (THE INTEGER-LISTP (CDR X))
                   (THE INTEGER (MINUS Y 1))))))
```

If for some target compiler, an INTEGER-LISTP is not useful, any such declaration could be toned down to (OR CONSP NULL).

While our system could likely produce improved efficiency from such an endeavor, the Nimble and TICL systems already seem to be quite adept at producing declarations for the types of atomic objects and support a richer collection of such types than we do. But these systems appear to provide little support for structured objects, which is the *forte* of our system. One could easily imagine a partnership between some such system and ours, whereby the one could produce type declarations for atomic objects, and our system would be responsible for annotating forms representing structured objects and objects derived from structures.

9.3.2 Teaming with a Theorem Prover

Perhaps the most fruitful potential placement of our type system would be to embed it within a more general automated reasoning system for Lisp functions. In particular, if our inference tool could work in tandem with a more general purpose theorem prover to perform tasks like guard verification, it is likely that such problems could be factored so that the inference tool could handle type-specific questions, and problems outside the type domain could be handled by the prover. Many typical problems presented to a theorem prover for computable functions have type-related components, and the type inference system could potentially operate as a specialized algorithm within a prover, just as a rewriter or a specialized decision procedure might also operate. Tailored as it is to recursive functions, our type inference system could be expected to make a significant contribution. On balance, we believe this approach could increase the capability of a prover to resolve type-related problems without user intervention.

Conversely, access to assistance from a general theorem prover could allow the type system to solve a greater number of problems in its own domain. If, for example, the prover could assist the type inference system in verifying guards containing predicates which lie partly outside the type domain, we could support much greater generality in the kinds of guards one could use and still receive certified signatures from the system. The class of functions whose signatures could be certified sound by the checker would grow, and the tool would be more useful as a result. In short, we believe there is fertile ground for symbiosis between the type inference system and a general purpose prover.

Appendix A
Signatures of Functions in the Initial State

This appendix illustrates the function signatures for all the functions in the initial system state. Each (*FREE-TYPE-VAR n) form is a placeholder form representing a type variable, which will be instantiated with a fresh variable when the signature is invoked for use. Multiple occurrences within a segment of the number associated with a *FREE-TYPE-VAR marker indicate that the same variable will appear in all such positions.

```
Function: CONS
Guard computed by the tool:
   (*UNIVERSAL *UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL *UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*FREE-TYPE-VAR 1.) (*FREE-TYPE-VAR 2.))
  -> (*CONS (*FREE-TYPE-VAR 1.) (*FREE-TYPE-VAR 2.)))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: CAR
Guard computed by the tool:
   ((*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*CONS (*FREE-TYPE-VAR 1.) *UNIVERSAL)) -> (*FREE-TYPE-VAR 1.))
 (($NIL) -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: CDR
Guard computed by the tool:
   ((*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL)))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*CONS *UNIVERSAL (*FREE-TYPE-VAR 1.))) -> (*FREE-TYPE-VAR 1.))
 (($NIL) -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
```

```
Signature is certified sound: T


Function: BINARY-+
Guard computed by the tool:
   ((*OR $INTEGER $NON-INTEGER-RATIONAL)
    (*OR $INTEGER $NON-INTEGER-RATIONAL))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $INTEGER $NON-INTEGER-RATIONAL)
   (*OR $INTEGER $NON-INTEGER-RATIONAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($INTEGER $INTEGER) -> $INTEGER)
 (($INTEGER $NON-INTEGER-RATIONAL) -> $NON-INTEGER-RATIONAL)
 (($NON-INTEGER-RATIONAL $INTEGER) -> $NON-INTEGER-RATIONAL)
 (($NON-INTEGER-RATIONAL $NON-INTEGER-RATIONAL)
  -> (*OR $INTEGER $NON-INTEGER-RATIONAL))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: UNARY--
Guard computed by the tool:
   ((*OR $INTEGER $NON-INTEGER-RATIONAL))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $INTEGER $NON-INTEGER-RATIONAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($INTEGER) -> $INTEGER)
 (($NON-INTEGER-RATIONAL) -> $NON-INTEGER-RATIONAL)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: BINARY-*
Guard computed by the tool:
   ((*OR $INTEGER $NON-INTEGER-RATIONAL)
    (*OR $INTEGER $NON-INTEGER-RATIONAL))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $INTEGER $NON-INTEGER-RATIONAL)
   (*OR $INTEGER $NON-INTEGER-RATIONAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
((($INTEGER $INTEGER) -> $INTEGER)
 (($INTEGER $NON-INTEGER-RATIONAL)
  -> (*OR $INTEGER $NON-INTEGER-RATIONAL))
 (($NON-INTEGER-RATIONAL $INTEGER)
   -> (*OR $INTEGER $NON-INTEGER-RATIONAL))
 (($NON-INTEGER-RATIONAL $NON-INTEGER-RATIONAL)
  -> (*OR $INTEGER $NON-INTEGER-RATIONAL))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
```

```
Signature is certified sound: T


Function: UNARY-/
Guard computed by the tool:
   ((*OR $INTEGER $NON-INTEGER-RATIONAL))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $INTEGER $NON-INTEGER-RATIONAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $INTEGER $NON-INTEGER-RATIONAL))
  -> (*OR $INTEGER $NON-INTEGER-RATIONAL))
TC segments contained in Segments: T
Recognizer descriptor:
  NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: <
Guard computed by the tool:
   ((*OR $INTEGER $NON-INTEGER-RATIONAL)
    (*OR $INTEGER $NON-INTEGER-RATIONAL))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $INTEGER $NON-INTEGER-RATIONAL)
   (*OR $INTEGER $NON-INTEGER-RATIONAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $INTEGER $NON-INTEGER-RATIONAL)
   (*OR $INTEGER $NON-INTEGER-RATIONAL))
  -> (*OR $NIL $T))
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: EQUAL
Guard computed by the tool:
   (*UNIVERSAL *UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL *UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($NIL $NIL) -> $T)
 (($T $T) -> $T)
 (((*FREE-TYPE-VAR 1.) (*FREE-TYPE-VAR 1.)) -> $T)
 (($NIL
   (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
  -> $NIL)
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
   $NIL)
  -> $NIL)
 (($T
   (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
```

```
        $STRING (*CONS *UNIVERSAL *UNIVERSAL)))
  -> $NIL)
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING (*CONS *UNIVERSAL *UNIVERSAL))
   (*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
  -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: CONSP
Guard computed by the tool:
   (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*CONS *UNIVERSAL *UNIVERSAL)) -> $T)
 (((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $STRING $T))
  -> $NIL))
TC segments contained in Segments: T
Recognizer descriptor: (*CONS *UNIVERSAL *UNIVERSAL)
TC validates recognizer: T
Signature is certified sound: T


Function: INTEGERP
Guard computed by the tool:
   (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($INTEGER) -> $T)
 (((*OR $CHARACTER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
  -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: $INTEGER
TC validates recognizer: T
Signature is certified sound: T


Function: RATIONALP
Guard computed by the tool:
   (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $INTEGER $NON-INTEGER-RATIONAL)) -> $T)
 (((*OR $CHARACTER $NIL $NON-T-NIL-SYMBOL $STRING $T
        (*CONS *UNIVERSAL *UNIVERSAL)))
```

```
  -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: (*OR $INTEGER $NON-INTEGER-RATIONAL)
TC validates recognizer: T
Signature is certified sound: T


Function: STRINGP
Guard computed by the tool:
   (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($STRING) -> $T)
 (((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
        $NON-T-NIL-SYMBOL $T (*CONS *UNIVERSAL *UNIVERSAL)))
  -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: $STRING
TC validates recognizer: T
Signature is certified sound: T


Function: CHARACTERP
Guard computed by the tool:
   (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($CHARACTER) -> $T)
 (((*OR $INTEGER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
  -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: $CHARACTER
TC validates recognizer: T
Signature is certified sound: T


Function: SYMBOLP
Guard computed by the tool:
   (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
  (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $NIL $NON-T-NIL-SYMBOL $T)) -> $T)
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $STRING
        (*CONS *UNIVERSAL *UNIVERSAL)))
  -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor: (*OR $NIL $NON-T-NIL-SYMBOL $T)
TC validates recognizer: T
Signature is certified sound: T
```

```
Function: NULL
Guard computed by the tool:
 (*UNIVERSAL)
Guard complete: T
All called functions complete: T
TC Guard:
 (*UNIVERSAL)
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (($NIL) -> $T)
 (((*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
   -> $NIL)
TC segments contained in Segments: T
Recognizer descriptor:
  $NIL
TC validates recognizer: T
Signature is certified sound: T


Function: DENOMINATOR
Guard computed by the tool:
   ((*OR $INTEGER $NON-INTEGER-RATIONAL))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $INTEGER $NON-INTEGER-RATIONAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $INTEGER $NON-INTEGER-RATIONAL)) -> $INTEGER)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: NUMERATOR
Guard computed by the tool:
   ((*OR $INTEGER $NON-INTEGER-RATIONAL))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $INTEGER $NON-INTEGER-RATIONAL))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $INTEGER $NON-INTEGER-RATIONAL)) -> $INTEGER)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: SYMBOL-NAME
Guard computed by the tool:
   ((*OR $NIL $NON-T-NIL-SYMBOL $T))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $NIL $NON-T-NIL-SYMBOL $T))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
```

```
 (((*OR $NIL $NON-T-NIL-SYMBOL $T)) -> $STRING)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T


Function: SYMBOL-PACKAGE-NAME
Guard computed by the tool:
   ((*OR $NIL $NON-T-NIL-SYMBOL $T))
Guard complete: T
All called functions complete: T
TC Guard:
  ((*OR $NIL $NON-T-NIL-SYMBOL $T))
TC Guard complete: T
TC All called functions complete: T
TC Guard Replaced by Tool Guard: NIL
Segments:
 (((*OR $NIL $NON-T-NIL-SYMBOL $T)) -> $STRING)
TC segments contained in Segments: T
Recognizer descriptor: NIL
TC validates recognizer: NIL
Signature is certified sound: T
```

Appendix B
Proofs of Selected Lemmas

B.1 Proof of TC-PREPASS-OK

TC-PREPASS normalizes IF expressions by ensuring that the test expressions return only T or NIL. In any case where it cannot determine that an IF test evaluates only to T or NIL, it encapsulates the test form in two calls to the NULL function. It also treats the special cases where the test is (or can be easily reduced to) T or NIL, so that the entire IF form is reduced to its second or third argument, as appropriate. The definition of TC-PREPASS and its subsidiary function TC-PREPASS-IF-PRED appears in Appendix G.6.

The property we wish to prove is that for any non-negative integer clock, if FS is a valid representation of our world of Lisp functions, evaluating (TC-PREPASS FORM FS) in any environment yields the same result as evaluating FORM in the environment.

Intuitively, the argument is very simple. The TC-PREPASS algorithm performs only the two transformations described above, and it performs them throughout the form. First let us consider the transformation of wrapping two calls to NULL around an IF test. Since NULL is a SUBR, once the argument is evaluated, checking the guard and returning a value require no clock time. So a BREAK-OUT-OF-TIME is produced on E’s evaluation of a call to NULL if and only if a BREAK-OUT-OF-TIME is produced while evaluating the parameter. Similarly for a BREAK-GUARD-VIOLATION, since the guard to NULL is T. If evaluating an IF test does not yield a break, then the only significance of the result is whether it is NIL or non-NIL.

```
(NULL (NULL NIL)) = (NULL T) = NIL
 and
(NULL (NULL <any non-NIL value>)) = (NULL NIL) = T
```

Thus, the evaluator will receive the same direction from the test, regardless of whether it is encased in the two calls to NULL.

The other transformations performed by TC-PREPASS are simplifications. In the case where the (possibly transformed) IF test is NIL, the IF is replaced by its else arm, and where the IF test is T, the IF is replaced by its then arm. Consider the case where the IF test is NIL. The argument for the T case will be symmetric, with the only difference being consideration of the then arm rather than the else.

If CLOCK < 1, evaluation of either the original form or the transformed one produces a BREAK-OUT-OF-TIME. Now consider the case where CLOCK ≥ 1. Evaluating NIL does not otherwise involve the clock, and evaluating IF does not involve ticking the clock down on the recursive calls of E to evaluate the test or either of the result arms. Furthermore, IF is lazy, in that it does not evaluate the arm which will not contribute to the result value. Thus, the clock on entry to the transformed form (i.e., the else form), will be the same as it would be on entry to the else form in the original IF, and therefore a form transformed in this manner will evaluate to a BREAK-OUT-OF-TIME if and only if the original IF form evaluates to a BREAK-OUT-OF-TIME. A similar argument holds for a guard violation, since there is no guard check on an IF or on NIL. Since the value returned by an IF form whose test is NIL is just the value returned by the else form, the value returned will be the same if the IF is replaced by its else form.

The preceding argument should suffice for a proof of TC-PREPASS-OK and spare the reader from delving into what lies below. But for the intrepid, the proofs below involve essentially an exhaustive examination of cases arising from one-level expansions of the definitions of E, TC-PREPASS, and TC-PREPASS-IF-PRED, down to recursive calls. The functions TC-PREPASS and TC-PREPASS-IF-PRED (the latter specifically treats IF tests) are mutually recursive, and each is specified by its own lemma. Hence, the proof of our evaluation property involves a proof of the conjunction of Lemma TC-PREPASS-OK and Lemma TC-PREPASS-IF-PRED-OK. We present this composite proof as a proof of each individual lemma under a joint computational induction scheme. The induction is on the number of calls to TC-PREPASS and TC-PREPASS-IF-PRED necessary to return a result from the top level call to TC-PREPASS. This will enable us to use each lemma as an inductive assertion on each subsidiary call.

```
Lemma TC-PREPASS-OK
 For any list of function signatures fs, Lisp world world,
 non-negative integer clock, Lisp form form, and binding
 environment env,

  (valid-fs fs world clock)
  =>
  (equal (E form env world clock)
         (E (tc-prepass-form form fs) env world clock))


 Lemma TC-PREPASS-IF-PRED-OK
  For any list of function signatures fs, Lisp world world, non-negative
  integer clock, Lisp IF form test, and binding environment env,

  (valid-fs fs world clock)
  =>
  (if (break-out-of-timep (E form env world clock))
      (break-out-of-timep
        (E (tc-prepass-if-pred form fs) env world clock))
      (if (break-guard-violationp (E form env world clock))
          (break-guard-violationp
            (E (tc-prepass-if-pred form fs) env world clock))
          (if (equal (E form env world clock) nil)
              (equal (E (tc-prepass-if-pred form fs) env world clock)
                     nil)
              (equal (E (tc-prepass-if-pred form fs) env world clock)
                     t))))
```

We present the proof of Lemma TC-PREPASS-OK first.

```
Proof of Lemma TC-PREPASS-OK

 First let us take care of the easy case where clock < 1.  In this
 case, by the definition of E, for any form, both
  (break-out-of-timep form env world clock)
 and
  (break-out-of-timep (tc-prepass form fs) env world clock)
 hold.

 For the case where clock ≥ 1, we will do a case analysis on
 the structure of form.

 Case 1: form is an atom

  By definition, (tc-prepass form fs) = form
  Trivially true.

 Case 2: form is a quoted form

  By definition, (tc-prepass form fs) = form
  Trivially true.
```

Case 3: form is a function call (f form$_1$ .. form$_n$)

(tc-prepass (f form$_1$ .. form$_n$) fs)
= by definition
(f (tc-prepass form$_1$ fs) .. (tc-prepass form$_n$ fs))

Therefore, we need to prove

(valid-fs fs world clock)
=>
(equal (E (f form$_1$ .. form$_n$) env world clock)
 (E (f (tc-prepass form$_1$ fs) .. (tc-prepass form$_n$ fs))
 env world clock))

In the expansion of both calls to E, the first action is to evaluate the actual parameters. I.e., for the first call to E, we first evaluate (E form$_1$ env world clock), and if it does not break, then (E form$_2$ env world clock), etc. until we have evaluated all n forms. Likewise for the prepassed forms.

We inductively assume our lemma is true. I.e., for all i in [1..n]

(valid-fs fs world clock)
=>
(equal (E form$_i$ env world clock)
 (E (tc-prepass form$_i$ fs) env world clock))

Thus, if either form resulted in either a break-out-of-time or a break-guard-violation, so did the other. By the definition of E, if evaluating an actual parameter results in a break, the value of E for the function call is the same break. So if any break occurs while evaluating the actual parameters, we have established our result.

Next, E finishes the evaluation of the function call, using the the values computed for the actual parameters. This "finishing" operation has the notation $f^{world,clock}$. So the expansion of both calls to E gives us:

(equal ($f^{world,clock}$((E form$_1$ env world clock)
 ..
 (E form$_n$ env world clock)))
 ($f^{world,clock}$((E (tc-prepass form$_1$ fs) env world clock)
 ..
 (E (tc-prepass form$_1$ fs) env world clock))))

Again we use our inductive assumption. Applying it for each i in [1..n] reduces the equality to T.

Case 4: form is an IF of the form (if test then-form else-form)

We can inductively assume the following (from TC-PREPASS-OK):

(E test env world clock) = (E (tc-prepass test fs) env world clock)
(E then-form env world clock)
 = (E (tc-prepass then-form fs) env world clock)
(E else-form env world clock)
 = (E (tc-prepass else-form fs) env world clock)

By definition of tc-prepass,

```
(tc-prepass (if test then-form else-form) fs)
=
(let ((prepassed-pred
        (tc-prepass-if-pred (tc-prepass test fs) fs)))
  (if (null prepassed-pred)
      (tc-prepass else-form fs)
```

```
       (if (equal prepassed-pred t)
           (tc-prepass then-form fs)
           (list 'if
                 prepassed-pred
                 (tc-prepass then-form fs)
                 (tc-prepass else-form fs)))))

Our goal is:

(valid-fs fs world clock)
=>
(equal
 (E (if test then-form else-form) env world clock)
 (E (tc-prepass (if test then-form else-form) fs) env world clock))

We will do a case analysis on the value returned by
(E test env world clock), with the cases being break-out-of-timep,
break-guard-violation, nil, or some non-NIL Lisp value.

Case 4.1  (break-out-of-timep (E test env world clock))

 In this case, by the definition of E,
 (break-out-of-timep
   (E (if test then-form else-form) env world clock))

 Using our inductive assumption and the case assumption gives us

 (break-out-of-timep (E (tc-prepass test fs) env world clock))

 Now use Lemma TC-PREPASS-IF-PRED-OK, instantiated with
 form = (tc-prepass test fs).  This gives us:

 (break-out-of-timep
   (E (tc-prepass-if-pred (tc-prepass test fs) fs) env world clock))

 We need to establish

 (break-out-of-timep
   (E (let ((prepassed-pred
              (tc-prepass-if-pred (tc-prepass test fs) fs)))
        (if (null prepassed-pred)
            (tc-prepass else-form fs)
            (if (equal prepassed-pred t)
                (tc-prepass then-form fs)
                (list 'if
                      prepassed-pred
                      (tc-prepass then-form fs)
                      (tc-prepass else-form fs)))))
      env world clock))

 We will do a case analysis based on the if structure above.

 Case 4.1.1  (tc-prepass-if-pred (tc-prepass test fs) fs) = nil

  Thus,
  (tc-prepass (if test then-form else-form) fs)
    = (tc-prepass else-form fs)

  (break-out-of-timep nil env world clock) = t only when
  (< clock 1).  This is not consistent with our case assumption.

 Case 4.1.2  (tc-prepass-if-pred (tc-prepass test fs) fs) = t
  As with 4.1.1

 Case 4.1.3  (tc-prepass-if-pred (tc-prepass test fs) fs) = neither t
               nor nil
  Thus,
  (tc-prepass (if test then-form else-form) fs)
   =
```

```
     (if (tc-prepass-if-pred (tc-prepass test fs) fs)
         (tc-prepass then-form fs)
         (tc-prepass else-form fs))
     (Note: By this, we mean a Lisp IF form whose test form is
      (tc-prepass-if-pred (tc-prepass test fs) fs) and whose then
      and else forms are the prepassed then-form and else-form,
      respectively.)

     Since we already know
     (break-out-of-timep
       (E (tc-prepass-if-pred (tc-prepass test fs) fs)
          env world clock))
     then
     (break-out-of-timep
       (E (if (tc-prepass-if-pred (tc-prepass test fs) fs)
              (tc-prepass then-form fs)
              (tc-prepass else-form fs))
          env world clock))
     either because the clock has already run down or because
     E first evaluates the if test, which breaks out of time.

   Case 4.2   (break-guard-violationp (E test env world clock))

    The argument goes just like that of Case 4.1, except that we do
    not need to mention the possibility of the clock running down
    in the final sub-case.

   Case 4.3  (E test env world clock) = nil

    By our inductive hypothesis,

H2   (E (tc-prepass test fs) env world clock) = nil

    Now use Lemma TC-PREPASS-IF-PRED-OK, instantiating with fs = fs,
    world = world, clock = clock, and form = (tc-prepass test fs).
    H2 lets us reduce the conclusion to

H3   (equal (E (tc-prepass-if-pred (tc-prepass test fs) fs)
               env world clock)
            nil)

   (and
H1  (valid-fs fs world clock)
H2  (E (tc-prepass test fs) env world clock) = nil
H3  (equal (E (tc-prepass-if-pred (tc-prepass test fs) fs)
              env world clock)
           nil) )
   =>
   (equal (E (if test then-form else-form) env world clock)
          (E (tc-prepass (if test then-form else-form) fs)
             env world clock))

    Expanding both calls of E (and using our case assumption in the
    first call), we get the conclusion

    (equal (E else-form env world clock)
           (E (let ((prepassed-pred
                       (tc-prepass-if-pred (tc-prepass test fs) fs)))
                (if (null prepassed-pred)
                    (tc-prepass else-form fs)
                    (if (equal prepassed-pred t)
                        (tc-prepass then-form fs)
                        (list 'if
                              prepassed-pred
                              (tc-prepass then-form fs)
                              (tc-prepass else-form fs)))))
              env world clock))

    Again, we will do a case analysis based on the if structure above.
```

```
    Case 4.3.1  (tc-prepass-if-pred (tc-prepass test fs) fs) = nil

     This gives the conclusion

     (equal (E else-form env world clock)
            (E (tc-prepass else-form fs) env world clock))

     which by our inductive hypothesis reduces to true.

    Case 4.3.2  (tc-prepass-if-pred (tc-prepass test fs) fs) = t

     This contradicts H3, since
      (E (tc-prepass-if-pred (tc-prepass test fs) fs) env world clock)
      =       by the case assumption
      (E t env world clock)
      = t
      but H3 says it equals nil.

    Case 4.3.3  (tc-prepass-if-pred (tc-prepass test fs) fs) equal some
                Lisp value which is neither t or nil

     Then by definition of tc-prepass,
     (tc-prepass (if test then-form else-form) fs)
       =
     (if (tc-prepass-if-pred (tc-prepass test fs) fs)
         (tc-prepass then-form fs)
         (tc-prepass else-form fs))

     By definition of E,
     (E (if (tc-prepass-if-pred (tc-prepass test fs) fs)
            (tc-prepass then-form fs)
            (tc-prepass else-form fs))
        env world clock)
     = (if (not (equal (E (tc-prepass-if-pred (tc-prepass test fs) fs)
                          env world clock)
                       nil))
           (E (tc-prepass then-form fs) env world clock)
           (E (tc-prepass else-form fs) env world clock))

     Instantiating Lemma TC-PREPASS-IF-PRED-OK with
     form = (tc-prepass test fs), and using our case assumption
     (E test env world clock) = nil, we establish
       (equal (E (tc-prepass-if-pred (tc-prepass test fs) fs)
                 env world clock)
              nil)
     Thus, the test in the if form just above simplifies to nil, and
     the whole if form simplifies to
       (E (tc-prepass else-form fs) env world clock)
     Now given our case assumption, the original left hand side of
     the conclusion simplifies to
       (E else-form env world clock)
     Use our inductive assumption on else-form to equate these two
     forms, establishing the conclusion.

   Case 4.4  (E test env world clock) = some non-nil Lisp value

    This case is symmetric with Case 4, switching the roles of nil
    and non-nil.

  This completes an exhaustive case analysis.  QED.
```

Now for the proof of Lemma TC-PREPASS-IF-PRED-OK.

Proof of TC-PREPASS-IF-PRED-OK

```
 First let us take care of the easy case where clock < 1.  In this
 case, by the definition of E, for any form, both
```

```
 (break-out-of-timep form env world clock)
and
 (break-out-of-timep (tc-prepass-if-pred form fs) env world clock)
hold.

For the case where clock ≥ 1, we will do a case analysis on
the structure of form.

Case 1: form = nil

 By definition of td-prepass-if-pred,
  (tc-prepass-if-pred form fs) = nil
 Trivial.

Case 2: form is an integer, T, a rational, a character, or a string

 By definition of tc-prepass-if-pred,
   (tc-prepass-if-pred form fs) = t
 By definition of E, (E form env world clock) = form
 (E t env world clock) = t

Case 3: form is a cons whose car is the atom quote and whose cadr
        is the atom nil.

 By definition of tc-prepass-if-pred,
   (tc-prepass-if-pred form fs) = nil
 By definition of E,
   (E (quote nil) env world clock) = nil
   (E nil env world clock) = nil

Case 4: form is a cons whose car is the atom quote and whose cadr
        is not nil.

 By definition of tc-prepass-if-pred,
   (tc-prepass-if-pred form fs) = t
 By definition of E,
   (E (quote form) env world clock) = form
   (E t env world clock) = t

Case 5: form is a variable identifier v

 By the definition of tc-prepass-if-pred,
  (tc-prepass-if-pred v fs) = (null (null v))
 By the definition of E, (since variable evaluation does not run the
 clock down),
   (E v env world clock) = (cdr (assoc v env))
 What about (E (null (null v)) env world clock)?
 Null has no guard, and since it is a subr, it does not run the clock
 down.  So since we know clock > 1, we will not get a break.

 Case 5.1  (equal (cdr (assoc v b)) nil)

  Our goal simplifies to

  (valid-fs fs world clock)
  =>
  (equal (E (null (null nil)) env world clock) nil)

  The null function returns T if its argument is nil, and nil if its
  argument is anything but nil, so (null (null nil)) = nil.
  Since (E nil env world clock) = nil, our goal is established.

 Case 5.2  (not (equal (cdr (assoc v b)) nil))

  The goal simplifies to

  (valid-fs fs world clock)
  =>
  (equal (E (null (null v)) env world clock) t)
```

```
  The null function returns t if its argument is nil, and nil
  if its argument is anything but nil, so (null (null v)) = t.
  Since (E t env world clock) = t, our goal is established.

Case 6: form is an if, (if test then-form else-form)

 By definition of E,
  (E (if test then-form else-form) env world clock)
  =
  (if (equal (E test env world clock) nil)
      (E else-form env world clock)
      (E then-form env world clock))

 By definition of tc-prepass-if-pred and E,
  (E (tc-prepass-if-pred (if test then-form else-form) fs)
     env world clock)
   =
  (E (if (tc-prepass-if-pred test fs)
         (tc-prepass-if-pred then-form fs)
         (tc-prepass-if-pred else-form fs))
     env world clock)
   =
  (if (equal (E (tc-prepass-if-pred test fs) env world clock) nil)
      (E (tc-prepass-if-pred else-form fs) env world clock)
      (E (tc-prepass-if-pred then-form fs) env world clock))

 Our inductive hypotheses assume the lemma is true of the function
 called recursively on the subexpressions of its arguments.  Our goal
 is

 (if (break-out-of-timep
       (E (if test then-form else-form) env world clock))
     (break-out-of-timep
       (E (tc-prepass-if-pred (if test then-form else-form) fs)
          env world clock))
     (if (break-guard-violationp
           (E (if test then-form else-form) env world clock))
         (break-guard-violationp
           (E (tc-prepass-if-pred (if test then-form else-form) fs)
              env world clock))
         (if (equal (E (if test then-form else-form) env world clock)
                    nil)
             (equal
               (E (tc-prepass-if-pred (if test then-form else-form) fs)
                  env world clock)
               nil)
             (equal
               (E (tc-prepass-if-pred (if test then-form else-form) fs)
                  env world clock)
               t))))

 Consider the cases suggested by this if structure.

 Case 6.1  (break-out-of-timep (E test env world clock))
  By definition of E,
   (break-out-of-timep
     (E (if test then-form else-form) env world clock)).
  From the inductive assumption of TC-PREPASS-IF-PRED-OK instantiated
  with form = test,
   (break-out-of-timep
     (E (tc-prepass-if-pred test fs) env world clock)),
  and by E,
   (break-out-of-timep
     (E (tc-prepass-if-pred (if test then-form else-form) fs)
         env world clock))

 Case 6.2  (break-guard-violationp (E test env world clock))
  By definition of E,
   (break-guard-violationp
```

```
      (E (if test then-form else-form) env world clock))
   From the inductive assumption of TC-PREPASS-IF-PRED-OK instantiated
   with form = test,
    (break-guard-violationp
      (E (tc-prepass-if-pred test fs) env world clock))
   and by E,
    (break-guard-violationp
       (E (tc-prepass-if-pred (if test then-form else-form) fs)
          env world clock))

  Case 6.3  (equal (E test env world clock) nil)
   By definition of E,
    (E (if test then-form else-form) env world clock)
     =
    (E else-form env world clock)
   From the inductive assumption of TC-PREPASS-IF-PRED-OK instantiated
   with form = test,
    (equal (E (tc-prepass-if-pred test fs) env world clock) nil)
   and by E,
    (E (tc-prepass-if-pred (if test then-form else-form) fs)
       env world clock)
      =
    (E (tc-prepass-if-pred else-form fs) env world clock)

   Case 6.3.1  (break-out-of-timep (E else-form env world clock))
    As with Case 6.1, using the inductive assumption of
    TC-PREPASS-IF-PRED-OK instantiated with form = else-form.

   Case 6.3.2  (break-guard-violationp (E test env world clock))
    As with Case 6.2, using the inductive assumption of
    TC-PREPASS-IF-PRED-OK instantiated with form = else-form.

   Case 6.3.3  (equal (E else-form env world clock) nil)
    By definition of E,
     (E (if test then-form else-form) env world clock) = nil
    Using the inductive assumption of TC-PREPASS-IF-PRED-OK,
    instantiated with form = else-form.
     (equal (E (tc-prepass-if-pred else-form fs) env world clock) nil)
    and by definition of E,
     (E (tc-prepass-if-pred (if test then-form else-form) fs)
        env world clock)
     = nil, establishing the goal for this case

   Case 6.3.4  (E else-form env world clock) some non-nil Lisp value v
    By definition of E,
     (E (if test then-form else-form) env world clock) = v  (non-nil)
    Using the inductive assumption of TC-PREPASS-IF-PRED-OK,
    instantiated with form = else-form.
     (equal (E (tc-prepass-if-pred else-form fs) env world clock) t)
    and by definition of E
     (E (tc-prepass-if-pred (if test then-form else-form) fs)
        env world clock)
     = t, establishing the goal for this case

  Case 6.4  (E test env world clock) = some non-nil Lisp value v
    As with Case 6.3, exchanging the role of nil and non-nil and using
    the inductive assumption of TC-PREPASS-IF-PRED-OK, instantiated
    with form = then-form rather than else-form.
```

Case 7: form is a function call of form (f form$_1$ .. form$_n$)

```
   Our inductive hypotheses assume the original conjecture of
   TC-PREPASS-OK is true of the function called recursively on the
   subexpressions of its arguments.  Thus, we can assume:

   (valid-fs fs world clock)
   =>
   for all i in [1..n],
```

(equal (E form$_i$ env world clock)

```
         (E (tc-prepass form_i fs) env world clock))
```

Our goal is:

```
(valid-fs fs world clock)
=>
(if (break-out-of-timep (E form env world clock))
    (break-out-of-timep
      (E (tc-prepass-if-pred form fs) env world clock))
    (if (break-guard-violationp (E form env world clock))
        (break-guard-violationp
          (E (tc-prepass-if-pred form fs) env world clock))
        (if (equal (E form env world clock) nil)
            (equal (E (tc-prepass-if-pred form fs) env world clock)
                   nil)
            (equal (E (tc-prepass-if-pred form fs) env world clock)
                   t))))
```

By definition,

```
 (td-prepass-if-pred (f form_1 .. form_n) fs)
 =
 if f can return only T or NIL (as determined in fs)
   (f (tc-prepass form_1 fs)
      ..
      (tc-prepass form_n fs))
  else
   (null (null (f (tc-prepass form_1 fs)
                  ..
                  (tc-prepass form_n fs))))
```

(This use of fs is actually an application of the definition of valid-fs.) Note that, by the definition of tc-prepass,

```
   (f (tc-prepass form_1 fs)
      ..
      (tc-prepass form_n fs))
   = (tc-prepass (f form_1 .. form_n))
```

Case 7.1 f can return only T or NIL

Our goal simplifies to

```
 (valid-fs fs world clock)
 =>
 (if (break-out-of-timep (E form env world clock))
     (break-out-of-timep
       (E (tc-prepass form fs) env world clock))
     (if (break-guard-violationp (E form env world clock))
         (break-guard-violationp
           (E (tc-prepass form fs) env world clock))
         (if (equal (E form env world clock) nil)
             (equal (E (tc-prepass form fs) env world clock) nil)
             (equal (E (tc-prepass form fs) env world clock) t))))
```

We inductively apply Lemma TC-PREPASS-OK to get:

```
 (valid-fs fs world clock)
 =>
 (if (break-out-of-timep (E form env world clock))
     (break-out-of-timep (E form env world clock))
     (if (break-guard-violationp (E form env world clock))
         (break-guard-violationp (E form env world clock))
         (if (equal (E form env world clock) nil)
             (equal (E form env world clock) nil)
             (equal (E form env world clock) t))))
```

Since f returns T whenever it does not return NIL, this is obviously true.

```
  Case 7.2  f might return something other than T or NIL

   Our goal simplifies to

   (valid-fs fs world clock)
   =>
   (if (equal (E (f form_1 .. form_n) env world clock) nil)
       (equal (E (null (null (tc-prepass (f form_1 .. form_n) fs)))
                 env world clock)
              nil)
       (equal (E (null (null (tc-prepass (f form_1 .. form_n) fs)))
                 env world clock)
              t))

   The argument here is the same as in the previous case, but extended
   as in Case 5 to drive E down through the calls to null.  Since null
   is a subr and its guard is T, evaluating (null (null x)) will cause
   an out-of-time break if and only if evaluating x causes one, and
   will cause a guard break if and only if evaluating x causes one.
 QED.
```

B.2 The Proof of RECOGNIZER-SEGMENTS-COMPLETE

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
Lemma RECOGNIZER-SEGMENTS-COMPLETE

 Given a recognizer function R of the form
  (defun R (x)
    (declare (xargs :guard t))
    body)
 with the signature:
  Guard: (*universal)
  Segments: (((targ) -> $t)
             ((nilarg) -> $nil))
 For any list of function signatures fs including R, Lisp world world
 including R, Lisp value v, non-negative integer clock, descriptor
 arg-td, and type variable binding b,

  (and
H1  (valid-fs fs world clock)
H2  (I arg-td v b)
H3  (contained-in-interface arg-td targ)
H4  (not (break-out-of-timep (R^{world,clock}(v)))) )
   =>
   (equal (R^{world,clock}(v)) t)


Proof of Lemma RECOGNIZER-SEGMENTS-COMPLETE

 The proof uses the lemmas DUNIFY-DESCRIPTORS-INTERFACE-OK and
 CONTAINED-IN-INTERFACE-OK to open up the characterizations of
 the assumptions.

 From the definition of recognizer function (See Section 5.6),
 we know that the segments are variable-free and that targ and
 nilarg represent disjoint sets.  When TC-INFER-SIGNATURE
 validates that a function is a recognizer, it establishes this
 disjointness property by validating H5 below by computation.
 Thus, the definition of recognizer gives us the hypotheses:

H5  (equal (dunify-descriptors-interface targ nilarg) *empty)
```

```
H6  (null (gather-vars-in-descriptor targ))
H7  (null (gather-vars-in-descriptor nilarg))
```

Instantiate Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK with tda = targ, tdb = nilarg, v = v, and b = b. Substituting from H5 into the result gives us:

```
H8  (and (I targ v b) (I nilarg v b))
    =>
    (I *empty v b)
```

But since we know (I *empty v b) cannot be true, then we know

```
H8' (not (and (I targ v b)
              (I nilarg v b)))
```

We instantiate Lemma CONTAINED-IN-INTERFACE-OK, with td1 = arg-td, td2 = targ, v = v, and b = b, to give us

```
    (and (contained-in-interface arg-td targ)
         (I arg-td v b))
    =>
    For some b', (I targ v b')
```

H2 and H3 give us the antecedents to this implication, so we can use the conclusion:

```
    For some b', (I targ v b')
```

Since targ is variable-free, any binding will do for b', so we might as well use b, giving us

```
H9  (I targ v b)
```

But this, combined with H8', gives us

```
H9' (not (I nilarg v b))
```

Now use the definition of valid-fs as it applies to R, with n = 1, gtd_1 = *universal, arg_1 = v, and segments = the ones given above for R. By the definition of recognizer, we know (tc-all-called-functions-complete body fs).
Thus, we have the hypothesis

H10 for all Lisp values v,
      (and (not (break-out-of-timep ($R^{world,clock}$(v))))
           (not (null (E t ((x . v)) world clock))))
      =>
      (and
        (not (break-guard-violationp ($R^{world,clock}$(v))))
        for some binding b' of type variables to Lisp values
          covering the segments
         (or (I (targ $t) (v ($R^{world,clock}$(v))) b')
             (I (nilarg $nil) (v ($R^{world,clock}$(v))) b')))

H4 establishes (not (break-out-of-timep ($R^{world,clock}$(v)))).
Since H4 also establishes that clock > 0, by the definition of E, (E t ((x . v)) world clock) = t. Thus, the hypotheses of H10 are relieved, and we establish its conclusion.

Since targ and nilarg are variable-free, any binding may be used for b', so again we choose b.

By expansion of the definition of I, the second conjunct in the conclusion of H10 gives us:

 (or (and (I targ v b) (I $t ($R^{world,clock}$(v)) b)) (and (I nilarg v b) (I $nil (R^{world,clock}(v)) b)))

which expands further to

(or (and (I targ v b) (equal t (R^{world,clock}(v))))
(and (I nilarg v b) (equal nil (R^{world,clock}(v)))))

Using H9', we can rule out the second disjunct. This establishes
(and (I targ v b) (equal t (R^{world,clock}(v)) b)),
the second conjunct of which is our goal. QED.

B.3 The Proof of VALID-FS-CLOCK

Lemma VALID-FS-CLOCK

For any list of function signatures fs, world world, and
non-negative integer clock,

(valid-fs fs world clock)
=>
(valid-fs fs world clock-1)

Proof of Lemma VALID-FS-CLOCK

By induction on clock.

The good-signaturep predicate required by valid-fs for each
function foo is:

for any Lisp values arg$_1$ .. arg$_n$,
(and
H1 (not (break-out-of-timep (foo^{world,clock}(arg$_1$ .. arg$_n$))))
H2 (not (null
(E guard-form ((a$_1$. arg$_1$) .. (a$_n$. arg$_n$)) world clock))))
=>
(and
C1 (not (break-guard-violationp (foo^{world,clock}(arg$_1$ .. arg$_n$))))
C2 for some k in [1..m],
for some binding b of type variables to Lisp values
covering td$_{k,1}$ .. td$_{k,n}$ and td$_k$,
(I (td$_{k,1}$ .. td$_{k,n}$ td$_k$)
(arg$_1$ .. arg$_n$ (foo^{world,clock}(arg$_1$ .. arg$_n$)))
b)))

Base case: clock = 1
(valid-fs fs world 0) = t trivially, since for every function foo, we
have the hypothesis

(not (break-out-of-timep (foo^{world,clock}(arg$_1$ .. arg$_n$))))

which is always false when clock = 0.

Inductive case: clock = i
Case 1. (break-out-of-timep (foo$^{world,c-1}$(arg$_1$ .. arg$_n$)))

Since this falsifies H1, the conclusion is trivially true.

Case 2. (not (break-out-of-timep (foo$^{world,c-1}$(arg$_1$ .. arg$_n$))))

Simple examination of E (See Section 5.4.) shows that, given two calls to E which are identical except for the clock parameter, if neither call runs out of time, both calls will return the same value. Since our case assumption establishes that neither call returns (break-out-of-time), we can equate the results of all the calls to E with clock = i to their counterparts with clock = i-1, and thereby establish the goal.

B.4 The Central Checker Algorithm -- TC-INFER

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

Lemma TC-INFER-OK

For any Lisp form form, function signature list fs, Lisp world world
  including all functions hereditarily in the call tree of form,
for any non-negative integer clock, type variable bindings b,
 Lisp values arg_1 .. arg_m,
 Lisp variables a_1 .. a_m,
 binding environment env of the form ((a_1 . arg_1) .. (a_m . arg_m))
  where a_1 .. a_m include all the free variables in form,
 ABS-ALIST of the form ((a_1 . tda_1) .. (a_m . tda_m)),
 CONC-ALIST of the form ((a_1 . tdc_1) .. (a_m . tdc_m)),
and denoting
  (tc-infer form abs-alist conc-alist fs)
 by
  ((($mintd_{1,1}$ .. $mintd_{1,m}$ $mintd_1$)
    ((a_1 . $tdconc_{1,1}$) .. (a_m . $tdconc_{1,m}$))
    ($maxtd_{1,1}$ .. $maxtd_{1,m}$ $maxtd_1$))
   ..
   (($mintd_{n,1}$ .. $mintd_{n,m}$ $mintd_n$)
    ((a_1 . $tdconc_{n,1}$) .. (a_m . $tdconc_{n,m}$))
    ($maxtd_{n,1}$ .. $maxtd_{n,m}$ $maxtd_n$)))

H1 (and (valid-fs fs world clock)
H2   (I (tda_1 .. tda_m) (arg_1 .. arg_m) b)
H3   (I (tdc_1 .. tdc_m) (arg_1 .. arg_m) b)
H4   (not (equal (tc-infer form abs-alist conc-alist fs)
                    *guard-violation))
H5   (tc-all-called-functions-complete form fs)
H6   (not (break-out-of-timep (E form env world clock))))
   =>
   (and
C1   (not (break-guard-violationp (E form env world clock)))
C2   for some i,
      for some binding b' covering the descriptors below,
       (and (I ($mintd_{i,1}$ .. $mintd_{i,m}$ $mintd_i$)
               (arg_1 .. arg_m (E form env world clock))
               b')
            (I ($maxtd_{i,1}$ .. $maxtd_{i,m}$ $maxtd_i$)
               (arg_1 .. arg_m (E form env world clock))
               b')
            (I ($tdconc_{i,1}$ .. $tdconc_{i,m}$) (arg_1 .. arg_m) b')
            (extends-binding b' b)))

Note:
H1 establishes that the signatures in the system state fs are valid.
H2 establishes that the abs-arglist is valid.

```
H3 establishes that the conc-arglist is valid.
H4 establishes that no guard violations are detected in the
   course of analyzing form.
H5 establishes that the guards of all functions in the call tree
   of form are complete.
H6 establishes that the evaluation of form terminates without
   exhausting the clock.
```

The functions EXTENDS-BINDING, MERGEABLE-BINDINGS, and MERGE-BINDINGS are defined in Appendix B.3. Also in that appendix section are the statements and proofs of Lemmas EXTENDS-BINDING-MONOTONIC and MERGE-BINDINGS-EXTENDS-BINDINGS. These functions and lemmas are utilized in the following proof.

Proof of TC-INFER-OK

Note that the second conjunct of the conclusion prescribes that the binding b′ is to cover exactly the collection of variables in the descriptors under the quantifier. Any binding which happens to contain excess elements can be trimmed as necessary to suit the exact coverage requirement. This is trivially obvious from the definition of I.

We will proceed by induction on the structure of expressions.

Let us denote (E form env world clock) with the abbreviation form-val.

Case 1: form is an variable in the environment

Denote this variable a_i. By definition,
(E form env world clock) = arg_i.

In this case, by definition, tc-infer returns a single result tuple:

(tc-infer var abs-alist conc-alist fs)
=
((((tda_1 .. tda_n) -> tda_i) conc-alist ((tdc_1 .. tdc_n) -> tdc_i)))

The descriptors returned for the form a_i are thus the ones associated with a_i in the two alists. We can use b as the binding we need in the conclusion. Thus, our goal is:

```
H1 (and (valid-fs fs world clock)
H2      (I (tda1 .. tdam) (arg1 .. argm) b)
H3      (I (tdc1 .. tdcm) (arg1 .. argm) b)
H4      (not (equal (tc-infer form abs-alist conc-alist fs)
                    *guard-violation))
H5      (tc-all-called-functions-complete form fs)
H6      (not (break-out-of-timep form-val)) )
        =>
   (and
C1   (not (break-guard-violationp form-val))
C2   (I (tda1 .. tdam tdai) (arg1 .. argm argi) b)
C3   (I (tdc1 .. tdcm tdci) (arg1 .. argm argi) b)
C4   (I (tdc1 .. tdcm) (arg1 .. argm) b)
C5   (extends-binding b b))
```

No guard violation can occur on evaluation of a variable, so C1 holds. The expansion of H2 is identical to the first m conjuncts of the expansion of C2, and the expansion of H3 is identical to the first m conjuncts of the expansion of C3. The ith conjunct of the expansion of H2 is identical to the last conjunct of the expansion of C2, and the ith conjunct of the expansion of H3 is identical to the last conjunct of the expansion of C3. C4 is identical to H3. C5 is trivially true.

Case 2: form is a self-evaluating literal (T, NIL, an integer, rational, string, or character), or is a quoted form.

In this case, by definition, tc-infer returns a list containing a single tuple:

(tc-infer form abs-alist conc-alist fs)
 =
((((tda$_1$ .. tda$_n$) -> (descriptor-from-quote form))
  conc-alist
  ((tdc$_1$ .. tdc$_n$) -> (descriptor-from-quote form))))

We can use b as the binding we need in the conclusion. Thus, our goal is

H1 (and (valid-fs fs world clock)
H2   (I (tda$_1$ .. tda$_m$) (arg$_1$ .. arg$_m$) b)
H3   (I (tdc$_1$ .. tdc$_m$) (arg$_1$ .. arg$_m$) b)
H4   (not (equal (tc-infer form abs-alist conc-alist fs)
                    *guard-violation))
H5   (tc-all-called-functions-complete form fs)
H6   (not (break-out-of-timep form-val)))
        =>
   (and
C1   (not (break-guard-violationp form-val))
C2   (I (tda$_1$ .. tda$_m$ (descriptor-from-quote form))
        (arg$_1$ .. arg$_m$ form-val)
        b)
C3   (I (tdc$_1$ .. tdc$_m$ (descriptor-from-quote form))
        (arg$_1$ .. arg$_m$ form-val)
        b)
C4   (I (tdc$_1$ .. tdc$_m$) (arg$_1$ .. arg$_m$) b)
C5   (extends-binding b b))

No guard violation can occur on evaluation of a quoted form, so C1 holds.

We use the following lemma, instantiating with form = form and otherwise with namesakes.

Lemma DESCRIPTOR-FROM-QUOTE-OK:

For any Lisp form, binding environment env, Lisp world, type variable binding b, and non-negative integer clock,

(and (is-quoted-form form)
     (clock $\neq$ 0))
=>
(I (descriptor-from-quote form)
   (E form env world clock)
   b)

The proof of this lemma is presented in Appendix B.1.

The expansion of H2 is identical to the first m conjuncts of the expansion of C2, and the expansion of H3 to the first m conjuncts of the expansion of C3. Lemma DESCRIPTOR-FROM-QUOTE-OK establishes the last conjunct of both C2 and C3. C4 is identical to H3. C5 is trivially true.

Case 3: form is an IF expression, say (IF test-form then-form else-form)

We will adopt the notation that

(tc-infer (IF test-form then-form else-form)
          ((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))
          ((a$_1$. tdc$_1$) .. (a$_m$. tdc$_m$))

fs)
= ((((mintd$_{1,1}$ .. mintd$_{1,m}$) -> mintd$_1$)
((a$_1$. tdconc$_{1,1}$) .. (a$_m$. tdconc$_{1,m}$))
((maxtd$_{1,1}$ .. maxtd$_{1,m}$) -> maxtd$_1$))
..
(((mintd$_{n,1}$ .. mintd$_{n,m}$) -> mintd$_n$)
((a$_1$. tdconc$_{n,1}$) .. (a$_m$. tdconc$_{n,m}$))
((maxtd$_{n,1}$ .. maxtd$_{n,m}$) -> maxtd$_n$)))

(tc-infer test-form
((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))
((a$_1$. tdc$_1$) .. (a$_m$. tdc$_m$))
fs)
= ((((test-mintd$_{1,1}$ .. test-mintd$_{1,m}$) -> test-mintd$_1$)
((a$_1$. test-tdconc$_{1,1}$) .. (a$_m$. test-tdconc$_{1,m}$))
((test-maxtd$_{1,1}$ .. test-maxtd$_{1,m}$) -> test-maxtd$_1$))
..
(((test-mintd$_{l,1}$ .. test-mintd$_{l,m}$) -> test-mintd$_l$)
((a$_1$. test-tdconc$_{l,1}$) .. (a$_m$. test-tdconc$_{l,m}$))
((test-maxtd$_{l,1}$ .. test-maxtd$_{l,m}$) -> test-maxtd$_l$)))

(tc-infer then-form
((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))
((a$_1$. test-maxtd$_{i,1}$) .. (a$_m$. test-maxtd$_{i,m}$))
fs)
= ((((then-mintd$_{i,1,1}$ .. then-mintd$_{i,1,m}$) -> then-mintd$_{i,1}$)
((a$_1$. then-tdconc$_{i,1,1}$) .. (a$_m$. then-tdconc$_{i,1,m}$))
((then-maxtd$_{i,1,1}$ .. then-maxtd$_{i,1,m}$) -> then-maxtd$_{i,1}$))
..
(((then-mintd$_{i,o_i,1}$ .. then-mintd$_{i,o_i,m}$) -> then-mintd$_{i,o_i}$)
((a$_1$. then-tdconc$_{i,o_i,1}$) .. (a$_m$. then-tdconc$_{i,o_i,m}$))
((then-maxtd$_{i,o_i,1}$ .. then-maxtd$_{i,o_i,m}$) -> then-maxtd$_{i,o_i}$)))

(tc-infer else-form
((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))
((a$_1$. test-maxtd$_{i,1}$) .. (a$_m$. test-maxtd$_{i,m}$))
fs)
= ((((else-mintd$_{i,1,1}$ .. else-mintd$_{i,1,m}$) -> else-mintd$_{i,1}$)
((a$_1$. else-tdconc$_{i,1,1}$) .. (a$_m$. else-tdconc$_{i,1,m}$))
((else-maxtd$_{i,1,1}$ .. else-maxtd$_{i,1,m}$) -> else-maxtd$_{i,1}$))
..
(((else-mintd$_{i,p_i,1}$ .. else-mintd$_{i,p_i,m}$) -> else-mintd$_{i,p_i}$)
((a$_1$. else-tdconc$_{i,p_i,1}$) .. (a$_m$. else-tdconc$_{i,p_i,m}$))
((else-maxtd$_{i,p_i,1}$ .. else-maxtd$_{i,p_i,m}$) -> else-maxtd$_{i,p_i}$)))

We note also that the result of
(tc-infer (IF test-form then-form else-form)
((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))
((a$_1$. tdc$_1$) .. (a$_m$. tdc$_m$))
fs)
is by definition a subsequence of the append of the results
over all i for which we call

(tc-infer then-form
((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))
((a$_1$. test-maxtd$_{i,1}$) .. (a$_m$. test-maxtd$_{i,m}$))
fs)
and
(tc-infer else-form
((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))

((a$_1$. test-maxtd$_{i,1}$) .. (a$_m$. test-maxtd$_{i,m}$))
fs)

(We shall see later that we do some filtering of cases such that we do not necessarily call tc-infer on both the then-form and the else-form for any given i.)

We can make the inductive assumption of our lemma as it pertains to recursive calls on subforms of the IF expression. Let us assume, then, the following hypothesis, for the test form.

(and (valid-fs fs world clock)
    (I (tda$_1$ .. tda$_m$) (arg$_1$ .. arg$_m$) b)
    (I (tdc$_1$ .. tdc$_m$) (arg$_1$ .. arg$_m$) b)
    (not (equal (tc-infer test-form abs-alist conc-alist fs)
            *guard-violation))
    (tc-all-called-functions-complete test-form fs)
    (not (break-out-of-timep (E test-form env world clock))))
=>
(and
 (not (break-guard-violationp (E test-form env world clock)))
 for some i in 1..l
  for some binding b′ covering the following descriptors,
   (and (I (test-mintd$_{i,1}$ .. test-mintd$_{i,m}$ test-mintd$_i$)
        (arg$_1$ .. arg$_m$ (E test-form env world clock))
        b)
      (I (test-maxtd$_{i,1}$ .. test-maxtd$_{i,m}$ test-maxtd$_i$)
        (arg$_1$ .. arg$_m$ (E test-form env world clock))
        b)
      (I (test-tdconc$_{i,1}$ .. test-tdconc$_{i,m}$)
        (arg$_1$ .. arg$_m$)
        b)
      (extends-binding b′ b)))

We need to relieve the antecedents to this hypothesis. (tc-all-called-functions-complete form fs) trivially implies (tc-all-called-functions-complete test-form fs). H6 implies (not (break-out-of-timep (E test-form env world clock))), since (E test-form env world clock) is part of the evaluation of (E (if test-form then-form else-form) env world clock). H3 establishes the *guard-violation hypothesis, since if the latter were not true, by definition of tc-infer, the former would not be true, either. Since all the other antecedents to this hypothesis are all hypotheses in our theorem, we have established the conclusions:

H7 (not (break-guard-violationp (E test-form env world clock)))
H8 for some i in 1..l,
  for some binding b$_i$ covering the following descriptors,
   (and (I (test-mintd$_{i,1}$ .. test-mintd$_{i,m}$ test-mintd$_i$)
        (arg$_1$ .. arg$_m$ (E test-form env world clock))
        b$_i$)
      (I (test-maxtd$_{i,1}$ .. test-maxtd$_{i,m}$ test-maxtd$_i$)
        (arg$_1$ .. arg$_m$ (E test-form env world clock))
        b$_i$)
      (I (test-tdconc$_{i,1}$ .. test-tdconc$_{i,m}$)
        (arg$_1$ .. arg$_m$)
        b$_i$)
      (extends-binding b$_i$ b))

By the definition of E, if the test expression of an IF evaluates without a break to any non-NIL value, the value of the IF expression is the value returned by the evaluation of then-form. If it evaluates to NIL, the value of the IF expression is the value returned by the evaluation of else-form. Thus, consider the pool

of results for all i in 1..l,

```
((test-mintd_{i,1} .. test-mintd_{i,m} test-mintd_i)
 (test-tdconc_{i,1} .. test-tdconc_{i,m})
 (test-maxtd_{i,1} .. test-maxtd_{i,m} test-maxtd_i))
```

in H8. For any i where test-maxtd$_i$ cannot possibly represent a non-NIL value, we do not need to consider the possibility that the corresponding context,

```
(test-mintd_{i,1} .. test-mintd_{i,m})
(test-tdconc_{i,1} .. test-tdconc_{i,m})
(test-maxtd_{i,1} .. test-maxtd_{i,m})
```

is relevant in computing a descriptor for then-form. Similarly, if test-maxtd$_i$ cannot possibly represent a NIL value, we do not need to consider this context in connection with else-form.

By definition, TC-INFER performs a screening operation to determine which of the test-form tuples need to be considered as a context for analyzing then-form and else-form. Specifically, TC-INFER unifies each test-maxtd$_i$ with a screening descriptor

```
(*OR $CHARACTER $INTEGER $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
     $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
```

removing from consideration in then-form any tuple for which this unification returns *EMPTY. Similarly, for the else arm $NIL is used as a screening descriptor.

So now let us consider two cases, depending on the outcome of the evaluation of test-form.

Case 3.1 (E test-form env world clock) = some non-NIL value

Thus, the value returned by the IF is the value returned by then-form.

For any binding b (since the screening descriptor is variable-free),

```
(I (*or $character $integer $non-integer-rational $non-t-nil-symbol
        $string $t (*cons *universal *universal))
   (E test-form env world clock)
   b)
```

Consider each tuple from the test-form.
The formal justification for the screening operation uses the lemma:

Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK
For any descriptors tda and tdb, Lisp value v and fully instantiating binding of type variables to Lisp values b,

```
(and (I tda v b)
     (I tdb v b))
=>
(I (dunify-descriptors-interface tda tdb) v b)
```

instantiated with v = (E test-form env world clock),
td1 = test-maxtd$_i$, and

```
td2 = (*or $character $integer $non-integer-rational $non-t-nil-symbol
           $string $t (*cons *universal *universal)).
```

Consider the case (dunify-descriptors-interface td1 td2) = *empty. (I *empty (E test-form env world clock) b) = nil. Therefore, there is no binding b$_i$ such that

```
(I test-maxtd_i (E test-form env world clock) b_i)
```

is true. This being the case, we need not consider this tuple in analyzing then-form, since there is no way this tuple can signify a context which can result in a non-NIL value for the test form. Removing it from the pool will not affect the truth of H8.

By definition, TC-INFER uses each of the remaining (test-maxtd$_{i,1}$ .. test-maxtd$_{i,m}$) to form a CONC-ALIST for a recursive call:

(tc-infer then-form
((a$_1$. tda$_1$) .. (a$_m$. tda$_m$))
((a$_1$. test-maxtd$_{i,1}$) .. (a$_m$. test-maxtd$_{i,m}$))
fs)

So use TC-INFER-OK as an inductive assumption applied to then-form. We know for any i and b$_i$ for which H8 holds, we can use this inductive assumption, since H1 is equal to H1' (by Hi' we mean here the i-th hypothesis of the inductive assumption), H2 equals H2', H8 guarantees H3', H4 guarantees H4', since if TC-INFER returned *guard-violation for the then-form, it would also return *guard-violation for the IF, H5 guarantees H5', since then-form is a component of the IF, and H6 guarantees H6', since under our case assumption, the evaluation of the then-form is part of the evaluation of the IF, and since there was sufficient clock for the whole thing, there must have been sufficient clock for any subsidiary computation. Thus, we establish the conclusion of the inductive hypothesis, where o$_i$ is the number of tuples generated by recursive calls to TC-INFER for the i-th type in the test-form result,

H9 (and
(not (break-guard-violationp (E then-form env world clock)))
for all i such that the ith tuple survived the screen,
for some j in [1..o$_i$],
for some b$_{i,j}$ covering the following descriptors,
(and (I (then-mintd$_{i,j,1}$ .. then-mintd$_{i,j,m}$ then-mintd$_{i,j}$)
(arg$_1$ .. arg$_m$ (E then-form env world clock))
b$_{i,j}$)
(I (then-maxtd$_{i,j,1}$ .. then-maxtd$_{i,j,m}$ then-maxtd$_{i,j}$)
(arg$_1$ .. arg$_m$ (E then-form env world clock))
b$_{i,j}$)
(I (then-tdconc$_{i,j,1}$ .. then-tdconc$_{i,j,m}$)
(arg$_1$ .. arg$_m$)
b$_{i,j}$)
(extends-binding b$_{i,j}$ b$_i$)))

Since the result returned by TC-INFER for the IF includes the appended tuples of all the results from the recursive calls on then-form, H9 virtually establishes our goal. Since the value of the IF is the value returned for then-form, the first conjunct of H9 establishes C1. Using the same existential quantifiers, the first three conjuncts of the second conjunct of H9 establish the first three conjuncts of C2. H8 established that (extends-binding b$_i$ b), and thus the final conjunct of H9 and the transitivity of extends-binding, given by Lemma EXTENDS-BINDING-TRANSITIVE, establish the final conjunct of C2. Thus, we are done with this case.

Case 3.2 (E test-form env world clock) = nil

The argument is exactly analogous to Case 3.1, using $NIL for the screen and using a recursive call on else-form to compute the tuples which guarantee the result.

Case 4: form is a function call, say (f $form_1$ .. $form_o$)

Let us adopt the following notation:

```
(tc-infer (f form_1 .. form_o)
          ((a_1 . tda_1) .. (a_m . tda_m))
          ((a_1 . tdc_1) .. (a_m . tdc_m))
          fs)
=
((((mintd_1,1 .. mintd_1,m) -> mintd_1)
  ((a_1 . tdconc_1,1) .. (a_m . tdconc_1,m))
  ((maxtd_1,1 .. maxtd_1,m) -> maxtd_1))
 ..
 (((mintd_n,1 .. mintd_n,m) -> mintd_n)
  ((a_1 . tdconc_n,1) .. (a_m . tdconc_n,m))
  ((maxtd_n,1 .. maxtd_n,m) -> maxtd_n)))
```

and for all i in [1..o], (where o is the number of function arguments)

```
(tc-infer form_i
          ((a_1 . tda_1) .. (a_m . tda_m))
          ((a_1 . tdc_1) .. (a_m . tdc_m))
          fs)
=
((((mintd_i,1,1 .. mintd_i,1,m) -> mintd_i,1)
  ((a_1 . tdconc_i,1,1) .. (a_m . tdconc_i,1,m))
  ((maxtd_i,1,1 .. maxtd_i,1,m) -> maxtd_i,1))
 ..
 (((mintd_i,l_i,1 .. mintd_i,l_i,m) -> mintd_i,l_i)
  ((a_1 . tdconc_i,l_i,1) .. (a_m . tdconc_i,l_i,m))
  ((maxtd_i,l_i,1 .. maxtd_i,l_i,m) -> maxtd_i,l_i)))
```

I.e., there are l_i different 3-tuples generated for the ith argument.

We assume our lemma inductively as it applies to each of the function arguments. The tc-all-called-functions-complete hypotheses are trivially implied by H5, the break-out-of-timep hypotheses are guaranteed by H6, and the *guard-violation hypothesis is guaranteed by H3, since if the former were not true, by definition of tc-infer, the latter would not hold, either. Since the antecedents for each of the other inductive hypotheses are identical with the hypotheses of our lemma, we will simply use the conclusion as our inductive hypothesis for each actual argument to f and list it as H7.

```
H7  for each i in [1..o],
    (and
     (not (break-guard-violationp (E form_i env world clock)))
     for some j in [1..l_i] (the number of tuples for this actual),
      there exists a binding b_i,j covering the descriptors below
       (and (I (mintd_i,j,1 .. mintd_i,j,m mintd_i,j)
               (arg_1 .. arg_m (E form_i env world clock))
               b_i,j)
            (I (maxtd_i,j,1 .. maxtd_i,j,m maxtd_i,j)
               (arg_1 .. arg_m (E form_i env world clock))
               b_i,j)
            (I (tdconc_i,j,1 .. tdconc_i,j,m)
               (arg_1 .. arg_m)
               b_i,j)
            (extends-binding b_i,j b)) )
```

Furthermore by the (valid-fs fs world clock) hypothesis H1 applied to f, we can assume the correctness of the signature from fs for the function f. Let us denote the formal parameters of f to be v_1 .. v_o, the guard form guard-form, and the body body-form. Let us also denote that the signature for f contains the guard
(gtd_1 .. gtd_o)

and the segments:

(($std_{1,1}$ .. $std_{1,o}$) -> std_1) .. (($std_{p,1}$ .. $std_{p,o}$) -> std_p)

From (valid-fs fs world clock), we know the following about this signature:

```
(and
  (tc-all-called-functions-complete guard-form fs)
  (tc-all-called-functions-complete body fs)
  (not (break-out-of-timep
         (f^{world,clock}
          ((E form_1 env world clock) .. (E form_o env world clock)))))
  (not (null (E guard-form
                ((v_1 . (E form_1 env world clock))
                  ..
                 (v_n . (E form_o env world clock)))
                world clock))) )
 =>
 (and
C1   (not (break-guard-violationp
          (f^{world,clock}
            ((E form_1 env world clock) .. (E form_o env world clock)))))
C2   for some k in [1..p]
       for some binding b of type variables to Lisp values
           covering std_{k,1} .. std_{k,o} and std_k
         (I (std_{k,1} .. std_{k,o} std_k)
            ((E form_1 env world clock) .. (E form_o env world clock)
             (f^{world,clock}
               ((E form_1 env world clock) .. (E form_o env world clock))))
            b) )
```

Notice that new variables are introduced to our problem state with the use of this signature. By definition, when TC-INFER uses a signature from fs, it replaces all the variables in the stored representation of the signature with "fresh" or previously unused variables. Thus, for any two triples returned by TC-INFER, the variables they share are limited to those in the argument list of the call.

We would like to use the conclusion of the above-stated implication characterizing the signature of f. First, we must relieve its antecedents. H5 guarantees its tc-all-called-functions-complete hypotheses. H6 guarantees its break-out-of-timep hypothesis, since the operation
($f^{world,clock}$((E $form_1$ env world clock) .. (E $form_o$ env world clock)))
is just a part of the evaluation of form, and H6 ensures adequate clock for the entire evaluation. The remaining hypothesis reflects the guard verification step, which is handled as follows.

For each actual parameter (E $form_i$ env world clock), TC-INFER gathers from the maximal segments the result types known to be possible (the $maxtd_{i,j}$'s from H7) and disjoins them with *OR as a conservatively large representation of the type of (E $form_i$ env world clock). I.e., consider in H7 the result descriptors $maxtd_{i,j}$ from the expansion for each j of:

for some j in [1..l_i]
 for some binding $b_{i,j}$,
  (I (maxtd$_{i,j,1}$ .. maxtd$_{i,j,m}$ maxtd$_{i,j}$)
     (arg$_1$ .. arg$_m$ (E form$_i$ env world clock))
     b$_{i,j}$)

Expand the definition of I and take the last conjunct of the result,

for some j in [1..l_i]
 for some binding $b_{i,j}$,
  (I maxtd$_{i,j}$ (E form$_i$ env world clock) b$_{i,j}$)

We can generalize the descriptor in each case by disjoining it with all the other maxtd$_{i,j}$'s. This is clearly valid, since the original call to I is in the immediate expansion of I in our transformed result:

H8 for some j in [1..l_i]
 for some binding $b_{i,j}$,
  (I ((*or maxtd$_{i,1}$ .. maxtd$_{i,l_i}$))
     (E form$_i$ env world clock)
     b$_{i,j}$)

Now we wish to consider together the descriptors formed in this manner for each i. Recall that the form above holds for all i in 1..o. We wish to claim:

H9 for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
 (and
  (I ((*or maxtd$_{1,1}$ .. maxtd$_{1,l_1}$)
      ..
      (*or maxtd$_{o,1}$ .. maxtd$_{o,l_o}$))
     ((E form$_1$ env world clock)
      ..
      (E form$_o$ env world clock))
     (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$))
  (extends-binding (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b))

MERGE-BINDINGS* simply merges all the bindings in its argument list, using MERGE-BINDINGS. See Appendix B.3.

Modulo the binding parameter, the expansion of I in this form is the conjunction of the forms just previously constructed. To establish H9, we need show for all i in 1..o,

for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
  (I ((*or maxtd$_{i,1}$ .. maxtd$_{i,l_i}$))
     (E form$_i$ env world clock)
     (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$))

Consider the bindings so merged to be some collection of bindings which sufficed for H8. We would like to use Lemma MERGE-BINDINGS-EXTENDS-BINDINGS to show that
(merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$)
extends each of its constituents. To do so, we must show that each of the constituents is mergeable with the others. Since each is an extension of b, all the bindings of variables from b are consistent among the constituents. The binding elements added by extension to each were the result of distinct calls of TC-INFER on the various form$_i$'s, and since TC-INFER introduces only variables which remain unique throughout the life of the analysis, and since the

bindings from the inductive assumptions contain no extraneous elements, no two invocations of TC-INFER will result in new bindings which conflict with one another in a merge. Thus, all the constituent bindings are mergeable with one another. And therefore by Lemma EXTENDS-BINDING-MONOTONIC, the claim above follows directly from H8. Furthermore, since H7 establishes that each b$_{i,j}$ used in the merge extends b, and since we just determined that the merge extends each constituent b$_{i,j}$, by Lemma EXTENDS-BINDING-TRANSITIVE we establish the second conjunct of H9, thus establishing the whole.

Having formed the *dlist descriptor in H9, TC-INFER performs the following computation:

```
H10  (contained-in-interface (*dlist (*or maxtd_{1,1} .. maxtd_{1,l_1})
                                     ..
                                     (*or maxtd_{o,1} .. maxtd_{o,l_o}))
                             (*dlist gtd_1 .. gtd_o))
```

Had this test failed, TC-INFER would have returned *guard-violation as its result, but H4 guarantees this did not happen, so we know the containment test succeeded. Thus, H10 is established.

Now we instantiate Lemma GUARD-COMPLETE (See Section 3.6.1) with fs = fs, clock = clock, world = world, n = o, each arg$_i$ = (E form$_i$ env world clock), argtd$_1$ .. argtd$_n$ = (*or maxtd$_{1,1}$ .. maxtd$_{1,l_1}$) .. (*or maxtd$_{o,1}$ .. maxtd$_{o,l_o}$), b = (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$), and rtd$_1$ .. rtd$_n$ = (gtd$_1$ .. gtd$_o$). H1 equals the H1 in GUARD-COMPLETE, H9 establishes its H2, H10 equals its H3, and H6 guarantees its H4, since the guard evaluation is a part of the larger evaluation for which there is sufficient clock time. So we can use the conclusion of the instantiated lemma to give us the final antecedent of the signature fact from valid-fs for the signature of f. Henceforth, we will have at our disposal its conclusion, which we tag H11.

Thus,
- **m is the number of identifiers in the context, indexed by g**
- **o is the number of parameters in the call to f, indexed by i**
- **l$_i$ is the number of tuples generated for each actual i, indexed by j**
- **p is the number of segments in the signature for f, indexed by k**
- **n is the number of tuples for (f form$_1$ .. form$_o$), indexed by s**
- **q is the number of elements in the cross product (see below), indexed by h**

Our goal is:

```
   (and
H1  (valid-fs fs world clock)
H2  (I (tda_1 .. tda_m) (arg_1 .. arg_m) b)
H3  (I (tdc_1 .. tdc_m) (arg_1 .. arg_m) b)
H4  (not (equal (tc-infer form abs-alist conc-alist fs)
          *guard-violation))
H5  (tc-all-called-functions-complete form fs)
H6  (not (break-out-of-timep (E form env world clock)))
H7  for each i in 1..o,
    (and
     (not (break-guard-violationp (E form_i env world clock)))
     for some j in [1..l_i] (the number of tuples for this actual),
      there exists a binding b_{i,j} covering the descriptors below
       (and (I (mintd_{i,j,1} .. mintd_{i,j,m} mintd_{i,j})
```

```
              (arg_1 .. arg_m (E form_i env world clock))
              b_{i,j})
           (I (maxtd_{i,j,1} .. maxtd_{i,j,m} maxtd_{i,j})
              (arg_1 .. arg_m (E form_i env world clock))
              b_{i,j})
           (I (tdconc_{i,j,1} .. tdconc_{i,j,m})
              (arg_1 .. arg_m)
              b_{i,j})
           (extends-binding b_{i,j} b)) )
H8  for each i in 1..o,
     for some j in [1..l_i],
      for some binding b_{i,j},
        (I ((*or maxtd_{i,1} .. maxtd_{i,l_i}))
           (E form_i env world clock)
           b_{i,j})

H11 (and
     (not (break-guard-violationp
           (f^{world,clock}
             ((E form_1 env world clock) .. (E form_o env world clock)))))
     for some k in [1..p],
       for some binding b' covering std_{k,1} .. std_{k,o} and std_k,
         (I (std_{k,1} .. std_{k,o} std_k)
            ((E form_1 env world clock) .. (E form_o env world clock)
             (f^{world,clock}
               ((E form_1 env world clock) .. (E form_o env world clock))))
            b') ) )
   =>
   (and
C1   (not (break-guard-violationp (E form env world clock)))
     for some s in [1..n],
      for some binding b'' covering the descriptors below,
C2     (and (I (mintd_{s,1} .. mintd_{s,m} mintd_s)
              (arg_1 .. arg_m form-val)
              b'')
C3          (I (maxtd_{s,1} .. maxtd_{s,m} maxtd_s)
              (arg_1 .. arg_m form-val)
              b'')
C4          (I (tdconc_{s,1} .. tdconc_{s,m}) (arg_1 .. arg_m) b'')
C5          (extends-binding b'' b)) )
```

(H9 and H10 are no longer useful and have been discarded.)

The first conjunct of H7 for each $form_i$ and the first conjunct of H11 establishes our goal C1, which says that the entire form will evaluate without guard violation.

Now consider H7 again. Dropping the break-guard-violation-p result and making all the j quantifiers distinct via subscripting as we did in deriving H9, and fixing each b_{i,j_i} to be some binding which serves for H7, we can restate this as

```
for some j_1 in 1..l_1, .., for some j_o in 1..l_o,
 for each i in 1..o
   (and (I (mintd_{i,j_i,1} .. mintd_{i,j_i,m} mintd_{i,j_i})
           (arg_1 .. arg_m (E form_i env world clock))
           b_{i,j_i})
        (I (maxtd_{i,j_i,1} .. maxtd_{i,j_i,m} maxtd_{i,j_i})
           (arg_1 .. arg_m (E form_i env world clock))
```

```
            b_{i,j_i})
         (I (tdconc_{i,j_i,1} .. tdconc_{i,j_i,m})
            (arg_1 .. arg_m)
            b_{i,j_i})
         (extends-binding b_{i,j_i} b))
```

Now, merge all the b_{i,j_i}, just as we did above. The same argument as used above, regarding the mergeability of these bindings and their binding extension properties, establishes

```
H12 for some j_1 in 1..l_1, .., for some j_o in 1..l_o,
     for each i in 1..o
       (and (I (mintd_{i,j_i,1} .. mintd_{i,j_i,m} mintd_{i,j_i})
               (arg_1 .. arg_m (E form_i env world clock))
               (merge-bindings* b_{1,j_1} .. b_{o,j_o}))
            (I (maxtd_{i,j_i,1} .. maxtd_{i,j_i,m} maxtd_{i,j_i})
               (arg_1 .. arg_m (E form_i env world clock))
               (merge-bindings* b_{1,j_1} .. b_{o,j_o}))
            (I (tdconc_{i,j_i,1} .. tdconc_{i,j_i,m})
               (arg_1 .. arg_m)
               (merge-bindings* b_{1,j_1} .. b_{o,j_o}))
            (extends-binding (merge-bindings* b_{1,j_1} .. b_{o,j_o}) b))
```

By definition of TC-INFER, the next operation of the checker algorithm is to unify two lists of descriptors, each of which represents the collected types of the variables in the environment, the types of the parameters in the function call, and the type of the result. One list corresponds to what is known about the called function from its signature, the other to what is known in the environment of the function call.

The first list is of the form:

```
 (*universal_1 .. *universal_m std_{k,1} .. std_{k,o} std_k)
```

where each $*universal_g$ represents the type of the g-th identifier in the environment, each $std_{k,i}$ represents the type of the ith formal parameter to the function call, and std_k represents the result type. Each instance of the first list is trivially constructed from a signature segment for the called function.

The second list is of the form:

```
 (tda_1 .. tda_m mintd_{1,j} .. mintd_{o,j} *universal),
```

where each tda_g is some representation of the type of the gth variable in the environment, each $mintd_{i,j}$ represents the type of the ith actual parameter to the function call, and *universal represents the result type. This list must be constructed from what we have already determined about our actual parameters. Each such list must represent some possible combination of the segments for all the actual parameters:

```
 (tc-infer form_i
           ((a_1 . tda_1) .. (a_m . tda_m))
           ((a_1 . tdc_1) .. (a_m . tdc_m))
           fs)
 =
 ((((mintd_{i,1,1} .. mintd_{i,1,m}) -> mintd_{i,1})
```

((a$_1$. tdconc$_{i,1,1}$) .. (a$_m$. tdconc$_{i,1,m}$))
((maxtd$_{i,1,1}$ .. maxtd$_{i,1,m}$) -> maxtd$_{i,1}$))
..
(((mintd$_{i,l_i,1}$ .. mintd$_{i,l_i,m}$) -> mintd$_{i,l_i}$)
((a$_1$. tdconc$_{i,l_i,1}$) .. (a$_m$. tdconc$_{i,l_i,m}$))
((maxtd$_{i,l_i,1}$ .. maxtd$_{i,l_i,m}$) -> maxtd$_{i,l_i}$)))

This construction is essentially a cross product combining the minimal segments for each parameter into one collection of segments with multiple results, one for each actual parameter. Although when we combine these segments, the actual parameter descriptors shuffle neatly into a list, the context descriptors get smashed together via descriptor unification. Thus, each combination of two segments yields a list of segments, and each segment from the list must be similarly combined with each segment for the next parameter. Each element of the cross product describes a possible combination of the types of the variables in the context which leads to the corresponding list of actual parameter types.

The function performing the cross product operation is TC-MAKE-ARG-CROSS-PRODUCT, and its correctness lemma is:

Lemma TC-MAKE-ARG-CROSS-PRODUCT-OK

Where o is the number of parameters in the function call,
l_1 **..** l_o **are the lengths of the lists of segments for the actual parameters,**
m is the number of variables in the context,
and denoting

(TC-MAKE-ARG-CROSS-PRODUCT
((((mintd$_{1,1,1}$ .. mintd$_{1,1,m}$) -> mintd$_{1,1}$)
(tdconc$_{1,1,1}$ .. tdconc$_{1,1,m}$)
((maxtd$_{1,1,1}$ .. maxtd$_{1,1,m}$) -> maxtd$_{1,1}$))
..
(((mintd$_{1,l_1,1}$ .. mintd$_{1,l_1,m}$) -> mintd$_{1,l_1}$)
(tdconc$_{1,l_1,1}$ .. tdconc$_{1,l_1,m}$)
((maxtd$_{1,l_1,1}$ .. maxtd$_{1,l_1,m}$) -> maxtd$_{1,l_1}$)))
...
((((mintd$_{o,1,1}$ .. mintd$_{o,1,m}$) -> mintd$_{o,1}$)
(tdconc$_{o,1,1}$ .. tdconc$_{o,1,m}$)
((maxtd$_{o,1,1}$ .. maxtd$_{o,1,m}$) -> maxtd$_{o,1}$))
..
(((mintd$_{o,l_o,1}$ .. mintd$_{o,l_o,m}$) -> mintd$_{o,l_o}$)
(tdconc$_{o,l_o,1}$ .. tdconc$_{o,l_o,m}$)
((maxtd$_{o,l_o,1}$ .. maxtd$_{o,l_o,m}$) -> maxtd$_{o,l_o}$)))))
=
(((cptda$_{1,1}$ .. cptda$_{1,m}$)
(cptdr$_{1,1}$ .. cptdr$_{1,o}$)
(cptdconc$_{1,1}$ .. cptdconc$_{1,m}$))
..
((cptda$_{q,1}$ .. cptda$_{q,m}$)
(cptdr$_{q,1}$ .. cptdr$_{q,o}$)
(cptdconc$_{q,1}$ .. cptdconc$_{q,m}$)))

For any mintd, maxtd, and tdconc descriptors, type variable binding b, non-negative integers l_1 **..** l_n**, m, and o, Lisp values arg**$_1$ **.. arg**$_m$**, and Lisp values actual**$_1$ **.. actual**$_o$**,**

for all i in [1..o],
for some j in [1..l_i**],**

(I (mintd$_{i,j,1}$ .. mintd$_{i,j,m}$ mintd$_{i,j}$ tdconc$_{i,j,1}$ .. tdconc$_{i,j,m}$)
(arg$_1$ .. arg$_m$ actual$_i$ arg$_1$ .. arg$_m$)
b)
=>
for some h in [1..q],
(I (cptda$_{h,1}$ .. cptda$_{h,m}$ cptdr$_{h,1}$ .. cptdr$_{h,o}$
cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$ actual$_1$ .. actual$_o$ arg$_1$ .. arg$_m$)
b)

The proof of this lemma is given in Appendix B.2.

For any j$_1$..j$_o$ for which H12 holds, we instantiate this lemma with our triples, i.e., with namesakes throughout, with each actual$_i$ instantiated with (E form$_i$ env world clock), and with b = (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$).
The antecedent for TC-MAKE-ARG-CROSS-PRODUCT-OK is trivially implied by H12, so we establish its conclusion for the correct j$_1$..j$_o$:

for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
for some h in [1..q],
(I (cptda$_{h,1}$ .. cptda$_{h,m}$
cptdr$_{h,1}$ .. cptdr$_{h,o}$
cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
arg$_1$ .. arg$_m$)
(merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$))

This is equivalent to the following, with the transformation accomplished simply by separating the independent conjuncts of the list into two groups.

H13 for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
for some h in [1..q],
(and
(I (cptda$_{h,1}$ .. cptda$_{h,m}$ cptdr$_{h,1}$ .. cptdr$_{h,o}$)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock))
(merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$))
(I (cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$)
(merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$)))

The next step in the algorithm is to unify each of the elements from the cross product (the desired property for one of which is represented in the first conjunct of H13) with each of the segments from the signature of the called function (represented in H11). Again, to make these descriptor lists align appropriately, they will need to be padded here and there. In the cross product elements, we need only add a *universal descriptor to represent the result of the function call. Since this introduces no new constraint, what we know from H13 will imply our claim about the extended cross product elements, giving us the revised hypothesis:

H13' for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
for some h in [1..q],
(and
(I (cptda$_{h,1}$ .. cptda$_{h,m}$ cptdr$_{h,1}$ .. cptdr$_{h,o}$ *universal)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)

form-val)
(merge-bindings* b_{1,j_1} .. b_{o,j_o}))
(I (cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$)
(merge-bindings* b_{1,j_1} .. b_{o,j_o})))

For f's signature segments, represented in H11, what is missing in this matchup are descriptors for each of the objects in the calling context. These will each be represented also with *universal, giving us a modified hypothesis:

H11' for some k in [1..p]
for some binding b' of type variables to Lisp values
covering std$_{k,1}$ .. std$_{k,o}$ and std$_k$
(I (*universal$_1$ .. *universal$_m$ std$_{k,1}$ .. std$_{k,o}$ std$_k$)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
form-val)
b')

Note that any b' and (merge-bindings* b_{1,j_1} .. b_{o,j_o})
are mergeable, since the variables from the segments of f are all fresh and hence totally disjoint from those in the merged binding. Thus, by Lemma MERGE-BINDINGS-EXTENDS-BINDINGS,
(merge-binding (merge-bindings* b_{1,j_1} .. b_{o,j_o}) b')
extends both its arguments, and hence by Lemma EXTENDS-BINDING-MONOTONIC, we can take any pair of bindings which serve in H11' and H13' and use their merge wherever either appeared before. This gives us:

H14 for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
for some h in [1..q],
for some k in [1..p],
for some binding b' covering std$_{k,1}$ .. std$_{k,o}$ and std$_k$,
(and
(I (cptda$_{h,1}$ .. cptda$_{h,m}$ cptdr$_{h,1}$ .. cptdr$_{h,o}$ *universal)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
form-val)
(merge-bindings (merge-bindings* b_{1,j_1} .. b_{o,j_o}) b'))
(I (cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$)
(merge-bindings (merge-bindings* b_{1,j_1} .. b_{o,j_o}) b'))
(I (*universal$_1$ .. *universal$_m$ std$_{k,1}$ .. std$_{k,o}$ std$_k$)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
form-val)
(merge-bindings (merge-bindings* b_{1,j_1} .. b_{o,j_o}) b'))

Now we unify pairwise the descriptor lists from H11 and H13, i.e., each pair characterized by the first and third conjuncts of H14. We create a list of results, each element of which is a list of the same configuration as the arguments. If an *or is returned from the unification, each disjunct becomes an element of this list. If a simple list is returned, the length of the list is one. We are just replacing the interpretation over the *or by the existential quantifier over the interpretations of the disjuncts. (Note that this is just another instance where we observe the packaging protocal for DUNIFY-DESCRIPTORS-INTERFACE when the arguments are *dlists. See the discussion in Section 6.6.2.) So let us denote this result:

((mintd$_{1,1}$ .. mintd$_{1,m}$ mintd$_{1,actual_1}$ .. mintd$_{1,actual_o}$ mintd$_1$)
..
(mintd$_{t_{k,h},1}$ .. mintd$_{t_{k,h},m}$ mintd$_{t_{k,h},actual_1}$ .. mintd$_{t_{k,h},actual_o}$ mintd$_{t_{k,h}}$)),

where by mintd$_{1,actual_1}$ we refer to a descriptor
corresponding to the type of the first actual parameter, i.e, the
type of (E form$_1$ env world clock), etc., and by t$_{k,h}$
we refer to the number of results produced by the unifier for the
k-th form from H11' (third conjunct of H14) and the h-th form from
H13' (first conjunct of H14).

These mintd's are the descriptors which will finally be a part of the
result returned by TC-INFER. In particular, in each 3-tuple of the
result, the minimal segment will be extracted from one of these
unified results. The appended results of all the pairwise
unifications will form the entire collection of minimal segments.

So we instantiate Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK as follows,
for each h in 1..q and k in 1..p such that H14 is satisfied,

(and (I (*universal$_1$ .. *universal$_m$ std$_{k,1}$ .. std$_{k,o}$ std$_k$)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
form-val)
(merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b'))
(I (cptda$_{h,1}$ .. cptda$_{h,m}$ cptdr$_{h,1}$ .. cptdr$_{h,o}$ *universal)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
form-val)
(merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b')))
=>
for some s in 1..t$_{k,h}$,
(I (mintd$_{s,1}$ .. mintd$_{s,m}$ mintd$_{s,actual_1}$ .. mintd$_{s,actual_o}$ mintd$_s$)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
form-val)
(merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b'))

Since we know that some descriptor list from each of H11' and
H13' satisfies the predicate with the appropriate bindings,
then by this lemma some descriptor list from the collected
results of all the pairwise unifications satisfies the
conclusion. Thus,

for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
for some s indexing the list of collected results,
(I (mintd$_{s,1}$ .. mintd$_{s,m}$ mintd$_{s,actual_1}$ .. mintd$_{s,actual_o}$ mintd$_s$)
(arg$_1$ .. arg$_m$
(E form$_1$ env world clock) .. (E form$_o$ env world clock)
form-val)
(merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b'))

Since we no longer need the descriptors for the actual parameters, we
can drop them from this predicate, giving us our minimal segments
and their correctness specification:

for some j$_1$ in 1..l_1, .., for some j$_o$ in 1..l_o,
for some s indexing the list of collected results,
(I (mintd$_{s,1}$ .. mintd$_{s,m}$ mintd$_s$)
(arg$_1$ .. arg$_m$ form-val)
(merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b'))

and we have established C2 of our goal.

These collected results are re-paired with the respective conc-alists which accompanied each of the cross product descriptors. In cases where a particular unification produced multiple results, the conc-alist is paired with each of the results. Thus, the truth of

(I (cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)
   (arg$_1$ .. arg$_m$)
   (merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b'))

which was established in H14, is preserved in the conclusion, where s is the index rather than h. This establishes C4. The cptdconc's are the tdconc's of our ultimate result.

The maximal segments for each tuple are computed by combining via unification of the descriptors in each minimal segment/conc-alist pair. C4 represents what we know of the conc-alist paired with each minimal segment. Again, since the CONC-ALIST gives only descriptors for the types of the variables in the environment and says nothing about the type of the result of the function call (i.e., the conc-alist is n-ary and the minimal segment is has arity n+1), we need to pad the descriptor lists associated with the CONC-ALIST before unifying it with our minimal segments. We simply add a *universal to correspond to the result type. Clearly, the result value satisfies *universal. Thus, for each s, we perform the unification:

(dunify-descriptors-interface
   (*dlist tdconc$_{s,1}$ .. tdconc$_{s,m}$ *universal)
   (*dlist mintd$_{s,1}$ .. mintd$_{s,m}$ mintd$_s$))
 =
((*dlist maxtd$_{s,1,1}$ .. maxtd$_{s,1,m}$ maxtd$_{s,1}$)
 ..
 (*dlist maxtd$_{s,x,1}$ .. maxtd$_{s,x,m}$ maxtd$_{x,s}$))

I.e., for any given minimal segment, unifying against the CONC-ALIST may produce multiple results, and we make a list of them. Once again, we invoke DUNIFY-DESCRIPTORS-INTERFACE-OK, instantiated for each combination:

(and (I (tdconc$_{s,1}$ .. tdconc$_{s,m}$ *universal)
        (arg$_1$ .. arg$_m$ form-val)
        (merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b'))
     (I (mintd$_{s,1}$ .. mintd$_{s,m}$ mintd$_s$)
        (arg$_1$ .. arg$_m$ form-val)
        (merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b')))
=>
for some y in [1..x]
  (I (maxtd$_{s,y,1}$ .. maxtd$_{s,y,m}$ maxtd$_{s,y}$)
     (arg$_1$ .. arg$_m$ form-val)
     (merge-bindings (merge-bindings* b$_{1,j_1}$ .. b$_{o,j_o}$) b'))

Thus, each minimal segment, at least one of which we have already shown to satisfy C1 for any given parameter values passing the guard test, produces via unification with the CONC-ALIST some number of maximal segments, and is formed into a minimal segment/conc-alist/ maximal segment triple with each of them. We just demonstrated that if our minimal segment and our CONC-ALIST are consistent with our given values, then one of the maximal segments will be consistent, too, and thus, for one of the triples, all are consistent. The collection of triples thus formed for all the minimal segment/conc-alist pairs already computed is the complete set returned by TC-INFER, and we have guaranteed that one of them satisfies all of C2, C3, and C4. The merged binding is the binding used in each conjunct, and since we have already shown

```
  that it is an extension of b, it satisfies C5 as well.
  QED.
```

B.1 The Proof of DESCRIPTOR-FROM-QUOTE-OK

The code for DESCRIPTOR-FROM-QUOTE appears in Appendix G.7.

```
Lemma DESCRIPTOR-FROM-QUOTE-OK

For any Lisp form form, binding environment env, world world, and
non-negative integer clock clock,

(and (is-quoted-form form)
     (clock ≠ 0))
=>
(I (descriptor-from-quote form)
   (E form env world clock)
   b)
```

IS-QUOTED-FORM returns T if its argument is a self-evaluating literal (T, NIL, a character, an integer, a rational, a string, or a keyword) or an explicitly quoted form. DESCRIPTOR-FROM-QUOTE derives a descriptor for any such form.

Lemma DESCRIPTOR-FROM-QUOTE-OK is a shell for the following immediately subsidiary lemma.

```
Lemma DESCRIPTOR-FROM-QUOTED-FORM-OK

For any Lisp form form, binding environment env, world world, and
non-negative integer clock clock,

(clock ≠ 0)
=>
(I (descriptor-from-quoted-form form)
   (E (quote form) env world clock)
   b)


Proof of DESCRIPTOR-FROM-QUOTE-OK

 Case 1: form is T, NIL, a character, an integer, a rational,
         a string, or a keyword

  By the definition of E,
  (E form env world clock) = (E (quote form) env world clock)
  Using this substitution, the result follows directly from
  Lemma DESCRIPTOR-FROM-QUOTED-FORM-OK, instantiated with
  form = form.

 Case 2: form is a cons whose car is the atom quote

  The result follows directly from using Lemma
  DESCRIPTOR-FROM-QUOTED-FORM-OK instantiated with form = (cadr form).


Proof of DESCRIPTOR-FROM-QUOTED-FORM-OK
 By induction on the cons structure of forms.

 Evaluation of quoted forms requires no clock time, so (clock ≠ 0)
 guarantees the E in the conclusion will not return break-out-of-time.
 Since there are no function calls to evaluate, it will not return
 a break-guard-violation break, either.

 Case 1: form = nil
```

```
  By definition, (descriptor-from-quoted-form nil) = $nil
  By definition, (E nil env world clock) = nil
  By definition, (I $nil nil b) = t

 Cases 2: form = T, an integer, a non-integer rational, a character,
          or a string

  As with Case 1.  E returns form in each case.

 Case 3: form is of the form (cons car-form cdr-form)

    By definition,

    (descriptor-from-quoted-form (cons car-form cdr-form))
     =
    (*cons (descriptor-from-quoted-form car-form)
           (descriptor-from-quoted-form cdr-form))

    Denote the result of (descriptor-from-quoted-form car-form)
    as tdcar and the result of (descriptor-from-quoted-form cdr-form)
    as tdcdr.  Use the inductive assumptions that

    (I tdcar (E (quote car-form) env world clock) b) = t
    (I tdcdr (E (quote cdr-form) env world clock) b) = t

    By the definition of E,

    (E (quote (cons car-form cdr-form)) env world clock)
     =
    (cons car-form cdr-form)

    (E (quote car-form) env world clock) = car-form
    (E (quote cdr-form) env world clock) = cdr-form.

    Thus, by equality substitution,

    (E (quote (cons car-form cdr-form)) env world clock)
     =
    (cons (E (quote car-form) env world clock)
          (E (quote cdr-form) env world clock))

    We wish to prove

    (I (*cons tdcar tdcdr)
       (E (quote (cons car-form cdr-form)) env world clock)
       b)

    But by the equality just derived,

    (I (*cons tdcar tdcdr)
       (cons (E (quote car-form) env world clock)
             (E (quote cdr-form) env world clock))
       b)

    which, by the definition of I and our inductive assumptions,
    equals t.
 QED.
```

B.2 The Proof of TC-MAKE-ARG-CROSS-PRODUCT-OK

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
 Lemma TC-MAKE-ARG-CROSS-PRODUCT-OK
```

```
Where o is the number of parameters in the function call,
l_1 .. l_o are the lengths of the lists of segments for the actual
  parameters,
m is the number of variables in the context,
and denoting
(TC-MAKE-ARG-CROSS-PRODUCT
 (((((mintd_{1,1,1} .. mintd_{1,1,m}) -> mintd_{1,1})
    (tdconc_{1,1,1} .. tdconc_{1,1,m})
    ((maxtd_{1,1,1} .. maxtd_{1,1,m}) -> maxtd_{1,1}))
   ..
   (((mintd_{1,l_1,1} .. mintd_{1,l_1,m}) -> mintd_{1,l_1})
    (tdconc_{1,l_1,1} .. tdconc_{1,l_1,m})
    ((maxtd_{1,l_1,1} .. maxtd_{1,l_1,m}) -> maxtd_{1,l_1})))
  ...
  ((((mintd_{o,1,1} .. mintd_{o,1,m}) -> mintd_{o,1})
    (tdconc_{o,1,1} .. tdconc_{o,1,m})
    ((maxtd_{o,1,1} .. maxtd_{o,1,m}) -> maxtd_{o,1}))
   ..
   (((mintd_{o,l_o,1} .. mintd_{o,l_o,m}) -> mintd_{o,l_o})
    (tdconc_{o,l_o,1} .. tdconc_{o,l_o,m})
    ((maxtd_{o,l_o,1} .. maxtd_{o,l_o,m}) -> maxtd_{o,l_o})))))
=
(((cptda_{1,1} .. cptda_{1,m})
  (cptdr_{1,1} .. cptdr_{1,o})
  (cptdconc_{1,1} .. cptdconc_{1,m}))
 ..
 ((cptda_{q,1} .. cptda_{q,m})
  (cptdr_{q,1} .. cptdr_{q,o})
  (cptdconc_{q,1} .. cptdconc_{q,m})))

For any mintd, maxtd, and tdconc descriptors, type variable binding b,
non-negative integers l_1 .. l_n, m, and o, Lisp values arg_1 .. arg_m,
and Lisp values actual_1 .. actual_o,

for all i in [1..o],
  for some j in [1..l_i],
   (I (mintd_{i,j,1} .. mintd_{i,j,m} mintd_{i,j} tdconc_{i,j,1} .. tdconc_{i,j,m})
      (arg_1 .. arg_m actual_i arg_1 .. arg_m)
      b)
=>
for some h in [1..q],
  (I (cptda_{h,1} .. cptda_{h,m} cptdr_{h,1} .. cptdr_{h,o}
      cptdconc_{h,1} .. cptdconc_{h,m})
     (arg_1 .. arg_m actual_1 .. actual_o arg_1 .. arg_m)
     b)
```

Recall that in the setting for TC-MAKE-ARG-CROSS-PRODUCT, each list of triples in its argument is the result of the call to TC-INFER on one of the actual parameters in our function call. Each element in the value of the call to TC-MAKE-ARG-CROSS-PRODUCT is the result of combining one element from each collection of segments corresponding to the actual parameters. Thus, each is a combination of m segments. The combinator formats the segments to align descriptors for matching values and calls DUNIFY-DESCRIPTORS-INTERFACE.

Proof of TC-MAKE-ARG-CROSS-PRODUCT-OK

The proof is by induction on o, the number of parameters.

Case 1: Base case, o = 1

By definition, TC-MAKE-ARG-CROSS-PRODUCT returns its argument intact, except for minor reformatting. I.e., q = l$_1$, for each k in 1..m, each cptda$_{h,k}$ = mintd$_{1,h,k}$, each cptdr$_{h,1}$ = mintd$_{1,h}$, and each cptdconc$_{h,k}$ = tdconc$_{1,h,k}$. Thus, the antecedent in our lemma is identical to the conclusion.

Case 2: Inductive case,
We inductively assume the lemma for the first r parameters, and show that it holds for the first r+1. In our inductive assumption, we use names prefixed with semi-cptd.. rather than cptd.. to indicate distinction from the cptd..'s in the conclusion. We can use the conclusion of the inductive assumption, since its antecedent is trivially implied by our hypothesis for the r+1 case. Thus, our goal is:

(and
H1 for all i in [1..r+1],
for some j in [1..l$_i$],
(I (mintd$_{i,j,1}$ .. mintd$_{i,j,m}$ mintd$_{i,j}$ tdconc$_{i,j,1}$ .. tdconc$_{i,j,m}$)
(arg$_1$ .. arg$_m$ actual$_i$ arg$_1$ .. arg$_m$)
b)
H2 for some h,
(I (semi-cptda$_{h,1}$ .. semi-cptda$_{h,m}$
semi-cptdr$_{h,1}$ .. semi-cptdr$_{h,r}$
semi-cptdconc$_{h,1}$ .. semi-cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$ actual$_1$ .. actual$_r$ arg$_1$ .. arg$_m$)
b))
=>
for some h in [1..q],
(I (cptda$_{h,1}$ .. cptda$_{h,m}$
cptdr$_{h,1}$ .. cptdr$_{h,r+1}$
cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$ actual$_1$ .. actual$_o$ arg$_1$ .. arg$_m$)
b)

By definition of TC-MAKE-ARG-CROSS-PRODUCT, the forms

(cptda$_{h,1}$ .. cptda$_{h,m}$ cptdr$_{h,1}$ .. cptdr$_{h,r+1}$ cptdconc$_{h,1}$ .. cptdconc$_{h,m}$)

in the conclusion are computed from the forms

(semi-cptda$_{h,1}$ .. semi-cptda$_{h,m}$
semi-cptdr$_{h,1}$ .. semi-cptdr$_{h,r}$
semi-cptdconc$_{h,1}$ .. semi-cptdconc$_{h,m}$)

from H2 and the forms

(mintd$_{i,j,1}$ .. mintd$_{i,j,m}$ mintd$_{i,j}$ tdconc$_{i,j,1}$ .. tdconc$_{i,j,m}$)

from H1 in the following manner:

(semi-cptda$_{h,1}$ .. semi-cptda$_{h,m}$
semi-cptdr$_{h,1}$ .. semi-cptdr$_{h,r}$
semi-cptdconc$_{h,1}$ .. semi-cptdconc$_{h,m}$)

in H2 is extended with a *universal descriptor after semi-cptdr$_{r,h}$ to correspond to the r+1th parameter, giving

(semi-cptda$_{h,1}$ .. semi-cptda$_{h,m}$
semi-cptdr$_{h,1}$ .. semi-cptdr$_{h,r}$
***universal**
semi-cptdconc$_{h,1}$ .. semi-cptdconc$_{h,m}$)

Given H2 is true, then H2 modified in this way and with actual$_{r+1}$

inserted in the value list surely is also true, since the expansion of the call to I in H2 differs from the expansion of the call to I in the augmented form only with the conjunction of (I *universal actual$_{r+1}$ b), which is trivially true, in the augmented form. I.e.,

(I (semi-cptda$_{h,1}$ .. semi-cptda$_{h,m}$
semi-cptdr$_{h,1}$ .. semi-cptdr$_{h,r}$
semi-cptdconc$_{h,1}$ .. semi-cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$ actual$_1$ .. actual$_r$ arg$_1$ .. arg$_m$)
b)
=>
(I (semi-cptda$_{h,1}$ .. semi-cptda$_{h,m}$
semi-cptdr$_{h,1}$ .. semi-cptdr$_{h,r}$
*universal
semi-cptdconc$_{h,1}$ .. semi-cptdconc$_{h,m}$))
(arg$_1$ .. arg$_m$ actual$_1$ .. actual$_r$ actual$_{r+1}$ arg$_1$ .. arg$_m$)
b)

Similarly,
(mintd$_{r+1,j,1}$ .. mintd$_{r+1,j,m}$ tdr$_{r+1,j}$ tdconc$_{r+1,j,1}$ .. tdconc$_{r+1,j,m}$)
(from H1 instantiated with i = r+1) is padded out with a *universal for each parameter 1..r, giving

(mintd$_{r+1,j,1}$ .. mintd$_{r+1,j,m}$
*universal$_1$ .. *universal$_i$
tdr$_{r+1,j}$
tdconc$_{r+1,j,1}$ .. tdconc$_{r+1,j,m}$),

with the justification being:

(I (mintd$_{r+1,j,1}$ .. mintd$_{r+1,j,m}$ tdr$_{r+1,j}$ tdconc$_{r+1,j,1}$ .. tdconc$_{r+1,j,m}$)
(arg$_1$ .. arg$_m$ actual$_{r+1}$ arg$_1$ .. arg$_m$)
b)
=>
(I (mintd$_{r+1,j,1}$ .. mintd$_{r+1,j,m}$ *universal$_1$ .. *universal$_r$ tdr$_{r+1,j}$
tdconc$_{r+1,j,1}$ .. tdconc$_{r+1,j,m}$)
(arg$_1$ .. arg$_m$ actual$_1$ .. actual$_{r+1}$ arg$_1$ .. arg$_m$)
b)

Again, this justification is trivial, since we are only adding the requirement that for i = 1..r, (I *universal actual$_i$ b) is true.

Now we unify *dlists made from our extended descriptors using DUNIFY-DESCRIPTORS-INTERFACE. We will split into cases, depending on whether DUNIFY-DESCRIPTORS-INTERFACE returns a simple *dlist or an *or of *dlists.

Case 2.1 The result is the simple *dlist
(*dlist cptda$_1$ .. cptda$_m$ cptdr$_1$ .. cptdr$_{r+1}$ cptdconc$_1$ .. cptdconc$_m$)

We use Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK, instantiated below. Its conclusion matches the above descriptor.

(and (I (semi-cptda$_{h,1}$ .. semi-cptda$_{h,m}$
semi-cptdr$_{h,1}$ .. semi-cptdr$_{h,r}$
*universal
semi-cptdconc$_{h,1}$ .. semi-cptdconc$_{h,m}$)
(arg$_1$ .. arg$_m$ actual$_1$ .. actual$_{r+1}$ arg$_1$ .. arg$_m$)
b)
(I (mintd$_{r+1,j,1}$ .. mintd$_{r+1,j,m}$
*universal$_1$ .. *universal$_i$
tdr$_{r+1,j}$

```
          tdconc_{r+1,j,1} .. tdconc_{r+1,j,m})
         (arg_1 .. arg_m actual_1 .. actual_{r+1} arg_1 .. arg_m)
         b))
 =>
 (I (cptda_1 .. cptda_m cptdr_1 .. cptdr_{r+1} cptdconc_1 .. cptdconc_m)
    (arg_1 .. arg_m actual_1 .. actual_{r+1} arg_1 .. arg_m)
    b)
```

The hypotheses of this lemma are relieved by the previously stated I predicates on the padded forms, so we establish the conclusion, which is the conclusion of our main goal.

Case 2.2 The result is the *or of *dlists

```
    (*or (*dlist cptda_{1,1} .. cptda_{1,m}
                 cptdr_{1,1} .. cptdr_{1,r+1}
                 cptdconc_{1,1} .. cptdconc_{1,m})
         ..
         (*dlist cptda_{t,1} .. cptda_{t,m}
                 cptdr_{t,1} .. cptdr_{t,r+1}
                 cptdconc_{t,1} .. cptdconc_{t,m}))
```

In this case, the same deduction is true of the *or form as was true of the simple *dlist from Case 3.1. But we split the result into separate elements in the cross product, where each element is a disjunct of the *or. Thus, Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK gives

```
 (or (I (cptda_{1,1} .. cptda_{1,m}
         cptdr_{1,1} .. cptdr_{1,r+1}
         cptdconc_{1,1} .. cptdconc_{1,m})
        (arg_1 .. arg_m actual_1 .. actual_{r+1} arg_1 .. arg_m)
        b)
     ..
     (I (cptda_{t,1} .. cptda_{t,m}
         cptdr_{t,1} .. cptdr_{t,r+1}
         cptdconc_{t,1} .. cptdconc_{t,m})
        (arg_1 .. arg_m actual_1 .. actual_{r+1} arg_1 .. arg_m)
        b))
```

Replacing the "or" with an existential quantifier, we just say that

```
for some h in [1..t]
  (I (cptda_{h,1} .. cptda_{h,m}
      cptda_{h,1} .. cptda_{h,r+1}
      cptdconc_{h,1} .. cptdconc_{h,m})
     (arg_1 .. arg_m actual_1 .. actual_{r+1} arg_1 .. arg_m)
     b)
```

There is one such collection of results for each pair of descriptor lists selected from those over which we quantified in H1 and H2. Thus, consider any pair where the first satisfied H1 and the second satisfied H2. The result just given establishes our conclusion. QED.

B.3 Binding Extension Lemmas

The definitions of several simple functions on bindings are as follows:

```
(DEFUN EXTENDS-BINDING (B1 B2)
  (IF (NULL B2)
      T
      (AND (ASSOC (CAR (CAR B2)) B1)
           (EQUAL (CAR B2) (ASSOC (CAR (CAR B2)) B1))
           (EXTENDS-BINDING B1 (CDR B2)))))

(DEFUN MERGEABLE-BINDINGS (B1 B2)
  (IF (NULL B2)
      T
      (AND (IF (ASSOC (CAR (CAR B2)) B1)
               (EQUAL (CDR (CAR B2)) (CDR (ASSOC (CAR (CAR B2)) B1)))
               T)
           (MERGEABLE-BINDINGS B1 (CDR B2)))))

(DEFUN MERGE-BINDINGS (B1 B2)
  (IF (NULL B2)
      B1
      (IF (ASSOC (CAR (CAR B2)) B1)
          (MERGE-BINDINGS B1 (CDR B2))
          (CONS (CAR B2) (MERGE-BINDINGS B1 (CDR B2))))))

(DEFUN MERGE-BINDINGS* (BLIST)
  (IF (NULL BLIST)
      BLIST
      (MERGE-BINDINGS (CAR BLIST) (MERGE-BINDINGS* (CDR BLIST)))))
```

Here are some lemmas about these functions which are used in the proof of Lemma TC-INFER-OK.

```
Lemma EXTENDS-BINDING-MONOTONIC

 For any descriptors td, Lisp value v, and bindings b and b’,
 where b covers the type variables in td,

 (and (I td v b)
      (extends-binding b’ b))
 =>
 (I td v b’)


Proof of Lemma EXTENDS-BINDING-MONOTONIC

 By the definition of EXTENDS-BINDING, every binding in b is
 preserved in b’.  Thus, the conclusion follows trivially.


Lemma MERGE-BINDINGS-EXTENDS-BINDINGS

 For all type variable bindings b1 and b2,

 (mergeable-bindings b1 b2)
 =>
 (and (extends-binding (merge-bindings b1 b2) b1)
      (extends-binding (merge-bindings b1 b2) b2))


Proof of Lemma MERGE-BINDINGS-EXTENDS-BINDINGS

 Simple inspection of the definition of MERGE-BINDINGS shows
 b1 is included in the result in its entirety, and no new
 binding is included for a variable in the domain of b1.
 So the first conjunct of the conclusion is trivially true.
```

```
 (mergeable-bindings b1 b2) says that if a variable &i occurs in
 both b1 and b2, it is bound to the same value.  MERGE-BINDINGS
 simply cons-es into the result any entry from b2 for which
 the variable is not in the domain of b1.  So every entry
 in b2 appears in (merge-bindings b1 b2), either because it
 also appears in b1 and is hence in the result, or because,
 on account of its not being in the b1, it is cons-ed to
 the result.  Thus, the second conjunct is also trivially true.


Lemma EXTENDS-BINDING-TRANSITIVE

 For all type bindings b1, b2, and b3,

  (and (extends-binding b2 b1)
       (extends-binding b3 b2))
  =>
  (extends-binding b3 b1)


Proof of Lemma EXTENDS-BINDING-TRANSITIVE

 This is obvious.  (extends-binding b2 b1) states that every
 binding element in b1 is present in b2.  (extends-binding b3 b2)
 states that every binding element in b2, including all the
 elements from b1, is present in b3.  This establishes the
 goal.
```

B.5 The Unifier -- DUNIFY-DESCRIPTORS

This section is a collection of detailed proofs of some significant lemmas which contributed to the proof of DUNIFY-DESCRIPTORS-INTERFACE-OK (see Section 7.6). The last subsection includes proofs for some of the rules with which DUNIFY-DESCRIPTORS handles the case where both arguments are *REC descriptors. The rules are stated in Section 6.6.3.

B.4 Proof of DAPPLY-SUBST-LIST-OK

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
Lemma DAPPLY-SUBST-LIST-OK
 For any well-formed substitution s, descriptor td, binding b,
 and value v, (where td can be a *dlist containing n descriptors
 iff v is a value list of length n, in which case the call to I
 is on the list of n descriptors)

 (and (interp-substs s b) (I td v b))
 =>
 (I (dapply-subst-list s td) v b)
```

A discussion of well-formed substitutions appears in Section 6.6.3, and the definition of the function WELL-FORMED-SUBSTS, which implements the well-formedness test, is in Appendix G.8. Essentially, each substitution element which maps a variable to another variable must be downward directed in its mapping with respect to a lexical ordering for variable names. Also, no circularities may exist within the substitution.

The definition of DAPPLY-SUBST-LIST is as follows:

```
(DEFUN DAPPLY-SUBST-LIST (SUBSTS THING)
  (LET ((ATTEMPT (DAPPLY-SUBST-LIST-1 SUBSTS THING)))
    (IF (NOT (EQUAL ATTEMPT THING))
        (DAPPLY-SUBST-LIST SUBSTS ATTEMPT)
        ATTEMPT)))

(DEFUN DAPPLY-SUBST-LIST-1 (SUBSTS THING)
  (IF (NULL SUBSTS)
      THING
      (DAPPLY-SUBST-LIST-1
        (CDR SUBSTS)
        (SUBST (CDR (CAR SUBSTS))
               (CAR (CAR SUBSTS))
               THING :TEST #'EQUAL))))
```

We will factor the proof, using the following lemma, which characterizes DAPPLY-SUBST-LIST-1 rather than DAPPLY-SUBST-LIST.

```
Lemma DAPPLY-SUBST-LIST-OK-1
 For any well-formed substitution s, descriptor td, binding b,
 and value v, (where td can be a *dlist containing n descriptors
 iff v is a value list of length n, in which case the call to I
 is on the list of n descriptors)

 (and (interp-substs s b) (I td v b))
 =>
 (I (dapply-subst-list-1 s td) v b)


Proof of Lemma DAPPLY-SUBST-LIST-OK

The proof is by computational induction on the number of
computational steps in the evaluation of the top level call
to (dapply-subst-list s td).  This is a partial correctness proof,
since we are not proving termination.  Thus, we do not rule out
the possibility that the number of computational steps is
infinitely large.

Our inductive assumption, which we can apply on any subsidiary
call of dapply-subst-list, is the lemma itself.

Case 1.  (dapply-subst-list-1 s td) = td

 Trivial.  In this case (dapply-subst-list s td) = td and
 (dapply-subst-list s td) = td.  So the conclusion is equal
 to the second hypothesis.

Case 2.  (dapply-subst-list-1 s td) ≠ td

 In this case, in the conclusion,

  (dapply-subst-list s td)
   =
  (dapply-subst-list (dapply-subst-list-1 s td) td)

 Apply Lemma DAPPLY-SUBST-LIST-OK-1, instantiated with namesakes.
 Since its hypotheses are the hypotheses of our goal, we establish
 its conclusion,

  (I (dapply-subst-list-1 s td) v b)

 Now apply our inductive assumption, instantiated with
 td = (dapply-subst-list-1 s td), s = s, v = v, and b = b.
 This establishes the conclusion.  QED.


Proof of Lemma DAPPLY-SUBST-LIST-OK-1
```

By induction on the length of s.

Case 1: (Base case) length of s = 0

Trivial. (dapply-subst-list-1 s td) = td. The conclusion of the lemma is equal to the second hypothesis.

Case 2: (Inductive step) Assume the lemma for length of s = n and prove it where length of s = n + 1

Our goal is

(and
H1 (interp-substs (s_1 .. s_{n+1}) b)
H2 (I td v b)
H3 (and (interp-substs (s_1 .. s_n) b) (I td v b))
=>
(I (dapply-subst-list-1 (s_1 .. s_n) td) v b))
=>
(I (dapply-subst-list-1 (s_1 .. s_{n+1}) td) v b)

We can use the conclusion of our inductive hypothesis, because the expansion of interp-substs in H1 results in a conjunction whose conjuncts are a superset of those in the expansion of (interp-substs (s_1 .. s_n) b) in the inductive hypothesis. So our goal is now

(and
H1 (interp-substs (s_1 .. s_{n+1}) b)
H2 (I td v b)
H3 (I (dapply-subst-list-1 (s_1 .. s_n) td) v b))
=>
(I (dapply-subst-list-1 (s_1 .. s_{n+1}) td) v b)

Let us denote s_{n+1} = (&i . td_i)

Case 2.1 &i does not appear in (dapply-subst-list-1 (s_1 .. s_n) td)

Trivial. By definition,

(dapply-subst-list-1 (s_1 .. s_{n+1}) td)
=
(dapply-subst-list-1 (s_1 .. s_n) td)

Therefore, the goal is equal to the inductive hypothesis.

Case 2.2 &i appears in (dapply-subst-list-1 (s_1 .. s_n) td)

Expanding (dapply-subst-list-1 (s_1 .. s_{n+1}) td) gives us

(subst &i td_i (dapply-subst-list-1 (s_1 .. s_n) td)

Now consider Lemma APPLY-SUBST-OK, which is stated just below. Using this lemma, instantiated with td = (dapply-subst-list-1 (s_1 .. s_n) td) and all other variables with their namesakes. H2 is equal to its first hypothesis, and its second hypothesis is the conjunct representing &i in the expansion of H1. So we can use its conclusion, which establishes our goal. QED.

Lemma APPLY-SUBST-OK

For any type variable &i, descriptor td containing &i, descriptor td_i not containing &i, Lisp value v, and type variable binding b covering the variables in td,

```
   (and
H1   (I td v b)
H2   (I td_i (cdr (assoc &i b)) b) )
   =>
   (I (subst td_i &i td) v b)
```

Proof of Lemma APPLY-SUBST-OK

We present two arguments. The first is somewhat operational in style and provides an intuitive approach. A more rigorous account follows. The operational account goes as follows.

Expand both calls of I in unison down to the point where we encounter &i in td. To this point, td and (subst td_i &i td) are equivalent, so the predicates generated in the evaluation of I with respect to td and (subst td_i &i td) are identical. At this point, though, where in td we encounter &i, we encounter td_i in (subst td_i &i td). Since we have been descending into the structure of v as we descend into td and (subst td_i &i td), let us also denote that we are considering the value v_i (just to give it a name). We need to demonstrate, then, that (I &i v_i b) => (I td_i v_i b). By the definition of I, (I &i v_i b) = (equal (cdr (assoc &i b))) v_i). Substitute this equality into the conclusion, (I td_i v_i b) to get (I td_i (cdr (assoc &i b)) b). This is equal to our second hypothesis, so we have succeeded with this joint encounter of &i in td and td_i in (subst td_i &i td). Proceed through the rest of the form in the same manner. An important thing to note when considering the correctness of this argument is that the descriptor language and the interpreter I are such that the local replacement of &i in td by td_i produces a strictly local effect on the evaluation of I with respect to the two descriptors.

Now for the more rigorous argument.

By induction on the structure of descriptors.

Case 1. td is one of $character, $integer, $nil, $non-integer-rational $non-t-nil-symbol, $string, $t, *empty, or *universal

(subst td_i &i td) = td, equating the conclusion with the first hypothesis.

Case 2. td is of the form (*cons tdcar tdcdr)

We use our lemma as an inductive hypothesis on both tdcar and tdcdr. In the first case, we instantiate with td = tdcar, v = (car v), b = b, and td_i = td_i. In the latter case, td = tdcdr and v = (cdr v). In each case, the antecedents are easily relieved, since the second antecedent is identical to H2, and the first antecedent is in the immediate expansion of H1. This gives us the new hypotheses H3 and H4 in the goal:

```
     (and
H1     (I td v b)
H2     (I td_i v b)
H3     (I (subst td_i &i tdcar) (car v) b)
H4     (I (subst td_i &i tdcdr) (cdr v) b) )
     =>
     (I (subst td_i &i td) v b)
```

By the definition of subst, the conclusion expands to

```
(I (*cons (subst td_i &i tdcar) (subst td_i &i tdcdr)) v b)
```

And by the expansion of I, to

(and (consp v)
(I (subst td$_i$ &i tdcar) (car v) b)
(I (subst td$_i$ &i tdcdr) (cdr v) b))

The expansion of H1 gives (consp v), and the other two conjuncts are from our inductive hypotheses.

Case 3. td is of the form (*dlist dl-td$_1$ .. dl-td$_n$)
(and hence v = (v$_1$ .. v$_n$))

The proof is as in Case 2, using an inductive hypothesis for each dl-td$_i$ and v$_i$, and without having to worry about the (consp v) test.

Case 4. td is of the form (*or or-td$_1$ .. or-td$_n$)

We employ the inductive hypothesis for each or-td$_i$, giving

H3 for each i in 1..n
(and (I or-td$_i$ v b)
(I td$_i$ (cdr (assoc &i b)) b))
=> (I (subst td$_i$ &i or-td$_i$) v b)

H1 expands to

(or (I or-td$_1$ v b) .. (I or-td$_n$ v b))

and the conclusion to

(or (I (subst td$_i$ &i or-td$_1$) v b)
..
(I (subst td$_i$ &i or-td$_1$) v b))

For whichever disjuncts are true in the expansion of H1, note that we have satisfied the antecedents of the inductive hypothesis for that or-td$_i$. Employing its conclusion establishes our goal.

Case 5. td is a *rec form

Using canonicalization Rule 2, open both occurrences of the *rec descriptor.

(and
H1 (I (open-rec-descriptor-absolute td) v b)
H2 (I td$_i$ (cdr (assoc &i b)) b))
=>
(I (subst td$_i$ &i (open-rec-descriptor-absolute td)) v b)

This transforms this case to one of the other cases.

QED.

B.5 Proof of DUNIFY-DESCRIPTORS-OK

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

Lemma DUNIFY-DESCRIPTORS-OK
Denoting (dunify-descriptors tda tdb substs)
by ((td$_1$. substs$_1$) .. (td$_n$. substs$_n$)),
for all v and fully instantiating b,

```
  (and
H1  (interp-substs substs b)
H2  (I tda v b)
H3  (I tdb v b))
  =>
  for some i,
    (and (interp-substs substs_i b)
         (I td_i v b))
```

For brevity in the exposition of the proof, we will abbreviate SUBSTS with "s", and INTERP-SUBSTS with "I-S".

Note that this lemma is part of the large nest of unification lemmas which are being proved together by computational induction. These lemmas include DUNIFY-DESCRIPTORS-INTERFACE-OK, DUNIFY-DESCRIPTORS-OK, DMERGE-OK, DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST-OK, DMERGE-SECOND-ORDER-OK, DMERGE-DUNIFIED-FORMS-INTO-SUBSTS-OK, and the various lemmas which are associated with the special case rules for unifying pairs of *REC descriptors. Within this collection of proofs, we may use any of these lemmas as an inductive assumption to characterize a subsidiary call of a function in question. A discussion of the induction appears in Section 7.6.

```
Proof of Lemma DUNIFY-DESCRIPTORS-OK

The case structure presented in this proof reflects that of the
algorithm itself.  In each case, we assume none of the previous
cases were appropriate.  For example, Case 1 is where tda = tdb.
For all later cases, we can assume that tda ≠ tdb.

The algorithm and the proof are somewhat symmetric in their case
analysis, in the following sense.  Whenever we consider a top
level case where tda is some particular kind of descriptor (for
example, Case 2 is where tda is *empty), the next top level case
is where we consider tdb to be the same kind of descriptor.  The
only assymetry is that in any tdb case, we know that tda is not
the same kind of descriptor, since we just considered that case
immediately before, and can therefore assume its negation.  Since
the proofs of the tdb cases will follow the same line of reasoning
as the tda case preceding it, we generally omit the argument to
save from being tedious.

Case 1. tda = tdb
 By definition, (dunify-descriptors tda tdb s) = ((tda . s))
 Trivial, since H1 = C1 and H2 = C2.

Case 2. tda = *empty
 By definition, (dunify-descriptors *empty tdb s) = nil
 Trivial, since we have a false hypothesis, (I *empty v b).

Case 2'. tdb = *empty
 As with Case 1.

Case 3. tda = *universal
 By definition, (dunify-descriptors *universal tdb s) = ((tdb . s))
 Trivial, since C1 is equivalent to H1 and C2 is equivalent to H3.

Case 3'. tdb = *universal
 As with Case 2.

Case 4. tda = (*or tda_1 .. tda_n)

 Case 4.1  for some i, tda_i = tdb
  By definition, (dunify-descriptors tda tdb s) = ((tdb . s))
  This case is trivial, since H1 = C1 and H3 = C3.

 Case 4.2  for all i, tda_i ≠ tdb
```

```
  By definition, DUNIFY-DESCRIPTORS unifies each tda_i with tdb.
  Ascribe the notation

   (dunify-descriptors tda_j tdb s)
    =
   ((tda_j b_1 . s_j,1) .. (tda_j b_m_j . s_j,m_j))

  to characterize the definition of DUNIFY-DESCRIPTORS for each tda_j,
  indicating that each unification produces a result with m_j
  different (descriptor . subst) pairs.  So by definition,

   (dunify-descriptors tda tdb s)
    =
   ((tda_1 b_1 . s_1,1) .. (tda_1 b_m_1 . s_1,m_1)
    ..
    (tda_n b_1 . s_n,1) .. (tda_n b_m_n . s_n,m_n))

  Let us apply our lemma inductively for each i, i.e.,

  For each j,
   (and (I-S s b) (I tda_j v b) (I tdb v b))
   =>
   For some i in [1..m_j]
    (and (I-S s_j,i b) (I tda_j b_i v b))

  H2 expands by definition of I to

   (or (I tda_1 v b) .. (I tda_n v b))

  For whichever of these disjuncts holds, instantiate our inductive
  hypothesis.  H1, H2, and H3 relieve its antecedents, so we can
  use its conclusion, which in turn establishes our goal.

Case 4'  tdb = (*or tdb_1 .. tdb_n)
 As with Case 3.

 Many of the cases within Case 5 will depend on lemmas about
 dmerge-new-substs.  We need a lemma showing that any subst
 produced by dmerge-new-subst is well-formed.

Lemma DMERGE-OK

  For any variable &i, descriptor td, and substitution list s,
  and denoting (dmerge-new-subst &i td s) = (s_1 .. s_n)

H1 (and (I-S s b)
H2      (I &i v b)
H3      (I td v b))
   => for some k in [1..n], (I-S s_k b)

 The proof of Lemma DMERGE-OK is in Appendix B.6.

Case 5.  tda = &i

 Case 5.1  tdb = &j, where i < j,
  By definition,
  (dunify-descriptors &i &j s) = ((&i . s_1) .. (&i . s_n))
  where (dmerge-new-subst &j &i s) = (s_1 .. s_n)

  Our goal is:
H1 (and (I-S s b)
H2      (I &i v b)
H3      (I &j v b))
  =>
  For some k,
C1  (and (I-S s_k b)
```

C2 (I &i v b))

C2 follows from H2. We need to establish (I-S s_k b).
Lemma DMERGE-OK, instantiated with s = s, &i = &i, v = v, b = b, and td = &j gives us C1.

Case 5.2 tdb = &j, where i > j,
By definition,
(dunify-descriptors &i &j s) = ((&j . s_1) .. (&j . s_n))
where (dmerge-new-subst &i &j s) = (s_1 .. s_n)

The proof is analogous to Case 5.1, with &i and &j switching places.

Case 5.3 tdb is a non-variable descriptor containing &i

Case 5.3.1 tdb is a *rec descriptor whose body is an *or in which &i is a disjunct.[36]

By definition, (dunify-descriptors &i tdb s) = ((&i . s))
Clearly, the theorem is true in this case. H1 guarantees C1 and H2 guarantees C2.

Case 5.3.2 otherwise, i.e., tdb is a *cons descriptor with an embedded &i, or it is a *rec descriptor in which &i appears, but not as a disjunct of a top-level *or, regardless of how the *rec can be expanded. (See the footnote on the previous case.)

Thus, the &i must be embedded in a *cons. (If tdb is a *rec, we know by assumption that &i cannot be a terminating disjunct. I.e, if we raised all disjunction in the *rec body to the top, &i could not be a top-level disjunct, and if any nested *rec descriptor appears as a top-level disjunct and its body is treated similarly, &i could not appear as one of its top-level disjuncts, and so on recursively. Therefore, any occurrence of &i must appear within a *cons.)

By definition in this case,

```
(dunify-descriptors &i tdb) =
  (dunify-descriptors &i (screen-var-from-descriptor &i tdb))
```

where screen-var-from-descriptor effectively replaces each occurrence of &i by *empty and canonicalizes, renaming any *rec descriptor whose body changed in the process.[37] Consider:

Lemma UNIFY-VAR-EMBEDDED-VAR

Where by (*cons .. &i ..), we mean a *cons descriptor in which a variable &i appears, separated from the top of the descriptor only by encasing *cons descriptors, (For example, (*cons &i td) or (*cons td1 (*cons td2 &i)))

```
(I &i v b)
=>
(not (I (*cons .. &i ..) v b))
```

Proof of Lemma UNIFY-VAR-EMBEDDED-VAR

36

Examples:

```
 (*rec foo (*or &i (*cons $t $nil) (*cons *universal (*recur foo))))

 (*rec bim (*or (*rec bar (*or &i (*cons $integer (*recur bar))))
                (*cons $character (*recur bim))))
```

37

The code implementing this case is presented in Appendix G.10.

```
  The hypothesis requires (equal v (cdr (assoc &i b))).
  (I (*cons .. &i ..) v b) requires
  (equal <some CAR/CDR nest of v> (cdr (assoc &i b))), where the
  CAR/CDR nest is the one constructed by the unwinding of
  (I (*cons .. &i ..) v b).  Since no CONS data value can
  contain a component which is equal to the CONS itself, this is a
  contradiction.  QED.

 This lemma establishes that we can reduce the problem with
 screen-var-from-descriptors, since it simply removes from
 consideration cases which would require (I &i v b) and
 (I (*cons .. &i ..) v b) to be simultaneously true.  Thus
 transformed, we use our inductive assumption of
 DUNIFY-DESCRIPTORS-OK on the recursive call, establishing
 the goal.

Case 5.4  tdb is a *rec descriptor in which some variable other than
          &i can be a top level disjunct.

  By definition,
   (dunify-descriptors &i tdb s)
    = (dunify-descriptors &i (open-rec-descriptor-absolute tdb)[38] s)

  OPEN-REC-DESCRIPTOR-ABSOLUTE simply represents the invocation of
  Canonicalization Rule #2, which is justified simply by opening up
  the definition of I on the *rec form.  This reduces the problem
  to one of those already considered, where tdb is either an *or
  or a *cons, and we are done by the inductive hypothesis.

Case 5.5  tdb is a non-variable not containing &i,

 In this case, by definition

  (dunify-descriptors &i tdb s) = ((&i . s1) .. (&i . sn))

 where (dmerge-new-subst &i tdb s) = (s1 .. sn).

 Our theorem is
H1 (and (I-S s b)
H2      (I &i v b)
H3      (I tdb v b))
 =>
 For some k,
C1  (and (I-S sk b)
C2       (I &i v b))

 As usual, C2 follows from H2.  We need to establish (I-S sk b).
 Lemma DMERGE-OK, instantiated with s = s, &i = &i, v = v, b = b,
 and td = tdb gives us C1.

Case 5'.  tdb is a variable.
 As with Case 5.

Case 6.  tda = (*rec foo foobody)

Case 6.1  tdb = (*rec bar barbody)
  This is not proven in general, but by virtue of application of a
  collection of reduction rules, each tried in sequence until one of
  them is found to be eligible and is used.  Each reduction rule is
  either proven to be sound with respect to our semantics for
  DUNIFY-DESCRIPTORS or could be proven sound by application of
  similar proof techniques.  The final reduction rule in the
  collection is unconditionally eligible, so some reduction rule
```

---

[38] OPEN-REC-DESCRIPTOR-ABSOLUTE returns the body (or CADDR) of tdb with the *RECUR form for the *REC replaced by tdb.

```
   will always produce a result, and the proof of soundness for this
   case is therefore the conjoined proofs for all the rules.  The
   rules are named Rule Dunify-*Reci for some integer i, and are
   stated and proved in Appendix B.7.

 Case 6.2  tdb is not a *rec descriptor
   In this case, DUNIFY-DESCRIPTORS heuristically chooses one of two
   different actions, each of which is demonstrated sound by the
   arguments below.

   Choice 6.2.1:
     Apply Canonicalization Rule 2 (which opens *rec descriptors) to
     tda.  Thus, (dunify-descriptors tda tdb s) =
      (dunify-descriptors (open-rec-descriptor-absolute tda) tdb s)

     The justification is the canonicalization rule proof that
     (I (*rec ..) v b)
     =>
     (I (open-rec-descriptor-absolute (*rec ..) v b)

     With the problem thus transformed, it falls under one of the
     other cases, and we are done by the inductive hypothesis.

   Choice 6.2.2: (punt)
     (dunify-descriptors tda tdb s) = ((tdb . s))
     This is trivially sound, as H1 is equivalent to C1 and C2 is
     equivalent to H3.  (This case represents discovery of a TERM-RECS
     loop.  See the discussion in Section 6.6.3.

Case 6'.  tdb = (*rec foo foobody)
 As with Case 6.2.

Case 7.  tda = (*cons tdacar tdacdr)

  Case 7.1  tdb is one of {$character, $integer, $nil,
                           $non-integer-rational, $non-t-nil-symbol,
                           $string $t}

    By definition,
      (dunify-descriptors (*cons tdacar tdacdr) tdb s) = ()

    Our conjecture is trivially true, since expanding H2 gives us

      (and (consp v) (I tdacar (car v) b) (I tdacdr (cdr v) b))

    and (consp v) is mutually exclusive with the expansion of H3 in
    each of the 7.1 cases, namely (characterp v), (integerp v), etc.

  Case 7.2  tdb = (*cons tdbcar tdbcdr)

    This can be treated analogously to Case 8 below, as a special
    case where the number of descriptors in each *DLIST is two.
    Although the flow of control through our I is down a different
    path in the algorithm, the procedure followed is virtually
    the same.  The I predicate for this case is

     (I (*cons tdcar tdcdr) (cons v1 v2) b)
     =
     (and (consp (cons v1 v2)) (I tdcar v1 b) (I tdcdr v2 b))
```

    Whereas for (*dlist td$_1$ .. td$_n$), and values v$_1$ .. v$_n$,

     (I (td$_1$ .. td$_n$) (v$_1$ .. v$_n$) b)
     =
     (and (I td$_1$ v$_1$ b) .. (I td$_n$ v$_n$ b)

```
    The only extra baggage is the consp test, which is trivially true.
```

Case 7'. tdb = (*cons tdbcar tdbcdr)
As with Case 7.1.

Case 8. tda = (*dlist tda$_1$ .. tda$_n$) and
tdb = (*dlist tdb$_1$ .. tdb$_n$)

Comment: *dlist forms are never nested within any other descriptors. Furthermore, they are only unified with other *dlists of the same length.

We denote that
(dunify-descriptors
(*dlist tda$_1$ .. tda$_n$) (*dlist tdb$_1$ .. tdb$_n$) s)
= (((*dlist tdab$_{1,1}$ .. tdab$_{n,1}$) . s$_1$)
..
((*dlist tdab$_{1,m}$ .. tdab$_{n,m}$) . s$_m$))}

Thus, the conjecture we wish to prove is:

For all (v$_1$ .. v$_n$) and b,
H1 (and (I-S s b)
H2 (I (tda$_1$ .. tda$_n$) (v$_1$ .. v$_n$) b)
H3 (I (tdb$_1$ .. tdb$_n$) (v$_1$ .. v$_n$) b))
=>
For some i in 1..m,
(and (I-S s$_i$ b) (I (tdab$_{1,i}$.. tdab$_{n,i}$)) (v$_1$ .. v$_n$) b))

Let us use the following notation:

(dunify-descriptors tda$_i$ tdb$_i$ s)
=
((tdab$_{i,1}$. s$_{i,1}$) .. (tdab$_{i,o}$. s$_{i,o}$))

The proof will be by induction on the length of the *dlist.

Base case: n = 0
By definition,
(dunify-descriptors (*dlist) (*dlist) s) = (((*dlist) . s))

Since I reduces immediately to true when given NIL arguments, we are left with the trivial inference:
(and (I-S s b) t t) => (I-S s b)

Inductive case: Assume the theorem for n - 1, prove it for n.

Let us denote

(dunify-descriptors
(*dlist tda$_1$ .. tda$_{n-1}$) (*dlist tdb$_1$ .. tdb$_{n-1}$) s)
= (((*dlist tdab$_{1,1}$ .. tdab$_{n-1,1}$) . s$_{n-1,1}$)
..
((*dlist tdab$_{1,p}$ .. tdab$_{n-1,p}$) . s$_{n-1,p}$))}

and for each k in 1..p
(dunify-descriptors tda$_n$ tdb$_n$ s$_{n-1,k}$)
= ((tdab$_{n,k,1}$. s$_{n,k,1}$) .. (tdab$_{n,k,m_k}$. s$_{n,k,m_k}$))

By definition,
(dunify-descriptors
(*dlist tda$_1$ .. tda$_n$) (*dlist tdb$_1$ .. tdb$_n$) s)
= the append for all k in 1..p of
(((*dlist tdab$_{1,k}$ .. tdab$_{n-1,k}$ tdab$_{n,k,1}$) . s$_{n,k,1}$)
..
((*dlist tdab$_{1,k}$ .. tdab$_{n-1,k}$ tdab$_{n,k,m_k}$) . s$_{n,k,m_k}$))

Our inductive hypothesis for

(dunify-descriptors
  (*dlist tda$_1$ .. tda$_{n-1}$) (*dlist tdb$_1$ .. tdb$_{n-1}$) s)
gives us

H4 For all v and b,
  (and (I-S s b)
    (I (tda$_1$ .. tda$_{n-1}$) (v$_1$ .. v$_{n-1}$) b)
    (I (tdb$_1$ .. tdb$_{n-1}$) (v$_1$ .. v$_{n-1}$) b))
  =>
  For some k in 1..p,
  (and (I-S s$_{n-1,k}$ b) (I (tdab$_{1,k}$ .. tdab$_{n-1,k}$) (v$_1$ .. v$_{n-1}$) b))

We assume DUNIFY-DESCRIPTORS-OK inductively on subsidiary calls, including the call

(dunify-descriptors tda$_n$ tdb$_n$ s$_{n-1,k}$)

Thus, the inductive hypothesis gives us:

H5 For any k in 1..p, v and b,
  (and (I-S s$_{n-1,k}$ b) (I tda$_n$ v b) (I tdb$_n$ v b))
  =>
  For some j in 1..m$_k$,
    (and (I-S s$_{n,k,j}$ b) (I tdab$_{n,k,j}$ v b))

H1, H2, and H3 relieve the antecedents for H4, since the expansions of I and I-S in H1, H2, and H3 establish a superset of the predicates produced by the expansions of those calls in the hypotheses for H4. Thus, we establish the conclusion in H4.

Now we would like to employ H5. Consider the cases k in 1..p satisfying the predicate in the conclusion of H4. The first conjunct of this predicate establishes the (I-S s$_{n-1,k}$ b) hypothesis in H5. The final conjunct in the expansions of I in H2 and H3 establish its second and third hypothesis, respectively. Thus, we establish the conclusion in H5 for some k and j.

Finally, the second conjunct in the conclusion of H4, plus the second conjunct of the conclusion of H5 establish the second conjunct of our goal, and for that same j and k, the first conjunct of H5 establishes the first conjunct of the goal, and we are done.

Case 9. tda = $integer; Otherwise without loss of generality, suppose the analogous proof will hold true for the cases where tda is $character, $nil, $non-integer-rational, $non-t-nil-symbol, $string, or $t.

Case 9.1 tdb in {$character $nil $non-integer-rational $non-t-nil-symbol $string $t}

(dunify-descriptors $integer tdb s) = ().
Trivially true, since (I $integer v b) = (integerp v), and (integerp v) cannot be true if (I tdb v b) is true. ((I tdb v b) = (characterp v) when tdb = $character, etc.) Thus we have a false hypothesis.

Case 9.2 tdb = (*cons tdcar tdcdr) for any tdcar and tdcdr.
(dunify-descriptors $integer tdb s) = ()
Trivially true, as with Case 7.1.

This completes an exhaustive case analysis.

QED.

B.6 Proof of DMERGE-OK

Note that DMERGE-OK and several other lemmas proved in this section are part of the large nest of unification lemmas which are being proved together by computational induction. These lemmas include DUNIFY-DESCRIPTORS-INTERFACE-OK, DUNIFY-DESCRIPTORS-OK, DMERGE-OK, DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST-OK, DMERGE-SECOND-ORDER-OK, DMERGE-DUNIFIED-FORMS-INTO-SUBSTS-OK, and the various lemmas which are associated with the special case rules for unifying pairs of *REC descriptors. Within this collection of proofs, we may use any of these lemmas as an inductive assumption to characterize a subsidiary call of a function in question. A discussion of the induction appears in Section 7.6.

The code implementing DMERGE-NEW-SUBST is in Appendix G.9.

```
Lemma DMERGE-OK

  For any variable &i, descriptor td, and substitution list s,
  and denoting (dmerge-new-subst &i td s) = (s1 .. sn)

H1 (and (I-S s b)
H2      (I &i v b)
H3      (I td v b))
   => for some k in [1..n], (I-S sk b)
```

To prove this lemma, we will need some lemmas about subsidiary functions within DMERGE-NEW-SUBST. DINSERT-SUBST simply inserts a new substitution element for &i into a substitution in which &i is not already mapped.

```
Lemma DINSERT-SUBST-OK

 For any substitution s, binding b, variable &i, Lisp value v,
 and descriptor td,

H1  (and (I-S s b)
H2       (I &i v b)
H3       (I td v b)
    => (I-S (dinsert-subst &i td s) b)

Proof of Lemma DINSERT-SUBST-OK
 This is easy.  (I-S s b) guarantees the I-S predicate for every
 element of (dinsert-subst &i td s) except the one for &i.  For
 &i we need to know (I td (cdr (assoc &i b)) b).  H2 gives us
 (equal (cdr (assoc &i b)) v), which we substitute into the H3 to
 equate it with the goal.

Lemma DREPLACE-SUBST-OK

 For any type variable &i, substitution s with &i in its domain,
 descriptor td, binding b, and Lisp value v,

H1  (and (I-S s b)
H2       (I &i v b)
H3       (I td v b)
    => (I-S (dreplace-subst &i td s) b)

 By definition, (dreplace-subst &i td ( .. (&i . tdold) .. )
 simply replaces the entry (&i . tdold) with (&i . td).
 The proof of DREPLACE-SUBST-OK is virtually identical to the proof
 of DINSERT-SUBST-OK.


Lemma DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST-OK

 For any non-negative integer o, well-formed substitutions
```

```
 s1 .. so and dus, and binding b,
 and denoting, where (dutd . dus) is some dunified form, and
  &i is a variable in the domain of dus whose binding is tdi,
  (DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST &i (dutd . dus) (s1 .. so))
    = (sm1 .. smn)

    (and
H1    (I-S dus b)
H2    for some k in [1..o], (I-S sk b) )
    => for some i in [1..n], (I-S smi b)
```

Proof of DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST-OK

```
  We will induct on o, the length of SUBSTS-LIST.  If o = 0, then
  H2 is false, and the lemma is trivially true.  Now we
  will assume the result for o = m and prove it for o = m + 1.
  Our inductive hypothesis is:

H3  (and (I-S dus b)
         for some k in [1..m], (I-S sk b))
    => for some i in [1..n], (I-S smi b)

  By definition,
  (DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST &i (td . s) (s1 .. so))
  is the append of the results of
  (DMERGE-NEW-SUBST &i (cdr (assoc &i s)) sj) for each j in 1..o.
  So we will also want to use an inductive hypothesis on
  DMERGE-NEW-SUBST, which is Lemma DMERGE-OK:

H4  For any variable &i, descriptor td, and substitution list s,
    where we denote (dmerge-new-subst &i td s) = (sdm1 .. sdmn)
     (and (I-S s b)
          (I &i v b)
          (I td v b))
     => for some j in [1..n], (I-S sdmj b)

  H1, as it applies to the variable &i in the domain of dus, gives us

H5    (I tdi (cdr (assoc &i b)) b)

  Since we are trying to prove our lemma for the case where o = m + 1,
  H2 gives us:

   for some k in [1..m+1], (I-S sk b)

  Case 1.  (I-S sm+1 b) is not true.
    Then H2 equates to the second hypothesis of H3, so we can use
    the conclusion of H3.  Since the substs over which it quantifies
    are part of the result, we have established the goal directly.

  Case 2.  (I-S sm+1 b) is true
    Then we can instantiate H4 with s = sm+1, b = b,
    v = (cdr (assoc &i b)), td = tdi, and &i = &i.  We can use the
    conclusion of H4 because our case assumption guarantees its first
    hypothesis, and H5 guarantees its third hypothesis.  The
    second antecedent, (I &i v b), expands to
    (equal v (cdr (assoc &i b))), which is established by our
    instantiation of v.  So we have:

     for some j, (I-S sdmj b)

    Since these substs are part of the result, one of them will
    satisfy the conclusion.  QED
```

Lemma DMERGE-SECOND-ORDER-OK

```
  For all descriptors dutd, substitution dus, and binding b, such
  that (&1 .. &p) are variables in the domain of dus[39] and
  (td1 .. tdp) are the bindings for (&1 .. &p) in dus,[40]
  and denoting
   (DMERGE-SECOND-ORDER-SUBSTS (&1 .. &p) (dutd . dus) (s1 .. sm))
     = (sm1 .. smn),

H1  (and (I-S dus b)
H2       for some k in [1..m], (I-S sk b))
    => for some i in [1..n], (I-S smi b)
```

Proof of DMERGE-SECOND-ORDER-OK

We will induct on p. If p = 0, the result (sm_1 .. sm_n) is equal to (s_1 .. s_m). Thus, H2 equates with the conclusion.

Now, we will assume the result for p = j and prove the result for p = j + 1. Our inductive hypothesis, then, is

```
H3       (and (I-S dus b)
              for some k in [1..m], (I-S sk b))
         => for some i in [1..o], (I-S smji b)

 where (DMERGE-SECOND-ORDER-SUBSTS (&1 .. &j) (dutd . dus) (s1 .. sm))
        = (smj1 .. smjo)
```

H1 satisfies the first hypothesis of H3, H2 the second, so we get to use the conclusion of H3. By definition (using &j+1 as a variable name),

```
 (DMERGE-SECOND-ORDER-SUBSTS (&1 .. &j+1) (dutd . dus) (s1 .. sm))
  =
 (DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST
    &j+1 (dutd . dus) (smj1 .. smjo))
```

So we wish to use Lemma DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST-OK, just proved above. We instantiate it with &i = &j+1 and (s_1 .. s_o) = (smj_1 .. smj_o). Our H1 satisfies its first hypothesis. Our specification that &j+1 is in the domain of dus satisfies its requirement that &i is in the domain of dus, and the conclusion of H3 satisfies its H2. So we can use its result to characterize the value returned by

```
 (DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST
     &j+1 (dutd . dus) (smj1 .. smjo)),
```

giving us

```
  for some i in [1..n], (I-S smi b)
```

By definition, the result returned by DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST-OK is the result we return. So this satisfies our conclusion. QED.

Lemma DMERGE-DUNIFIED-FORMS-INTO-SUBSTS-OK

For any substitution list s, binding b, descriptors tds_1 .. tds_m, and substitutions ss_1 .. ss_m, such that the variable

---

39 This is a notational convenience which may be somewhat misleading. These variables need not be contiguously named, nor is &1 required to be among them. We mean merely to say that there is a list of variable names of length p, each of which is in the domain of dus.

40 It is known that the bindings of those variables are different from the ones in the s_1 .. s_m.

&i is bound in s, and denoting
(dmerge-dunified-forms-into-substs
((tds$_1$. ss$_1$) .. (tds$_m$. ss$_m$)) &i s)
= (s$_1$ .. s$_n$)

H1 (and (I-S s b)
H2 for some j in [1..m], (and (I-S ss$_j$ b) (I tds$_j$ v b))
H3 (I &i v b))
=> for some k in [1..n] (I-S s$_k$ b)

Proof of DMERGE-DUNIFIED-FORMS-INTO-SUBSTS-OK

DMERGE-DUNIFIED-FORMS-INTO-SUBSTS simply maps the function DMERGE-DUNIFIED-FORM-INTO-SUBSTS over the pairs in its first argument, appending all the results. We will consider whether DMERGE-DUNIFIED-FORM-INTO-SUBSTS preserves the I-S property when the pair it is given satisfies the I and I-S properties.

Case 1. ss$_j$ contains no substs (other than for &i) which differ from s.

In this case, by definition,

(dmerge-dunified-form-into-substs (tds$_j$. ss$_j$) &i s)
= ((dreplace-subst &i tds$_j$ s))

I.e., we just replace the binding for &i in s with tds$_j$.
By Lemma DREPLACE-SUBST-OK, since we have a case where
(I tds$_j$ v b) and (I-S s b) are true, then
(I-S (dreplace-subst &i tds$_j$ s) b) is true, establishing our goal.

Case 2. ss$_j$ contains substs (other than for &i) which differ from s.

In this case, by definition,

(dmerge-dunified-form-into-substs (tds$_j$. ss$_j$) &i s)
=
(dreplace-subst-in-second-order-substs
&i tds$_j$ (dmerge-second-order-substs diff-substs (tds$_j$. ss$_j$) (s)))

where diff-substs is the list of variables whose bindings in ss$_j$ differ from those in s.

So we wish to use Lemma DMERGE-SECOND-ORDER-OK, instantiating for the j satisfying H2 with (&1 .. &p) = diff-substs, dutd = tds$_j$, dus = ss$_j$, (s$_1$.. s$_n$) = the singleton list (s), and b = b.
Our H1 establishes its H2, and our H2 for the appropriate j satisfies its H1, so we can use its conclusion, giving us (where (mso$_1$ .. mso$_o$) = the substitution lists returned by DMERGE-SECOND-ORDER-SUBSTS)

H4 for some l in [1..o], (I-S mso$_l$ b)

By definition,

(dreplace-subst-in-second-order-substs &i tds$_j$ (mso$_1$ .. mso$_o$))
=
((dreplace-subst &i tds$_j$ mso$_1$)
..
(dreplace-subst &i tds$_j$ mso$_o$))

So consider Lemma DREPLACE-SUBST-OK. Instantiated for the l which suffices for H4 with s = mso$_l$, &i = &i, v = v, b = b, and td = tds$_j$, it gives us

(and (I-S mso$_l$ b)

```
       (I &i v b)
       (I tds_j v b)
  => (I-S (dreplace-subst &i tds_j mso_l) b)

  H4 relieves its first hypothesis, H3 its second, and H2 its third.
  Thus, it yields

  for some l in [1..o], (I-S (dreplace-subst &i tds_j mso_l) b)

  And since our result is the list of results of all these
  dreplace-subst invocations, this establishes our goal.
  (I.e., o = n and each (dreplace-subst &i tds_j mso_l) = s_l.)
  QED.
```

Now we can return to our proof of DMERGE-OK.

```
Lemma DMERGE-OK

  For any variable &i, descriptor td, and substitution list s,
  and denoting (dmerge-new-subst &i td s) = (s_1 .. s_n)

H1 (and (I-S s b)
H2      (I &i v b)
H3      (I td v b))
   => for some k in [1..n], (I-S s_k b)


Proof of Lemma DMERGE-OK

Consider two cases, as suggested by the function dmerge-new-subst.

Case 1: (assoc &i s) = nil

 In this case, (dmerge-new-subst &i td s) = ((dinsert-subst &i td s)).

 Then Lemma DINSERT-SUBST-OK gives us our result directly.

Case 2: (cdr (assoc &i s)) is not equal to nil.
 Let us say that (cdr (assoc &i s)) = tdsi and adopt the notation that
 (dunify-descriptors td tdsi s) = ((tdsi_1 . ssi_1) .. (tdsi_m . ssi_m))

 Expanding I-S in H1, we have a conjunct
 (I tdsi (cdr (assoc &i b)) b).  Expanding I in H2 gives us
 (equal (cdr (assoc &i b)) v).  We use DUNIFY-DESCRIPTORS-OK
 as in inductive hypothesis, instantiated with tda = td,
 tdb = tdsi, and substs = s.  This gives us

  (and (I-S s b) (I td v b) (I tdsi v b))
  =>
  for some j, (and (I-S ssi_j b) (I tdsi_j v b))

 We can use the conclusion of the inductive hypothesis, since H1 gives
 its first hypothesis, H3 gives the second, and
 (I tdsi (cdr (assoc &i b)) b) and the equality
 (equal (cdr (assoc &i b)) v) give us the third hypothesis.
 This gives us conclusion:

H4      for some j, (and (I-S ssi_j b) (I tdsi_j v b))

 By definition in this case,
  (dmerge-new-subst &i td s)         and hence (s_1 .. s_n)
   =
  (dmerge-dunified-forms-into-substs
    ((tdsi_1 . ssi_1) .. (tdsi_m . ssi_m)) &i s))

 So instantiate Lemma DMERGE-DUNIFIED-FORMS-INTO-SUBSTS-OK with
 s = s, b = b, tds_i = tdsi_i for all i, &i = &i, and
```

```
ssi = ssii for all i.  The antecedents for this lemma
are satisfied by H1, H2, and H4.  So where

 (dmerge-dunified-forms-into-substs
    ((tdsi1 . ssi1) .. (tdsim . ssim)) &i s)
   = (s1 .. sn)

this lemma gives us

 for some k in [1..n] (I-S sk b)

which is our goal.  QED.
```

B.7 The DUNIFY-DESCRIPTORS *REC Rules

This section contains the proofs of some, but not all of special-case rules employed by DUNIFY-DESCRIPTORS to handle the situations where both its DESCRIPTOR1 and DESCRIPTOR2 arguments are *REC descriptors. The rules are described and listed in Section 6.6.3. Each rule corresponds with a lemma which validates the application of the rule. The collected proofs of these lemmas are what satisfy Case 6.1 of the proof of DUNIFY-DESCRIPTORS-OK.

For selected rules, we state and prove the associated lemmas here. In many respects, the rules are very similar. Because of this similarity, the proofs of the corresponding lemmas are very similar in style and technique. Rather than burdening this document with the exhaustive proofs of all the rules, we present selected examples which are sufficient to show the proof techniques to provide convincing evidence that the other rules can be proven sound by similar arguments. The rules proved include some of the simplest and the most challenging, since the simple ones exemplify the proof style most succinctly, and the difficult ones bear the most scrutiny. The rule base is intended to be extendable, so other rules could be added, and their proofs would almost certainly follow the models given here.

Recall, as discussed in Section B.5, that each of these lemmas is included in the large nest of lemmas about mutually recursive functions in the unifier. The proof of this collection of lemmas is by a grand computational induction on the length of the unification computation. Any lemma in the nest may be used as an inductive assertion corresponding to the result of any subsidiary call in the algorithm. The *REC rules typically include recursive calls to DUNIFY-DESCRIPTORS-INTERFACE in cases where the arguments are known to be variable-free, or to DUNIFY-DESCRIPTORS for arguments which could contain variables. In the first case, we use DUNIFY-DESCRIPTORS-INTERFACE-OK as the inductive assertion, in the latter we use DUNIFY-DESCRIPTORS-OK. As with the other lemmas in the nest, the *REC rule lemmas are partial correctness results, since we have not give a termination proof.

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value. Also, we abbreviate INTERP-SUBSTS with "I-S". Furthermore, each rule is stated in terms of named *REC descriptors, with names like "foo", "bar", and "bim". In each case, after having introduced these descriptors, we refer to them simply by name. Finally, arbitrary component descriptors of the *RECs are typically named di, where i is some integer, and we use dij to signify the result returned by (DUNIFY-DESCRIPTORS-INTERFACE di dj).

B-A Rule Dunify-*Rec1

The rule is:

```
Rule Dunify-*Rec1
  (dunify-descriptors
    (*rec foo (*or $nil (*cons d1 (*recur foo))))
    (*rec bar (*or $nil (*cons d2 (*recur bar))))
    s)
   =
  (((*rec bim (*or $nil (*cons d12 (*recur bim)))) . s))
```

This is a very simple *REC rule. We do not have to worry about variables at all; they cannot occur in a replicating component of a *REC descriptor, so D1 and D2 are variable-free. (See the discussion of replicating components in Section 5.2.) Thus, substitutions are not an issue. The substitution s never gets extended, and it appears in the result intact.

Our rule for this case says that the definition of (DUNIFY-DESCRIPTORS foo bar s) is the single pair formed by s and a *REC descriptor which is of the same form as foo but with (DUNIFY-DESCRIPTORS-INTERFACE d1 d2) replacing d1. Thus, the lemma which corresponds to our rule is:

```
Lemma DUNIFY-*REC1-OK

 For any value v, substitution s, and binding b,
  (and
H1  (I-S s b)
H2  (I foo v b)
H3  (I bar v b)
H4  (equal (dunify-descriptors-interface d1 d2) d12) )
  =>
  (and (I-S s b) (I bim v b))
```

H1 - H3 are the standard hypotheses about the arguments to DUNIFY-DESCRIPTORS in lemmas about the unifier. H4 specifies the result returned from the recursive call to DUNIFY-DESCRIPTORS-INTERFACE, tying it to the use of the same term d12 in bim. The conclusion is the image of the conclusion to Lemma DUNIFY-DESCRIPTORS-OK for this case, stated in terms of the result constructed according to the rule.

```
Proof of Lemma DUNIFY-*REC1-OK

H1 establishes the first conjunct of the conclusion.  To establish
the second conjunct, we will induct on the structure of v.

 Case 1: v = nil

  By the definition of I, the second conjunct of the conclusion
  expands to

  (or (equal v nil) .. )

  establishing the goal trivially.

 Case 2: v2 = some non-NIL atom

  This contradicts H2, which by the definition of I expands to

  (or (equal v nil) (and (consp v) .. ) )

 Case 3: v is of the form = (cons vcar vcdr)

  Expand H2 and H3 and eliminate the null disjuncts, which are
```

```
  false under our case assumption, to give

H2' (and (consp v) (I d1 vcar b) (I foo vcdr b))
H3' (and (consp v) (I d2 vcar b) (I bar vcdr b))

  Using H4, instantiate Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK with
  tda = d1, tdb = d2, b = b, and v = vcar.  The second conjuncts of
  H2' and H3' relieve the hypotheses in the instantiated lemma,
  allowing us to use the conclusion,

H5 (I d12 vcar b)

  Now use Lemma DUNIFY-*REC1-OK as an inductive assertion, instantiating
  with foo = foo, bar = bar, b = b, d1 = d1, d2 = d2, and v = vcdr.
  H2', H3', and H4 relieve the hypotheses, giving us the conclusion

H6 (I bim vcdr b)

  Expanding the definition of I in the conclusion gives

  (or (equal v nil)
      (and (consp v) (I d12 vcar b) (I bim vcdr b)))

  Our case assumption establishes (consp v), H5 gives (I d12 vcar b),
  and H6 gives (I bim vcdr b), establishing the goal and completing
  the case analysis.  QED.
```

B-B Rule Dunify-*Rec3

```
Rule Dunify-*Rec3
 Where d2, d3, and d4 contain no variables and d3 is either a
 primitive descriptor or a disjunction of primitive descriptors,
 and denoting
 (dunify-descriptors &j (*rec bar (*or d3 (*cons d4 (*recur bar)))) s)
   = ((&j . s1) .. (&j . sn)),

 (dunify-descriptors (*rec foo (*or &j (*cons d2 (*recur foo))))
                     (*rec bar (*or d3 (*cons d4 (*recur bar))))
                     s)
  =
 (((*rec bim (*or &j (*cons d24 (*recur bim)))) . s1)
   ..
  ((*rec bim (*or &j (*cons d24 (*recur bim)))) . sn) )
```

Rule Dunify-*Rec3, stated as a theorem in terms of the interpreter, and assuming the constraints we placed on our descriptors above, is:

```
Lemma DUNIFY-*REC3-OK

 For any Lisp value v, substitution s, and binding b,
  (and
H1  (I-S s b)
H2  (I foo v b)
H3  (I bar v b)
H4  (equal (dunify-descriptors-interface d2 d4) d24)
H5  (equal (dunify-descriptors &j d3) ((&j . s1) .. (&j . sn))) )
  =>
  for some i in 1..n,
    (and (I-S si b) (I bim v b))


Proof of Lemma DUNIFY-*REC3-OK

 Expand I in H2 and H3 to give
```

H2' (or (I &j v b) (and (consp v) (I d2 (car v) b) (I foo (cdr v) b)))
H3' (or (I d3 v b) (and (consp v) (I d4 (car v) b) (I bar (cdr v) b)))

Expand I in the conclusion also, giving

for some i in 1..n,
(and (I-S s_i b)
(or (I &j v b)
(and (consp v) (I d24 (car v) b) (I bim (cdr v) b))))

We induct on the structure of v.

Case 1. v is atomic

Instantiate Lemma DUNIFY-DESCRIPTORS-OK with tda = &j, tdb = bar, substs = s, v = v, and b = b. The consp disjunct in H2 is clearly false, so the first disjunct must hold. This, H1, and H3 allow us to use the conclusion of the instantiated DUNIFY-DESCRIPTORS-OK, giving us

for some i,
(and (I-S s_i b) (I &j v b))

This establishes our conclusion.

Case 2. v is of the form (cons vcar vcdr)

Since d3 is either a primitive or a disjunction of primitives, it cannot represent a cons. Thus, the second disjunct of H3' must hold, giving us

H3'' (and (consp v) (I d4 (car v) b) (I bar (cdr v) b))

Now consider the cases suggested by H2'.

Case 2.1 (I &j v b)

Instantiate Lemma DUNIFY-DESCRIPTORS-OK with tda = &j, tdb = bar, substs = s, v = v, and b = b. H1, our Case 2.1 assumption, and H3 relieve the lemma's hypotheses, so we can use the conclusion,

for some i,
(and (I-S s_i b) (I &j v b))

This establishes our conclusion.

Case 2.2 (and (consp v) (I d2 (car v) b) (I foo (cdr v) b))

Instantiate Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK with tda = d2, tdb = d4, v = vcar, and b = b. The second conjunct of H3'' and the second conjunct of our Case 2.2 assumption relieve the lemma'a hypotheses, establishing the conclusion:

H6 (I d24 (car v) b)

Now instantiate Lemma DUNIFY-DESCRIPTORS-OK with td1 = foo, td2 = bar, v = (cdr v), b = b, and substs = s. H1, the third conjunct of the case assumption, and the third conjunct of H3'' relieve the hypotheses, giving us the conclusion

H7 for some i,
(and (I-S s_i b) (I bim (cdr v) b))

Choosing the i for which H7 holds, the first conjunct of H7 establishes the first conjunct of the conclusion. (consp v) from the case assumption, H6, and the second conjunct of H7 establish the second disjunct of the second conjunct of the

```
   goal, thus establishing the goal for this case and completing
   the case analysis.  QED.
```

B-C Rule Dunify-*Rec5

```
Rule Dunify-*Rec5

With no variables appearing in either descriptor, and denoting
  (dunify-descriptors-interface
     d1 (*cons d4 (*rec bar (*or d3 (*cons d4 (*recur bar))))))
    = d14bar, and
  (dunify-descriptors-interface
     d3 (*cons d2 (*rec foo (*or d1 (*cons d2 (*recur foo))))))
    = d32foo

(dunify-descriptors
  (*rec foo (*or d1 (*cons d2 (*recur foo))))
  (*rec bar (*or d3 (*cons d4 (*recur bar))))
  s)
 =
(((*rec bim (*or d13 d14bar d32foo (*cons d24 (*recur bim)))) . s))
```

Rule Dunify-*Rec5 is very general, in that it does not require d1 or d3 to be cons-free. Since all the non-replicating terms in a *REC body can be gathered into an *OR, d1 and d3 represent any such terms, with the only other restriction being that they are variable-free. The lemma corresponding to the rule is:

```
Lemma DUNIFY-*REC5-OK

 For any Lisp value v, binding b, and substitution s,
  (and
H1  (I-S s b)
H2  (I foo v b)
H3  (I bar v b)
H4  (equal (dunify-descriptors-interface d1 d3) d13)
H5  (equal (dunify-descriptors-interface d1 (*cons d4 (*recur bar)))
           d14bar)
H6  (equal (dunify-descriptors-interface d3 (*cons d2 (*recur foo)))
           d32foo)
H7  (equal (dunify-descriptors-interface d2 d4) d24) )
  =>
  (and (I-S s b) (I bim v b))


Proof of Lemma DUNIFY-*REC5-OK

 H1 establishes the first conjunct of the conclusion.  The second
 conjunct expands to:

C1 (or (I d13 v b)
C2     (I d14bar v b)
C3     (I d32foo v b)
C4     (and (consp v) (I d24 (car v) b) (I bim (cdr v) b)))

 Expanding the definition of I in H2 and H3 yields

H2' (or (I d1 v b)
        (and (consp v) (I d2 (car v) b) (I foo (cdr v) b)))
H3' (or (I d3 v b)
        (and (consp v) (I d4 (car v) b) (I bar (cdr v) b)))

 We will induct on the structure of v.

 Case 1.  v is atomic
```

```
  This establishes the first disjunct of H2’ and H3’, since the
  consp tests will fail.  As suggested by H4, instantiate Lemma
  DUNIFY-DESCRIPTORS-INTERFACE-OK with tda = d1, tdb = d3, v = v,
  and b = b.  H2’ and H3’ relieve its hypotheses, giving us the
  conclusion,

  (I d13 v b)

  which equals C1, establishing the goal

 Case 2.  v is of the form (cons vcar vcdr)

  We will consider pairwise the cases suggested by H2’ and H3’.
  In each case, we will instantiate Lemma
  DUNIFY-DESCRIPTORS-INTERFACE-OK as suggested by one or more of
  H4 - H7.

  Case 2.1  (and (I d1 v b) (I d3 v b))

   Proceed exactly as in Case 1, using the Case 2.1 assumption to
   relieve the hypotheses of DUNIFY-DESCRIPTORS-INTERFACE-OK.

  Case 2.2  (and (I d1 v b)
                 (and (consp v) (I d4 (car v) b) (I bar (cdr v) b)))

   Instantiate DUNIFY-DESCRIPTORS-INTERFACE-OK according to H5,
   with tda = d1, tdb = (*cons d4 bar), v = v, and b = b.  Our
   case assumption establishes the antecedents, giving us the
   conclusion, (I d14bar v b), which is equal to C2, establishing
   the goal.

  Case 2.3  (and (and (consp v) (I d2 (car v) b) (I foo (cdr v) b))
                 (I d3 v b))

   Instantiate DUNIFY-DESCRIPTORS-INTERFACE-OK according to H6,
   with tda = d3, tdb = (*cons d2 foo), v = v, and b = b.  Our
   Case 2.3 assumption establishes the first antecedent, and
   expanding the definition of I in the second antecedent yields
   a form equal to the first disjunct of our case assumption.
   Thus we establish the conclusion, (I d32foo v b), which is
   equal to C3, establishing the goal.

  Case 2.4  (and (and (consp v) (I d2 (car v) b) (I foo (cdr v) b))
                 (and (consp v) (I d4 (car v) b) (I bar (cdr v) b)))

   Instantiate DUNIFY-DESCRIPTORS-INTERFACE-OK with tda = d2,
   tdb = d4, v = vcar, and b = b.  The antecedents are relieved
   by the second conjunct of each conjunct of the case assumption.
   This gives us

H8  (I d24 (car v) b)

   Now use our Lemma DUNIFY-*REC5-OK as an inductive assertion,
   instantiating with v = (cdr v), s = s, and b = b.  H1 and
   H4 - H7 are identical to their respective antecedents in the
   inductive assertion, and the conjuncts of our case assumption
   establish H2 and H3 respectively in the inductive assertion.
   This gives us the conclusion

H9  (I bim (cdr v) b)

   The (consp v) from the case assumption, H8, and H9 combine
   to establish C4, establishing the goal and completing the
   case analysis.  QED.
```

B-D Rule Dunify-*Rec11

```
Rule Dunify-*Rec11

 Where d2, d3, and d4 contain no variables, and adopting the notation

 (dunify-descriptors &j (*rec bar (*or d3 (*cons d4 (*recur bar)))) s)
   = ((&j . s1) .. (&j . sn))
 (dunify-descriptors
    d3 (*cons d2 (*rec foo (*or &j (*cons d2 (*recur foo))))) s)
   = ((d32foo1 . sd31) .. (d32foom . sd3m)),

 (dunify-descriptors (*rec foo (*or &j (*cons d2 (*recur foo))))
                     (*rec bar (*or d3 (*cons d4 (*recur bar))))
                     s)
  =
 (((*rec bim (*or &j (*cons d24 (*recur bim)))) . s1)
   ..
  ((*rec bim (*or &j (*cons d24 (*recur bim)))) . sn)
  ((*rec bam1 (*or d32foo1 (*cons d24 (*recur bam1)))) . sd31)
   ..
  ((*rec bamm (*or d32foom (*cons d24 (*recur bamm)))) . sd3m)
 )
```

For further notational convenience, let

```
  "bami" signify (*rec bami (*or d32fooi (*cons d24 (*recur bami))))
```

Rule Dunify-*Rec11, stated as a theorem in terms of the interpreter, and assuming the constraints we placed on our descriptors above, is:

```
Lemma DUNIFY-*REC11-OK

 For any Lisp value v, binding b covering foo and bar, and
 substitution s,
  (and
H1  (I-S s b)
H2  (I foo v b)
H3  (I bar v b)
H4  (equal (dunify-descriptors-interface d2 d4) d24)
H5  (equal (dunify-descriptors &j bar s)
           ((&j . s1) .. (&j . sn)))
H6  (equal (dunify-descriptors d3 (*cons d2 foo) s)
           ((d32foo1 . sd31) .. (d32foom . sd3m))) )
  =>
  (or
C1   for some i in 1..n,
       (and (I-S si b)
            (I (*rec bim (*or &j (*cons d24 (*recur bim)))) v b))
C2   for some k in 1..m
       (and (I-S sd3k b)
            (I (*rec bamk (*or d32fook (*cons d24 (*recur bamk)))) v b)))


Proof of Lemma DUNIFY-*REC11-OK

 First we expand I in H2 and H3 to get

H2' (or (I &j v b)
        (and (consp v) (I d2 (car v) b) (I foo (cdr v) b)))
H3' (or (I d3 v b)
        (and (consp v) (I d4 (car v) b) (I bar (cdr v) b)))
```

```
  C1 expands by definition of I to

C1' for some i in 1..n,
     (and (I-S si b)
          (or (I &j v b)
              (and (consp v) (I d24 (car v) b) (I bim (cdr v) b))))

  C2 expands by definition of I to

C2' for some k in 1..m
     (and (I-S sd3k b)
          (or (I d32fook v b)
              (and (consp v) (I d24 (car v) b) (I bamk (cdr v) b))))

 We will induct on the structure of v.

 Case 1.  v is an atom

  This rules out the consp disjuncts in H2' and H3'.  Instantiate
  Lemma DUNIFY-DESCRIPTORS-OK as suggested by H5, with tda = &j,
  tdb = bar, v = v, substs = s, and b = b.  H1, H2', and H3 relieve
  its hypotheses, establishing the conclusion:

   for some i,
     (and (I-S si b) (I &j v b))

  This establishes both conjuncts of C1'.

 Case 2.  v is of the form (cons vcar vcdr)

  Consider pairwise the cases suggested by H2' and H3'.

  Case 2.1  (I &j v b)

   Proceed as with Case 1.

  Case 2.2  (and (and (consp v) (I d2 (car v) b) (I foo (cdr v) b))
                 (I d3 v b))

   Instantiate Lemma DUNIFY-DESCRIPTORS-OK as suggested by H6, with
   tda = d3, tdb = (*cons d2 foo), v = v, substs = s, and b = b.  H1
   and the second conjunct of our case assumption are equal to the
   first two antecedents, and the expansion of the third antecedent
   is equal to the first conjunct of the case assumption.  So we have
   established the conclusion:

    for some k in 1..m,
      (and (I-S sd3k b) (I d32fook v b))

   This establishes both conjuncts of C2'.

  Case 2.3  (and (and (consp v) (I d2 (car v) b) (I foo (cdr v) b))
                 (and (consp v) (I d4 (car v) b) (I bar (cdr v) b)))

   First instantiate Lemma DUNIFY-DESCRIPTORS-INTERFACE-OK as
   suggested by H4, with tda = d2, tdb = d4, v = (car v), and
   b = b.  Thus, its first hypothesis is (I d2 (car v) b), and
   the second is (I d4 (car v) b).  These are both relieved by
   the case assumption, so we have established the conclusion:

H7  (I d24 (car v) b)

   Now use our Lemma DUNIFY-*REC11-OK as an inductive assertion,
   instantiated with v = (cdr v), s = s, and b = b.  H1 and H4 - H6
   are identical with the hypotheses of the inductive assertion,
   and H2 and H3 are established by the last conjunct of each
   conjunct of our case assumption.  We do not need C1 here, but
   C2 gives us:
```

```
H8  for some k in 1..m
     (and (I-S sd3k b) (I bamk (cdr v) b))

   This establishes the first conjunct of our goal.  The (consp v)
   term from our case assumption, H7, and H8 combine to establish
   the second disjunct of the second conjunct.  Thus, we have
   established the goal.

  This completes the case analysis.  QED.
```

B.6 Descriptor Canonicalization

The canonicalization rules are stated in the form "td1 ==> td2", where td1 is a generic representation of descriptors eligible for the canonicalization, and td2 is the representation of the form into which those descriptors are canonicalized. By "==>" we mean only that

```
(PCANONICALIZE-DESCRIPTOR TD1) = TD2
```

Although the canonicalization is directed, each rule happens to be an equality, in the sense that td1 and td2 represent the same set of values under any interpretation by INTERP-SIMPLE. All that is required for soundness of each rule "td1 ==> td2" is proof of an associated lemma of the form:

```
For all value v and binding b,
 (I td1 v b)
 =>
 (I td2 v b)
```

Since it would be tedious to restate each rule in this form, we simply provide the rule notation, but the proofs follow the INTERP-SIMPLE model. Each rule is annotated with the name of the function in which it is implemented.

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
Rule 1:
  (*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL
       $NON-T-NIL-SYMBOL $STRING $T (*CONS *UNIVERSAL *UNIVERSAL))
  ==> *UNIVERSAL
From function PCONSOLIDATE-UNIVERSAL-DESCRIPTORS

Proof: (I *universal v b) = T for any v and b.
```

The reverse implication is also straightforward. It requires the knowledge that the only data items in our world are characters, integers, NIL, non-integer rational numbers, symbols other than T or NIL, strings, T, and conses whose atomic leaves are among the aforementioned. This, of course, is a basic and valid assumption in the problem domain. Thus, proof is simply to show by cases in the data space that for some disjunct of the *OR descriptor, (I <disjunct> v b) holds for any v and b. I.e., if the object v were a character, then for any b,

```
(I (*or $character $integer $nil $non-integer-rational
        $non-t-nil-symbol $string $t (*cons *universal *universal))
   v
   b)
 = T
```

since this call to I would expand to a disjunction of calls, one of which, (I $character v b), would hold. The same kind of argument would apply to all other types of data in the world.

```
Rule 2:
  (*rec foo (... (*recur foo) ...))
  ==> (... (*rec foo (... (*recur foo) ...)) ...)
from function OPEN-REC-DESCRIPTOR-ABSOLUTE
```

Comment: This is a little strange, because the very function which performs this canonicalization is part of the definition of the semantics. When the interpreter encounters a *REC form, it calls itself recursively with the argument (OPEN-REC-DESCRIPTOR-ABSOLUTE form). Thus, the validity of this canonicalization is trivially established.

```
Proof:
 (I (*rec foo (... (*recur foo) ...)) v b)
  =   by expanding the definition of I
 (I (... (*rec foo (... (*recur foo) ...)) ...) v b)

Rule 3:
  (... (*rec foo (... (*recur foo) ...)) ...)
  ==> (*rec foo (... (*recur foo) ...))
From function PFOLD-RECS-IF-POSSIBLE
```

Comment: By this notation we indicate that the forms represented by the (... ...) outside the *REC descriptor are identical to those represented by the (... ...) inside the *REC but outside the (*RECUR FOO). Moreover, we intend that there may be more than one (*RECUR FOO) inside the *REC, and in this case, the larger form would have matching occurrences of the *REC.

```
Proof:
This follows directly from the proof for Rule 2.  If we apply the
interpreter to the right hand side and let it run one step, we
reduce the right hand side to the left hand side.

Rule 4:
 (*or .. (*or d1 .. d2) ..) ==> (*or .. d1 .. d2 ..)
From function PCANONICALIZE-OR-DESCRIPTOR

Proof:
 (I (*or .. (*or d1 .. d2) ..))
   =  (by definition of I)
 (or ..
     (I (*or d1 .. d2) v b)
     .. )
   = (by definition of I)
 (or (or (I d1 v b)
         ..
         (I d2 v b))
     .. )
   = (by associativity of OR)
 (or ..
     (I d1 v b)
     ..
     (I d2 v b)
     .. )
and
 (I (*or .. d1 .. d2 ..) v b)
   = (by definition of I)
 (or ..
     (I d1 v b)
     ..
     (I d2 v b)
     .. )
Thus, the two sides are equivalent.
```

```
Rule 5:
 (*or .. d1 .. d1 ..) ==> (*or .. d1 .. ..)
From function PCANONICALIZE-OR-DESCRIPTOR
```

Comment: This rule characterizes any multiple occurrence of a descriptor as arguments of an *or.

```
Proof:
 (I (*or .. d1 .. d1 ..) v b)
   = (by definition of I)
 (or ..
     (I d1 v b)
     ..
     (I d1 v b)
     .. )
   = (by tautology)
 (or ..
     (I d1 v b)
     ..
     ..)
   = (by definition of I)
 (I (*or .. d1 .. ..) v b)
```

```
Rule 6.
 (*or .. *empty .. ) ==> (*or .. .. )
From function PCANONICALIZE-OR-DESCRIPTOR
```

Comment: This rule characterizes any occurrence of *EMPTY as an argument of an *OR. The proof is analogous in any such scenario.

```
Proof:
 (I  (*or .. *empty .. ) v b)
   = (by definition of I)
 (or ..
     (I *empty v b)
     .. )
   = (by definition of I in the *empty case)
 (or ..
     nil
     ..)
   = (by disjunctive syllogism)
 (or ..
     .. )
   = (by definition of I)
 (I (*or .. ..) v b)

Rule 7.
 Where d does not contain a (*recur foo),
  (*or d (*rec foo (*or d .. (*recur foo))))
  ==> (*rec foo (*or d .. (*recur foo)))
From function PREMOVE-RECURSIVE-EXPANSION-DUPLICATES

Proof:
 (*or d (*rec foo (*or d .. (*recur foo))))
  = (by Rule 2)
 (*or d (*or d .. (*rec foo (*or d .. (*recur foo)))))
  = (by Rule 5)
 (*or d .. (*rec foo (*or d .. (*recur foo))))
  = (by Rule 3)
 (*rec foo (*or d .. (*recur foo)))

Rule 8:
 (*or (*cons d1 d2) (*cons d3 d2)) ==> (*cons (*or d1 d3) d2)
From PMERGE-OR-DESCRIPTOR-CONSES

Proof:
 (I (*or (*cons d1 d2) (*cons d3 d2)) v b)
   = (by definition of I)
 (or (I (*cons d1 d2) v b)
```

```
    (I (*cons d3 d2) v b))
  = (by definition of I)
 (or (and (consp v)
          (I d1 (car v) b)
          (I d2 (cdr v) b))
     (and (consp v)
          (I d3 (car v) b)
          (I d2 (cdr v) b)))
  = (by distributive property)
 (and (or (and (consp v)
               (I d1 (car v) b)
               (I d2 (cdr v) b))
          (consp v))
      (or (and (consp v)
               (I d1 (car v) b)
               (I d2 (cdr v) b))
          (I d3 (car v) b))
      (or (and (consp v)
               (I d1 (car v) b)
               (I d2 (cdr v) b))
          (I d2 (cdr v) b)))
  = (by simplification)
 (and (consp v)
      (or (and (consp v)
               (I d1 (car v) b)
               (I d2 (cdr v) b))
          (I d3 (car v) b))
      (I d2 (cdr v) b))
  = (by distributive property)
 (and (consp v)
      (and (or (consp v)
               (I d3 (car v) b))
           (or (I d1 (car v) b)
               (I d3 (car v) b))
           (or (I d2 (cdr v) b)
               (I d3 (car v) b)))
      (I d2 (cdr v) b))
  = (by simplification)
 (and (consp v)
      (or (consp v)
          (I d3 (car v) b))
      (or (I d1 (car v) b)
          (I d3 (car v) b))
      (or (I d2 (cdr v) b)
          (I d3 (car v) b))
      (I d2 (cdr v) b))
  = (by distributive property)
 (and (consp v)
      (consp v)
      (or (I d1 (car v) b)
          (I d3 (car v) b))
      (I d2 (cdr v) b)
      (I d2 (cdr v) b))
  = (by tautology)
 (and (consp v)
      (or (I d1 (car v) b)
          (I d3 (car v) b))
      (I d2 (cdr v) b))
  = (by definition of i)
 (I (*cons (*or d1 d3) d2))
```

```
Rule 9:
 (*or (*cons d1 d2) (*cons d1 d3)) ==> (*cons d1 (*or d2 d3))
From PMERGE-OR-DESCRIPTOR-CONSES

Proof:
 As with Rule 8.

Rule 10:
```

```
 (*or .. *universal ..) ==> *universal
from PCANONICALIZE-OR-DESCRIPTOR

Proof:
 (I *universal v b) = T, so the proof is trivial.

Rule 11:
 (*or d) ==> d
I.e., the *OR encapsulates only one disjunct.
From PCANONICALIZE-OR-DESCRIPTOR

Proof:
 (I (*or d) v b)
   = (by definition of INTERP-SIMPLE-OR)
 (or (I d v b)
     nil)
   = (by definition of OR)
 (I d v b)

Rule 12:
 (*or) ==> *empty
Here, the *OR encapsulates no disjuncts.
From PCANONICALIZE-OR-DESCRIPTOR

Proof:
 (I (*or) v b)
   = (by definition of INTERP-SIMPLE-OR)
 NIL

 (I *empty v b)
   = (by definition of INTERP-SIMPLE-OR)
 NIL

Rule 13:
 (*OR .. d1 .. d2 ..) ==> (*OR .. d2 .. d1 ..)
From PCANONICALIZE-OR-DESCRIPTOR
```

Comment: This rule allows the placement of *OR disjuncts into a canonical order, determined by the lexical ordering function DESCRIPTOR-ORDER.

```
Proof:
 (I (*or .. d1 .. d2 ..) v b)
   = (by definition of INTERP-SIMPLE-OR)
 (or ..
     (I d1 v b)
     ..
     (I d2 v b)
     ..)
   = (by commutativity of OR)
 (or ..
     (I d2 v b)
     ..
     (I d1 v b)
     ..)
   = (by definition of INTERP-SIMPLE-OR)
 (I (*or .. d2  .. d1 ..) v b)

Rule 14:
 (*cons *empty d) ==> *empty
from PCANONICALIZE-CONS-DESCRIPTOR

Proof:
 (I (*cons *empty d) v b)
   = (by definition of I)
 (and (consp v)
      (I *empty v b)
      (I d v b))
   = (by definition of I)
```

```
 (and (consp v)
      nil
      (I d v b))
   = (by simplification)
 nil
   = (by definition of I)
 (I *empty v b)
```

Rule 15:

```
 (*cons d *empty) ==> *empty
From PCANONICALIZE-CONS-DESCRIPTOR
```

Proof: As with Rule 14.

Rule 16:

```
 (*rec foo (*or td_foo (*recur foo))) ==> (*rec bar td_bar)
where td_bar = (subst (*recur bar) (*recur foo) td_foo)
and bar is a new *rec name and hence (*recur bar) does not occur in
td_foo.
```

From function CANONICALIZE-REC-FOR-DUNIFY.

Comment: The critical feature of the left hand side is that the (*RECUR FOO) form is a top-level disjunct in the body of the *REC. The idea motivating this canonicalization is that since the (*RECUR FOO) adds no information not already present in the rest of the descriptor, it can be eliminated. The *REC must be renamed, since all occurrences of *REC forms with the same label in any given context must be identical.

Proof:

We wish to prove that for any value v and binding b,

```
 (and (I (*rec foo (*or td_foo (*recur foo))) v b)
      (equal td_bar (subst (*recur bar) (*recur foo) td_foo))
      for any x,
       (equal (subst x (*recur bar) td_foo) td_foo)
 =>
 (I (*rec bar td_bar) v b)
```

The third hypothesis is the image of the requirement that bar is a fresh *rec label.

The "subst" function here uses the "equal" test.

We will do this proof by proving a slightly more general lemma:

For any descriptor td, Lisp value v and binding b,

```
 (and
H1 (I td v b)
H2 (equal td_bar (subst (*recur bar) (*recur foo) td_foo))
H3 for any x, (equal (subst x (*recur bar) td_foo) td_foo)
 =>
 (I (subst (*rec bar td1_bar)
           (*rec foo (*or td1_foo (*recur foo)))
           td)
    v
    b)
```

Our main lemma is just an instantiation of this lemma, with td = (*rec foo (*or td$_{foo}$ (*recur foo))).

We will proceed by computational induction on the length of the computation by I. In a strange sense, this is a partial correctness proof, since a certain ordering of the computation of
 (I (*rec foo (*or td$_{foo}$ (*recur foo))) v b)
is obviously non-terminating, i.e., the ordering which dives infinitely into the second disjunct.

Case 1. td = *empty
This falsifies H1.

Case 2. td is a primitive descriptor, a type variable, or *universal
The substitution in the conclusion is a no-op, so H1 establishes the goal.

Case 3. td is of the form (*cons tdcar tdcdr)
By definition of subst, the conclusion becomes

```
(I (*cons (subst (*rec bar td1_bar)
                 (*rec foo (*or td1_foo (*recur foo)))
                 tdcar)
          (subst (*rec bar td1_bar)
                 (*rec foo (*or td1_foo (*recur foo)))
                 tdcdr))
   v b)
 =   by the definition of I
(and (consp v)
     (I (subst (*rec bar td1_bar)
               (*rec foo (*or td1_foo (*recur foo)))
               tdcar)
        (car v) b)
     (I (subst (*rec bar td1_bar)
               (*rec foo (*or td1_foo (*recur foo)))
               tdcar)
        (cdr v) b))
```

H1 expands, by the definition of I, to

```
(and (consp v) (I tdcar (car v) b) (I tdcdr (cdr v) b))
```

Thus, the first conjunct of the goal is established. The other conjuncts are established by using the lemma as an inductive assertion, instantiated first with td = tdcar, v = (car v), and b = b, and again with td = tdcdr, v = (cdr v), and b = b. This establishes the goal.

Case 4. td is of the form (*or td_1 .. td_n)

By the definition of I, H1 becomes

```
(or (I td_1 v b) .. (I td_n v b))
```

and the conclusion becomes

```
(or (I (subst (*rec bar td1_bar)
              (*rec foo (*or td1_foo (*recur foo)))
              td_1)
       v b)
    ..
    (I (subst (*rec bar td1_bar)
              (*rec foo (*or td1_foo (*recur foo)))
              td_n)
       v b))
```

Choose any td_i for which H1 holds and use the lemma as an inductive assertion, instantiating with td = td_i, v = v, and b = b. This establishes one disjunct of the goal, which suffices.

Case 5. td is a *rec form other than foo

By the definition of I, H1 becomes

```
(I (open-rec-descriptor-absolute td) v b)
```

Since the form in the conclusion is also a *rec descriptor, the conclusion becomes

(I (open-rec-descriptor-absolute
    (subst (*rec bar td1$_{bar}$)
        (*rec foo (*or td1$_{foo}$ (*recur foo)))
        td)))
  v b)

Since (open-rec-descriptor-absolute td) replaces no instances of (*rec foo (*or td1$_{foo}$ (*recur foo))), this form equals

(I (subst (*rec bar td1$_{bar}$)
        (*rec foo (*or td1$_{foo}$ (*recur foo)))
        (open-rec-descriptor-absolute td))
  v b)

Use the lemma as an inductive assertion with v = v, b = b, and td = (open-rec-descriptor-absolute td).

Case 6. td = (*rec foo (*or td1$_{foo}$ (*recur foo)))

Notice that the conclusion simplifies to

(I (*rec bar td1$_{bar}$) v b)

By the definitions of I and open-rec-descriptor-absolute, H1 becomes

(or (I (subst (*rec foo (*or td1$_{foo}$ (*recur foo)))
            (*recur foo)
            td1$_{foo}$)
      v b)
  (I (*rec foo (*or td1$_{foo}$ (*recur foo))) v b))

Consider each case separately.

Case 6.1 (I (*rec foo (*or td1$_{foo}$ (*recur foo))) v b)

Use the lemma as an inductive assertion with v = v, b = b, and td = (*rec foo (*or td1$_{foo}$ (*recur foo))). The conclusion simplifies to our goal.

Case 6.2 (I (subst (*rec foo (*or td1$_{foo}$ (*recur foo)))
            (*recur foo)
            td1$_{foo}$)
      v b)

Use the lemma as an inductive hypothesis, with v = v, b = b, and td = (subst (*rec foo (*or td1$_{foo}$ (*recur foo)))
            (*recur foo)
            td1$_{foo}$)

Since our case assumption establishes its antecedent, this gives us

(I (subst (*rec bar td1$_{bar}$)
        (*rec foo (*or td1$_{foo}$ (*recur foo)))
        (subst (*rec foo (*or td1$_{foo}$ (*recur foo)))
            (*recur foo)
            td1$_{foo}$))
  v b)

Notice that td1$_{foo}$ cannot lexically contain (*rec foo (*or td1$_{foo}$ (*recur foo))), since that would make

```
   it a lexically infinite term.  So, the outer subst can replace
   only those instances of (*rec foo (*or td1_foo (*recur foo)))
   which were injected by the inner subst.  So the term above is
   equal to

   (I (subst (*rec bar td1_bar)
             (*recur foo)
             td1_foo)
      v b)

   Now expand the definition of I in our conclusion.

   (I (open-rec-descriptor-absolute (*rec bar td1_bar)) v b)

   By the definition of open-rec-descriptor-absolute, this equals

   (I (subst (*rec bar td1_bar) (*recur bar) td1_bar)

   Substitute, using the equality in H2, to get

   (I (subst (*rec bar td1_bar)
             (*recur bar)
             (subst (*recur bar) (*recur foo) td_foo))
      v b)

   Substitute, using the equality in H3, to get

   (I (subst (*rec bar td1_bar)
             (*recur foo)
             td_foo)
      v b)

   This is what we established above.
  QED.


Rule 17:
Where within (.. (*RECUR FOO) ..) there is no *OR enclosing the
(*RECUR FOO), but some *CONS enclosing it,

 (*REC FOO (.. (*RECUR FOO) ..)) ==> *EMPTY
From function PCANONICALIZE-REC-DESCRIPTOR
```

Comment: With this notation, we indicate a *REC structure with no terminating disjunct. Thus, it can represent only infinite objects, and since these are not allowed in our subset, we reduce the form to *EMPTY.

```
Proof:
Let us perform the proof on a canonical example, so we avoid getting
tangled up in notation.  We will prove

 (*REC FOO (*CONS d (*RECUR FOO))) = (*REC FOO (*CONS d *EMPTY))

We will prove this by induction on the depth (in the CDR) of the data
object v.

First, clearly (I (*REC FOO (*CONS d *EMPTY)) v b) is always NIL,
since if v is atomic, it is not a CONS, and if it is a CONS, the
expansion of (I (*REC FOO (*CONS d *EMPTY)) v b),
i.e.,

  (AND (CONSP V)
       (I d (CAR V) B)
       (I *EMPTY (CDR V) B))

is always NIL, since (I *EMPTY (CDR V) B) = NIL.
```

```
So our task is to prove for any v and b,

 (I (*REC FOO (*CONS d (*RECUR FOO))) v b) = NIL

For the sake of illustrating our induction schema, consider

(DEFUN LEN (X)
  (IF (ATOM X) 0 (+ 1 (LEN (CDR X)))))

Given a CONS structure, LEN returns an integer which is its
length, i.e., the number of successive calls to CDR required
to reach an atomic value.  We induct on the length of v.

Base Case 1: v is atomic ((len v) = 0)
 (I (*rec foo (*cons d (*recur foo))) v b) = nil
 since v is not a cons.

Inductive Case: Assume the lemma is true for (len v) = n, and
  prove it for (len v) = n+1

  So v = (cons vcar vcdr), where (len v) = n+1 and (len vcdr) = n
  The inductive hypothesis is,
   For all v' = (cons vcar' vcdr'), where (len vcdr') = n,
     (I (*rec foo (*cons d (*recur foo))) v' b) = nil

  (I (*rec foo (*cons d (*recur foo))) (cons vcar vcdr) b)
   =
  (and (consp (cons vcar vcdr))
       (I d vcar b)
       (I (*rec foo (*cons d (*recur foo))) vcdr b))
   =   by the inductive hypothesis, since (len vcdr) = n,
  (and (consp (cons vcar vcdr))
       (I d vcar b)
       nil)
   = nil

QED

Rule 18:
  Where (*dlist d1 .. dfoo .. dn) and (*dlist d1 .. dbar .. dn)
  are identical except that they differ in one position, dfoo vs. dbar:
    (*or ... (*dlist d1 .. dfoo .. dn) ... (*dlist d1 .. dbar .. dn) ...)
  = (*or ... (*dlist d1 .. (*or dfoo dbar) .. dn) ...)
From function PCANONICALIZE-OR-DESCRIPTOR
```

Comment: An *OR of *DLISTS is not a conventional descriptor. But it is formed on exit from DUNIFY-DESCRIPTORS-INTERFACE when we call it with *DLIST descriptors, whenever multiple results are returned from its call to DUNIFY-DESCRIPTORS. Thus, rather than there being a semantic notion of an *OR of *DLISTs, this packaging is just a protocol with any function which calls DUNIFY-DESCRIPTORS-INTERFACE with *DLIST arguments, allowing DUNIFY-DESCRIPTORS-INTERFACE to behave polymorphically. The caller has the obligation to unpack the *OR. and to treat the list of results as a disjunction of possibilities. Hence, this canonicalization allows a merging of results.

```
Proof:
  We will let v be the list of values (v1 .. vfoobar .. vn).

  (or (I (d1 .. dfoo .. dn) (v1 .. vfoobar .. vn) b)
      (I (d1 .. dbar .. dn) (v1 .. vfoobar .. vn) b))
  = (or (and (I d1 v1 b)
             ..
             (I dfoo vfoobar b)
             ..
             (I dn vn b))
        (and (I d1 v1 b)
             ..
             (I dbar vfoobar b)
```

```
             ..
             (I dn vn b))
  = distributing OR over AND and reducing all instances of
       (OR x x) to x
    (and (I d1 v1 b)
         ..
         (or (I dfoo vfoobar b)
             (I dbar vfoobar b))
         ..
         (I dn vn b))
  = by definition of I
    (and (I d1 v1 b)
         ..
         (I (*or dfoo dbar) vfoobar b)
         ..
         (I dn vn b))
  = by definition of I
    (I (d1 .. (*or dfoo dbar) .. dn)
       (v1 .. vfoobar .. vn)
       b)
 QED
```

B.7 The Containment Algorithm

The lemmas proved in this section are CONTAINED-IN-OK and ICONTAINED-IN-OK and some lemmas associated with their *REC rules, ICONTAINED-IN-EQUAL-TDS, which is subsidiary to ICONTAINED-IN-OK, and UNIVERSALIZE-SINGLETON-VARS-1-OK, which is subsidiary to CONTAINED-IN-INTERFACE-OK.

For a full discussion of the containment algorithm, see Section 6.8.

B.8 Proof of Lemma CONTAINED-IN-OK

CONTAINED-IN-OK is the soundness lemma for the CONTAINED-IN algorithm, which is employed by CONTAINED-IN-INTERFACE whenever neither of the descriptors contain type variables. The main part of the proof is in the section which follows immediately. One case in this proof is deferred in this section. This is the case where both DESCRIPTOR1 and DESCRIPTOR2 are *REC descriptors. The proof approach for this case is in the second section to follow.

For a description of CONTAINED-IN, see Section 6.8.3.

B-E CONTAINED-IN-OK, Top Level Proof

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

```
Lemma CONTAINED-IN-OK

 For any descriptors td1 and td2, Lisp value v, and binding b1,

  (and
H1  (null (gather-variables-in-descriptor td1))
H2  (null (gather-variables-in-descriptor td2))
H3  (contained-in td1 td2)
H4  (I td1 v b1))
  =>
```

```
  For some b2, (I td2 v b2)

Comment: The b2 used here is irrelevant, since the descriptors are
variable-free.
```

The proof of soundness of CONTAINED-IN is actually the "simultaneous" proof of the conjunction of a collection of lemmas whose proofs rely on one another. The nest includes Lemma CONTAINED-IN-OK and the correctness lemmas for all the *REC containment rules. The proof is by computational induction on the length of the computation by CONTAINED-IN. This allows the assumption of any appropriate lemma within the nest to characterize the result of any recursive call in the computation subsidiary to the top level call.

Recall that CONTAINED-IN also takes a third argument, TERM-RECS, which is employed solely as a mechanism for terminating the computation in certain cases. As such, it has no effect on the soundness argument, which does not rely on termination, and to avoid clutter in the lemmas and proofs, we simply omit mention of it. One could think of it as being a universally quantified variable in the lemma with no particular demands placed on it.

```
Proof of Lemma CONTAINED-IN-OK

Our case analysis takes each case in sequence, so each one
assumes the negation of all previous cases.

Since td1 and td2 are variable-free and the algorithm introduces no
new variables, b1 and b2 are irrelevant to the evaluation of
(I td1 v b1) and (I td2 v b2).  We fix b2 to equal b1 and simply
refer to b to eliminate any confusion.  Now proceed by cases.

 Case 1. td1 = td2
  Trivial.  For any v and b, (I td1 v b) => (I td2 v b)

 Case 2. td1 = *empty
  By definition, (contained-in td1 td2) = t.
  Trivial.  For any v and b, (I td1 v b) = nil, falsifying H4.

 Case 3. td2 = *empty
  Trivial.  By definition, (contained-in td1 td2) = nil, falsifying H3.

 Case 4. td1 = *universal
   By definition, (contained-in td1 td2) = nil, falsifying H3.

 Case 5. td2 = *universal
   By definition, (contained-in td1 td2) = t.
   For all v and b, (I *universal v b) = t, establishing the
   conclusion.

 Case 6. td1 = (*or td1_1 .. td1_n)
   By definition,
   (contained-in td1 td2) =
     all i in [1..n], (contained-in td1_i td2)
   By definition,
    (I (*or td1_1 .. td1_n) v b) = (or (I td1_1 v b) .. (I td1_n v b)).
   Use the inductive assumption of our lemma for each td1_i, i.e.

   For all td1_i and td2, v, and b,
    (and (null (gather-variables-in-descriptor td1_i))
         (null (gather-variables-in-descriptor td2))
         (contained-in td1_i td2)
         (I td1_i v b))
    =>
    (I td2 v b)
```

```
  We have already established that the first three antecedents hold.
  For any i such that (I td1_i v b) is true, use the inductive
  assumption to note that (I td2 v b) is true.  This establishes the
  goal.

Case 7.  td1 is a *rec descriptor

 Case 7.1.  td2 = (*or .. td1 .. )
   By definition, (contained-in td1 td2) = t.
   By definition, (I (*or .. td1 .. ) v b) = (or .. (I td1 v b) .. )
   Since (I td1 v b) is the antecedent in the conclusion, this makes
   the implication trivially true.

 Case 7.2.  td2 is a *rec descriptor
   This case is handled with a set of rules exhaustively covering all
   possible pairs of *rec descriptors.  Selected rules are proved
   separately.  For these proofs, see Appendix B-F.
   For a complete list of the rules, see Section 6.8.3.
   By definition for this case, contained-in tests the enabling
   condition for each rule in succession until it finds an
   appropriate case, and then the result is given by the action
   component of the rule.  A default rule which is unconditionally
   enabled ensures that all cases are handled.  Thus, by our
   computational induction schema, the proof for this case is by
   straightforward instantiation of the lemma associated with the
   appropriate rule.

 Case 7.3.  td2 falls under neither Case 7.1 nor 7.2.
   By definition, (contained-in td1 td2)
    = (contained-in (open-rec-descriptor-absolute td1) td2).
   This equality is maintained under canonicalization Rule 2 (See
   Section 6.7) for opening *rec descriptors.
   Therefore, we appeal to the inductive hypothesis, instantiated
   with td1 = (open-rec-descriptor-absolute td1).

Case 8.  td2 = (*or td2_1 .. td2_n)

  By definition, (contained-in td1 td2)
    = (or (contained-in td1 td2_1) .. (contained-in td1 td2_n)).
  Also by definition, (I td2 v b) = (or (I td2_1 v b) .. (I td2_n v b)).

  Use the inductive assumption for any i in [1..n] that:

  For all td1, td2_i, v, and b,
   (and (null (gather-variables-in-descriptor td1))
        (null (gather-variables-in-descriptor td2_i))
        (contained-in td1 td2_i)
        (I td1 v b))
   =>
   (I td2_i v b)

  Choose any i such that (contained-in td1 td2_i).  We know one
  exists, by the expansion of H3.  Then use the inductive assumption
  for that i to establish (I td2_i v b).  This in turn establishes
  (or .. (I td2_i v b) .. ) and our goal, (I td2 v b).

Case 9.  td2 is a *rec descriptor
 By definition, (contained-in td1 td2)
  = (contained-in td1 (open-rec-descriptor-absolute td2)).
 This transformation is valid under canonicalization Rule 2 for
 opening *rec descriptors.  We appeal to the inductive hypothesis,
 instantiated with td2 = (open-rec-descriptor-absolute td2).

Case 10.  td1 = (*dlist td1_1 .. td1_n) and td2 = (*dlist td2_1 .. td2_n)
   v is a list of values (v_1 .. v_n).

 By definition,
```

```
  (contained-in td1 td2)
    = (and (contained-in td1_1 td2_1) .. (contained-in td1_n td2_n))

 Consider the inductive assumptions for each i,

 For any descriptors td1_i and td2_i, value v_i, and binding b,
  (and (null (gather-variables-in-descriptor td1_i))
       (null (gather-variables-in-descriptor td2_i))
       (contained-in td1_i td2_i)
       (I td1_i v_i b))
  =>
  (I td2_i v_i b)

 We can use these inductive hypotheses because:

  1) If there are no variables in td1 and td2, then there are no
     variables in the components of td1 and td2.
  2) (contained-in td1_i td2_i) is established by case assumption.
  3) For all i, (I td1_i v_i b) is established by the
     antecedent in our main goal, (I td1 v b), which expands directly to
     (and (I td1_1 v_1 b) .. (I td1_n v_n b)).

 We can choose an arbitrary b for all the inductive hypotheses to share,
 since in the absence of variables the binding has no bearing on the
 value of I.  This same b will suffice for our goal, which is then
 established by the conjunction of the conclusions of our inductive
 hypotheses.

Case 11.  td1 = (*cons td1car td1cdr) and td2 = (*cons td2car td2cdr)
 This is analogous to Case 10, where the lists of arguments are
 of length 2, the descriptor constructor is *cons rather than *dlist,
 and the value is (cons (car v) (cdr v) rather than (v_1 .. v_n).
 The proof is exactly analogous to Case 10, with these surface issues
 taken into consideration.

Case 12.  td1 = (*cons td1car td2cdr) and td2 is a primitive descriptor
  (contained-in td1 td2) = nil, falsifying the antecedent H3.

Case 13.  td1 is a primitive descriptor and td2 = (*cons td2car td2cdr).
  (contained-in td1 td2) = nil, falsifying the antecedent H3.

Case 14. td1 = $integer (Without loss of generality, this same argument
  applies to the cases for other primitive descriptors)

 Case 14.1.  td2 = $integer
  To show (contained-in td1 td2) = t, we consider two cases.

  Case 14.1.1  (integerp v)
   (I td1 v b) = t for any b
   (I td2 v b) = t for any b

  Case 14.1.2  (not (integerp v))
   (I td1 v b) = nil for any b, falsifying H4.

 Case 14.2   td2 = some other primitive
   (contained-in td1 td2) = nil, falsifying the antecedent H3.

 This completes an exhaustive case analysis.
QED.
```

B-F The CONTAINED-IN *REC Rules

This section gives the proof approach for the case where both DESCRIPTOR1 and DESCRIPTOR2 are *REC descriptors, which is handled by application of a collection of special purpose rules. Each rule has an enabling condition which is essentially a pattern matching predicate on the forms of the descriptors. Each rule also has an action, which determines whether containment exists for that case. The first such rule which is enabled for given arguments makes the complete determination. Collectively, the proofs of soundness of all these rules is sufficient to prove the soundness of the algorithm for this case, since collectively they provide an exhaustive case analysis.

Each rule coincides with a lemma stated in terms of INTERP-SIMPLE. The statement of these lemmas and their proofs are supplied for only some of the rules stated here, but these examples should provide adequate demonstration that the other rules are provable by the same techniques. In particular, Rule Contain-*REC2 was chosen for proof because it is a simple rule illustrating the basic technique, and Rule Contain-*REC5 was chosen because it is perhaps the most complex of the rules.

Recall that the lemma corresponding to each rule is a member of the nest of lemmas, including CONTAINED-IN-OK, which we are proving simultaneously via computational induction on the length of the computation by CONTAINED-IN. This allows the assumption of each lemma with regard to every subsidiary call within the computation.

There are numerous rules, and they are all stated in Section 6.8.3. The rules are stylistically quite similar. Most reduce the problem to one or more containment problems for component descriptors, and hence the assumption of CONTAINED-IN-OK in their proofs.

Since the CONTAINED-IN algorithm is only invoked when both of its arguments are variable-free, we have a standing assumption on all the rules that the descriptors in question are variable-free. Thus their components are also variable-free, and CONTAINED-IN may be called recursively.

Many of the rules have a NIL result, indicating that containment does not exist. The soundness proofs for these rules are trivial, since returning a NIL result negates one of the antecedents to the lemma which would correspond to any such rule.

Furthermore, there are some rules whose names end with a "’". These rules are generally symmetric variants of the rules with the same, but unprimed, names. Their proofs would be similarly symmetric. Rule Contain-*REC2 has just such a variant.

The rules are employed in the following manner. When CONTAINED-IN is called with the two *REC descriptors, it invokes each rule in sequence until it finds one whose enabling condition establishes that the *REC descriptors are of the form indicated by canonical example as FOO and BAR. The rule then indicates the definition of CONTAINED-IN for *REC descriptors of that form. The lemmas are generically of the same following form as CONTAINED-IN-OK:

```
For any variable-free *rec descriptors of the form foo and bar,
Lisp value v, and type variable binding b1,

 (and (contained-in foo bar)
      (I foo v b1))
 =>
 for some binding b2,
  (I bar v b2)
```

but with the (CONTAINED-IN FOO BAR) hypothesis replaced by the right hand side of the containment rule. This simply reflects the fact that for this case, the right hand side provides the definition of CONTAINED-IN for this case.

So consider the following containment rule.

```
Rule Contain-*REC2

 (contained-in (*rec foo (*or $nil (*cons d1 (*recur foo))))
               (*rec bar (*or $nil (*cons d2 (*recur bar)))))
 =
 (contained-in d1 d2)
```

This rule reduces the containment problem for *REC descriptors of the form characterized by FOO and BAR to the containment problem for D1 and D2.

For notational convenience, let

```
 "foo" denote (*rec foo (*or $nil (*cons d1 (*recur foo))))
 "bar" denote (*rec bar (*or $nil (*cons d2 (*recur bar))))
```

We wish to prove a lemma validating this rule as an inference in the CONTAINED-IN algorithm. The rule corresponds to and is justified by the lemma:

```
Lemma CONTAIN-*REC2-OK
 For any (variable-free) descriptors d1 and d2, where foo is a
 descriptor of the form (*rec foo (*or $nil (*cons d1 (*recur foo))))
 and bar is a descriptor of the form
 (*rec bar (*or $nil (*cons d2 (*recur bar)))),
 for any Lisp value v, and type variable binding b1,

  (and
H1  (null (gather-variables-in-descriptor foo))
H2  (null (gather-variables-in-descriptor bar))
H3  (contained-in d1 d2)
H4  (I foo v b1))
  =>
  For some b2, (I bar v b2)
```

We will use CONTAINED-IN-OK as an inductive hypothesis.

```
Proof of Lemma CONTAIN-*REC2-OK

 We will induct on the structure of the value v.

 In H4, (I foo v b1) expands by definition of I to

 (or (I $nil v b1) (I (*cons d1 foo) v b1)),

 which expands further, giving the revised hypothesis,

H4' (or (I $nil v b1)
        (and (consp v) (I d1 (car v) b1) (I foo (cdr v) b1))).

 Similarly in the conclusion, (I bar v b2) expands to

 (or (I $nil v b2) (I (*cons d2 bar) v b2)),

 which expands further, giving the revised conclusion

  For some b2,
   (or (I $nil v b2)
       (and (consp v) (I d2 (car v) b2) (I bar (cdr v) b2))).
```

```
 Now let us consider the cases suggested by H4'.

 Case 1.  (I $nil v b1)
  This establishes the first disjunct of the conclusion directly.

 Case 2.  (and (consp v) (I d1 (car v) b1) (I foo (cdr v) b1))

 Here we will use Lemma CONTAINED-IN-OK as an inductive hypothesis
 on both (car v) and (cdr v).  Instantiate it once with td1 = d1,
 td2 = d2, b1 = b1, and v = (car v).  We need to relieve its
 hypotheses.  Since d1 is syntactically a part of foo, H1
 guarantees (null (gather-variables-in-descriptor d1)).
 Likewise for H2 and d2.  H3 is preserved.  Our case assumption
 relieves its H4.  Thus, we can use the conclusion:

H5  For some b2', (I d2 (car v) b2')

 Now instantiate Lemma CONTAINED-IN-OK again, with td1 = foo,
 td2 = bar, v = (cdr v), and b1 = b1.  Again, H1, H2, and our
 case assumption relieve its hypotheses, giving us

H6  For some b2'', (I bar (cdr v) b2'')

 Since foo, bar, d1, and d2 are all variable-free,
 the contents of the bindings quantified in H5, H6, and the
 conclusion are all immaterial.  For the sake of clean exposition,
 then, we simply instantiate b2, b2', and b2'' with b1.
 Then, H5, H6, and the (consp v) conjunct from our case assumption
 together establish the conclusion.  QED
```

Now let us consider:

```
Rule Contain-*REC5

 (contained-in (*rec foo (*or d1 (*cons d2 (*recur foo))))
               (*rec bar (*or d3 (*cons d4 (*recur bar)))))
 =
 (or (and (contained-in d1 d3) (contained-in d2 d4))
     (contained-in foo d3)
     (and (contained-in d1 bar) (contained-in d2 d4)))
```

This rule corresponds to and is justified by the following lemma:

```
Lemma CONTAIN-*REC5-OK

 For any descriptors d1, d2, d3, and d4, where foo is a descriptor of
 the form (*rec foo (*or d1 (*cons d2 (*recur foo)))) and bar is a
 descriptor of the form (*rec bar (*or d3 (*cons d4 (*recur bar)))),
 For any value v and binding b1,

  (and
H1  (null (gather-variables-in-descriptor foo))
H2  (null (gather-variables-in-descriptor bar))
H3  (or (and (contained-in d1 d3) (contained-in d2 d4))
        (contained-in foo d3)
        (and (contained-in d1 bar) (contained-in d2 d4)))
H4  (I foo v b1))
  =>
  For some binding b2, (I bar v b2)
```

For notational convenience, let

```
 "foo" denote (*rec foo (*or d1 (*cons d2 (*recur foo))))
 "bar" denote (*rec bar (*or d3 (*cons d4 (*recur bar))))
```

Proof of Lemma CONTAIN-*REC5-OK

```
 Since foo and bar are variable-free, then d1, d2, d3, and d4, by
 virtue of being syntactically contained in either foo or bar, are
 also variable-free, and we will proceed under that notion without
 further mention.  Furthermore, there is no need to be concerned
 about the bindings b1, b2, or any others, since because there
 are no variables in the problem state, they never come into play
 in the evaluation of I.  So let us just use b1 as a witness for
 any binding.

 We will induct on the structure of v.

 In H4, (I foo v b1) expands by definition of I to

 (or (I d1 v b1) (I (*cons d2 foo) v b1)),

 which expands further to

H4' (or (I d1 v b1)
        (and (consp v) (I d2 (car v) b1) (I foo (cdr v) b1)))

 Similarly in the conclusion, (I bar v b1) (using b1 for b2)
 expands to (or (I d3 v b1) (I (*cons d4 bar) v b1)), which
 expands further to the revised conclusion:

  (or (I d3 v b1)
      (and (consp v) (I d4 (car v) b1) (I bar (cdr v) b1)))

 Now proceed by the cases suggested by H3.

 Case 1. (and (contained-in d1 d3) (contained-in d2 d4))

  Case 1.1  v is an atomic value

   Since (consp v) cannot hold, H4' gives us
     (I d1 v b1)
   Instantiate Lemma CONTAINED-IN-OK with td1 = d1, td2 = d3, and
   v = v.  The first conjunct of the case assumption relieves its
   H3, and (I d1 v b1) relieves its H4, giving us,
     (I d3 v b1)
   which establishes the first disjunct of the conclusion.

  Case 1.2  v is a cons

   Now consider the cases suggested by H4'

   Case 1.2.1  (I d1 v b1)

    Instantiate Lemma CONTAINED-IN-OK as above and proceed to the
    goal in the same way.

   Case 1.2.2  (and (consp v) (I d2 (car v) b1) (I foo (cdr v) b1))

    Instantiate Lemma CONTAINED-IN-OK with td1 = d2, td2 = d4,
    and v = (car v).  Our case assumptions (contained-in d2 d4)
    and (I d2 (car v) b1) relieve its hypotheses, giving
      (I d4 (car v) b1)
    Now use our Lemma CONTAIN-*REC5-OK as the inductive hypothesis
    for (cdr v).  Instantiate it with td1 = foo, td2 = bar, and
    v = (cdr v).  Our Case 1 assumption and (I foo (cdr v) b1)
    from the Case 1.2.2 assumption let us use its conclusion,
      (I bar (cdr v) b1)
    These two results, plus our (consp v) case assumption establish
    the goal.

 Case 2.  (contained-in foo d3)

  Instantiate Lemma CONTAINED-IN-OK with td1 = foo, td2 = d3, and
```

```
  v = v.  The case assumption and the original H4 relieve its
  hypotheses.  This gives us
   (I d3 v b1)
  which establishes the goal.

 Case 3.  (and (contained-in d1 bar) (contained-in d2 d4))

  Case 3.1  v is an atom

   Instantiate Lemma CONTAINED-IN-OK with td1 = d1, td2 = bar, and
   v = v.  H4' gives us
     (I d1 v b1)
   This and our case assumption (contained-in d1 bar) allow us to
   use the conclusion,
     (I bar v b1)
   which establishes the goal.

  Case 3.2  v is a cons

   Consider cases suggested by H4'

   Case 3.2.1  (I d1 v b1)

    Proceed as with Case 3.1.

   Case 3.2.2  (and (consp v) (I d2 (car v) b1) (I foo (cdr v) b1))

    Instantiate Lemma CONTAINED-IN-OK with td1 = d2, td2 = d4,
    and v = (car v).  (contained-in d2 d4) from our Case 3
    assumption and (I d2 (car v) b1) from our Case 3.2.2 assumption
    relieve its hypotheses, giving us
      (I d4 (car v) b1)

    Now use Lemma CONTAIN-*REC5-OK as the inductive hypothesis
    for (cdr v).  Instantiate it with td1 = foo, td2 = bar, and
    v = (cdr v).  Our Case 3 assumption relieves its H3, and
    (I foo (cdr v) b1) from the Case 3.2.2 assumption relieves its
    H4, so we obtain its conclusion,
      (I bar (cdr v) b1)

    These two results, plus our (consp v) case assumption establish
    the goal.

 This completes the case analysis.  QED.
```

B.9 ICONTAINED-IN-OK

ICONTAINED-IN-OK is the soundness lemma for the ICONTAINED-IN algorithm, which is employed by CONTAINED-IN-INTERFACE whenever either of the descriptors contain type variables. The main part of the proof is in the section which follows immediately. One of its more significant cases is factored into Lemma ICONTAINED-IN-EQUAL-TDS, which is proved in the next section. Also deferred in the main text is the case where both DESCRIPTOR1 and DESCRIPTOR2 are *REC descriptors. The proof approach for this case is in the third section to follow.

For a description of ICONTAINED-IN, see Section 6.8.5.

B-G Proof of Lemma ICONTAINED-IN-OK

Here, as in all our other lemmas and proofs, "I" is a shorthand notation representing INTERP-SIMPLE when its second and third arguments are lists, INTERP-SIMPLE-1 when these arguments are a single descriptor and value.

Lemma ICONTAINED-IN-OK:

```
 For any descriptors td1 and td2, simple mapping m, Lisp value v,
 symbolic value reference vref, and binding b1 covering the
 variables in td1,

  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  (well-formed-mapping m vref b1)
H3  (icontained-in td1 (dapply-subst-list-1 m td2) vref)
H4  (I td1 v b1) )
  =>
  For some b2, (I td2 v b2)
```

Note: Recall that ICONTAINED-IN takes another argument, TERM-RECS, which is employed solely as a mechanism for terminating the computation in certain cases. As such, it has no effect on the soundness argument, which does not rely on termination. To avoid clutter in the lemmas and proofs, we simply omit further mention of it, though one could think of it as being universally quantified and arbitrarily instantiable.

Proof of Lemma ICONTAINED-IN-OK

```
 For brevity, we abbreviate the function name DAPPLY-SUBST-LIST-1
 with "A-S-1".

 Let us eliminate the existential quantifier on b2 from the outset,
 claiming and later demonstrating that a binding which would suffice
 is constructed by (make-binding-from-mapping m v b1), defined as
 follows:


(DEFUN MAKE-BINDING-FROM-MAPPING (M V B1 VREF)
  (IF (NULL M)
      NIL
    (IF (EQUAL (CDR (CAR M)) '$T)
        (CONS (CONS (CAR (CAR M)) T)
              (MAKE-BINDING-FROM-MAPPING (CDR M) V B1 VREF))
      (IF (EQUAL (CDR (CAR M)) '$NIL)
          (CONS (CONS (CAR (CAR M)) NIL)
                (MAKE-BINDING-FROM-MAPPING (CDR M) V B1 VREF))
        (IF (VARIABLE-NAMEP (CDR (CAR M)))
            (CONS (CONS (CAR (CAR M)) (CDR (ASSOC (CDR (CAR M)) B1)))
                  (MAKE-BINDING-FROM-MAPPING (CDR M) V B1 VREF))
          (IF (INP-EQUAL VREF (CDR (CAR M)))
              (CONS (CONS (CAR (CAR M))
                          (EVAL (SUBST (LIST 'QUOTE V)
                                       VREF
                                       (CDR (CAR M)))))
                    (MAKE-BINDING-FROM-MAPPING (CDR M) V B1 VREF))
              (MAKE-BINDING-FROM-MAPPING (CDR M) V B1 VREF)))))))

 The EVAL function used here is the normal Lisp EVAL, since this
 allows MAKE-BINDING-FROM-MAPPING to be executable in a normal
 Lisp world.  In our formal world, this call of EVAL corresponds
 to a call to E with the same form, any binding (since the form
 is a ground term), any world containing the SUBRs CAR, CDR,
 DLIST-ELEM and REC-TAIL, and any positive integer clock.  Since
 the only functions invoked are SUBRs, no clock ticks are
 consumed.  (Although DLIST-ELEM and REC-TAIL are not in our initial
 database of function signatures, they certainly could be.  We chose
 not to include them because the scenario just described here is
 somewhat hypothetical and is relevant only to the proof, not to
 the normal operation of the system.)

 Thus our theorem is transformed to:
```

For any descriptors td1 and td2, simple mapping m, Lisp value v, symbolic value reference vref, and binding b1 covering the variables in td1,

```
  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  (well-formed-mapping m vref b1)
H3  (icontained-in td1 (A-S m td2) vref)
H4  (I td1 v b1) )
  =>
  (I td2 v (make-binding-from-mapping m v b1 vref))
```

H2 guarantees that if (VARIABLE-NAMEP (CDR (CAR M))), (CDR (CAR M)) is mapped in B1, and also that

```
  (SUBST (LIST 'QUOTE V) VREF (CDR (CAR M)))
```

is a valid form to evaluate and will do so without error.

The proof is by computational induction on the length of the computation by ICONTAINED-IN. This allows the assumption of the lemma on every recursive call.

Our case analysis takes each case in sequence, as in the algorithm, so each one assumes the negation of the case assumptions of all previous cases.

Case 1. td1 = (A-S-1 m td2)
By definition, (icontained-in td1 (A-S-1 m td2) vref) = t

Since there are no value references in td1, our case assumption rules out the possibility that m maps &i to a value ref.

This case is captured in the following lemma:

Lemma ICONTAINED-IN-EQUAL-TDS

For any descriptors td1 and td2, simple mapping m containing no symbolic value references on the right hand side, symbolic value reference vref, Lisp value v, and type variable binding b1 covering the variables in td1,

```
  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  (well-formed-mapping m vref b1)
H3  (I td1 v b1)
H4  (equal td1 (dapply-subst-list-1 m td2)) )
  =>
  (I td2 v (make-binding-from-mapping m v b1 vref))
```

The proof of this lemma is in Appendix B-H.

Case 2. td1 = *empty
Trivial. For any v and b1, (I td1 v b1) = nil, falsifying the antecedent of the conclusion.

Case 3. (A-S-1 m td2) = *empty, and therefore td2 = *empty.
Trivial. By definition, (icontained-in td1 *empty m) = nil, falsifying H3.

Case 4. (A-S-1 m td2) = vref
In this case, m maps some type variable &i to vref, and td2 = &i. By definition of make-binding-from-mapping, then, the binding produced by (make-binding-from-mapping m v b1 vref) maps &i to v. Therefore, the conclusion simplifies as follows, by the definition of I:

```
 (I &i v (make-binding-from-mapping m v b1 vref)) = (equal v v).

Case 5. (and (value-refp (A-S-1 m td2))
             (not (equal (A-S-1 m td2) vref)))
 By definition (icontained-in td1 (A-S-1 m td2) vref) = nil
 falsifying hypothesis H3.

Case 6. td1 = *universal
  By definition for any td2 except *universal (already considered
  in Case 1), (icontained-in *universal td2 m) = nil, falsifying H3.

Case 7. (A-S-1 m td2) = *universal
    (and therefore td2 = *universal)
  For all v and b2, (I td2 *universal b2) = t, establishing the
  conclusion.

Case 8. td1 = (*or td1_1 .. td1_n)
  By definition,
   (icontained-in td1 (A-S-1 m td2) vref)
      = all i in [1..n],
         (icontained-in td1_i (A-S-1 m td2) vref)
  By definition,
   (I (*or td1_1 .. td1_n) v b) = (or (I td1_1 v b) .. (I td1_n v b)).
  This case follows by use of ICONTAINED-IN-OK as an inductive
  assumption for each i in 1..n.  For any i such that
  (I td1_i v b) is true, use the inductive assumption to note
  that, (I td2 v (make-binding-from-mapping m v b1 vref)) is true.
  This establishes the goal.

Case 9.  td1 is a *rec descriptor
 Case 9.1.  (A-S-1 m td2) = (*or .. td1 .. )
   By definition, (icontained-in td1 (*or .. td1 .. ) vref) = t.
   A-S-1 preserves the structure of td2, modifying only the
   positions in which variables appear.  We know, then, that td2 is
   of the form (*or .. td2_i ..), where (A-S-1 m td2_i) = td1.
   The conclusion simplifies as follows, by the definition of I:

    (I (*or .. td2_i ..) v (make-binding-from-mapping m v b1 vref))
     = (or .. (I td2_i v (make-binding-from-mapping m v b1 vref)) .. )

   From this point, we appeal to Lemma ICONTAINED-OK as an inductive
   assumption, instantiating with td1 = td1, td2 = td2_i, v = v,
   m = m, b1 = b1, and vref = vref.  This establishes the goal by
   establishing the disjunct displayed above.

   Note: The "equality" test the system employs to compare the first
    argument of icontained-in to the disjuncts is really an isomorphism
    test which allows the disjunct to be identical except for the
    name used as a *rec label.  A trivial observation is that for
    any two name-isomorphic *rec descriptors rec1 and rec2,
    for all v and b, (I rec1 v b) => (I rec2 v b).  Thus, name
    isomorphism is all that is required here.

 Case 9.2.  (A-S-1 m td2) is a *rec descriptor
   This case is handled with a set of rules exhaustively covering all
   possible pairs of *rec descriptors.  For a complete list of the rules
   and for proofs of some selected rules, see Appendix B-I.
   By definition for this case, icontained-in tests the enabling
   condition for each rule in succession until it finds an
   appropriate one, and then the result is given by the action
   component of the rule.  Thus, by our computational induction
   schema, the proof for this case is by straightforward
   instantiation of the lemma associated with the appropriate
   rule.

 Case 9.3.  (A-S-1 m td2) falls under neither Case 9.1 nor 9.2.
   By definition, (icontained-in td1 (A-S-1 m td2) vref)
```

```
      = (icontained-in (open-rec-descriptor-absolute td1)
                       (A-S-1 m td2)
                       vref).
    We appeal to the inductive assumption instantiating with
    td1 = (open-rec-descriptor-absolute td1).  Note that
    open-rec-descriptor-absolute introduces no new variables, so
    H1 is preserved, H3 is preserved by the definition of
    icontained-in (as above), and H4 is guaranteed by
    canonicalization Rule 2 for opening *rec descriptors.
```

Case 10. (A-S-1 m td2) = (*or $td2_1'$ .. $td2_n'$)
Recall that applying our mapping m to td2 preserves structure except for variables. Thus, we can see that td2 is of the form (*or $td2_1$ .. $td2_n$) and

(*or $td2_1'$ .. $td2_n'$)
=
(*or (A-S-1 m $td2_1$) .. (A-S-1 m $td2_n$))

Since the cases where td1 is one of td2, *universal, *empty, an *or, or a *rec have been covered above, we know td1 is a *cons or a primitive.

By definition, (icontained-in td1 (A-S-1 m td2) vref)
= (or (icontained-in td1 (A-S-1 m $td2_1$) vref)
..
(icontained-in td1 (A-S-1 m $td2_n$) vref)).

Also by definition of I,
(I td2 v b2) = (or (I $td2_1$ v b2) .. (I $td2_n$ v b2)).

We will use ICONTAINED-IN-OK as an inductive assumption applied for each $td2_i$ as necessary. If there is no i in [1..n] such that (icontained-in td1 (A-S-1 m $td2_i$) vref) is true, then H3 is false. So there is such an i, and we use the inductive assumption for that i to establish
(I $td2_i$ v (make-binding-from-mapping m v b1 vref)).
This in turn establishes
(or .. (I $td2_i$ v (make-binding-from-mapping m v b1 vref)) ..)
and therefore
(I td2 v (make-binding-from-mapping m v b1 vref)).
That is our goal.

Case 11. (A-S-1 m td2) is a *rec descriptor
By definition,

```
(icontained-in td1 (A-S-1 m td2) vref)
 = (icontained-in td1
                  (open-rec-descriptor-absolute (A-S-1 m td2))
                  vref).
```

We appeal to the inductive assumption, noting that open-rec-descriptor-absolute introduces no new variables, so H1 is preserved, and H3 is preserved by the definition of icontained-in (as above).

To use the lemma as an inductive assumption in this manner requires that the second argument be an A-S-1 form, but in the second argument here, the A-S-1 form is encapsulated in a call to open-rec-descriptor-absolute. Note, however, the following lemma, which is obviously true.

Lemma A-S-1-OPEN-REC-COMMUTE
For any *rec descriptor td and well-formed mapping m,

```
(open-rec-descriptor-absolute (A-S-1 m td))
 =
(A-S-1 m (open-rec-descriptor-absolute td))
```

Thus, we can properly use the inductive assumption.

```
Case 12.  td1 = (*dlist td1_1 .. td1_n) and
          (A-S-1 m td2) = (*dlist td2_1' .. td2_n')
```

In this case, v is a list of values, which we denote (v$_1$ .. v$_n$).

We know, because A-S-1 preserves structure, that td2 may be written as (*dlist td2$_1$ .. td2$_n$), where

```
 (*dlist td2_1' .. td2_n')
  =
 (*dlist (A-S-1 m td2_1) .. (A-S-1 m td2_n))
```

By definition,

```
 (icontained-in
   (*dlist td1_1 .. td1_n)
   (*dlist (A-S-1 m td2_1) .. (A-S-1 m td2_n))
   vref)
  =
 (and (icontained-in td1_1 (A-S-1 m td2_1) (dlist-elem vref 1))
      ..
      (icontained-in td1_n (A-S-1 m td2_n) (dlist-elem vref n)))
```

Consider our inductive assumption, ICONTAINED-IN-OK instantiated for all i, with td1 = td1$_i$, td2 = td2$_i$, v = v$_i$, vref = (dlist-elem vref i), b1 = b1, and m = m. (By (dlist-elem vref i) we mean the form whose CAR is dlist-elem, whose cadr is the symbolic value reference vref, and whose caddr is the integer i.) Thus, the inductive assumption is:

```
  (and
H1  (disjoint (gather-variables-in-descriptor td1_i)
              (gather-variables-in-descriptor td2_i))
H2  (well-formed-mapping m (dlist-elem vref i) b1)
H3  (icontained-in td1_i (A-S-1 m td2_i) (dlist-elem vref i))
H4  (I td1_i v_i b1) )
  =>
  (I td2_i
     v_i
     (make-binding-from-mapping m v_i b1 (dlist-elem vref i)))
```

We can use each instance of the inductive assumption, since

1. The disjointness of variables in td1 and td2 guarantees the disjointness of variables in the respective components of td1 and td2.
2. (well-formed-mapping m vref b1) =>
   (well-formed-mapping m (dlist-elem vref i) b1)
   since the well-formed-mapping predicate's concern with the second argument is limited to its root, and
   (root-of-var-ref vref) = (root-of-var-ref (dlist-elem vref i)).
   See the discussion of WELL-FORMED-MAPPING in Section 7.8.
3. H3 of each inductive hypothesis is guaranteed by the expansion of our H3, given above, and
4. H4 of the main theorem expands directly into a conjunction of terms for all i, (I td1$_i$ v$_i$ b1).

The expansion by definition of I of our goal,
(I td2 v (make-binding-from-mapping m v b1 vref)), is

```
(and (I td2_1 v_1 (make-binding-from-mapping m v b1 vref))
     ..
     (I td2_n v_n (make-binding-from-mapping m v b1 vref)))
```

By Lemma EXTENDS-BINDING-MONOTONIC (See Appendix B.3), if we can

```
 show that for all i in 1..n, (make-binding-from-mapping m v b1 vref)
 extends (make-binding-from-mapping m v_i b1 (dlist-elem vref i)),
 we will have established our goal.  This requires us to show that
 every binding in
 (make-binding-from-mapping m v_i b1 (dlist-elem vref i))
 is preserved in (make-binding-from-mapping m v b1 vref).  By
 examining the definition of make-binding-from-mapping above,
 one may clearly see that if a variable &j is mapped to $T,
 $NIL, or a variable &k in b1, then make-binding-from-mapping
 produces the same binding element in both cases.  Neither
 the second nor fourth arguments are involved.  If &j is mapped
 to a symbolic value reference m-vref, there are two cases to
 consider.  First, if (dlist-elem vref i) does not occur in
 m-vref, there will be no binding element for &j in
 (make-binding-from-mapping m v_i b1 (dlist-elem vref i)).
 So we do not need to be concerned about the binding for &j
 in (make-binding-from-mapping m v b1 vref) to establish the
 extends property.  Second, if (dlist-elem vref i) does occur
 within m-vref, then
 (make-binding-from-mapping m v_i b1 (dlist-elem vref i))
 includes a binding element mapping &j to the value produced
 by evaluating m-vref with the form (dlist-elem vref i) replaced
 by the value v_i.  Also,
 (make-binding-from-mapping m v b1 vref) includes a binding
 element mapping &j to the value produced by evaluating m-vref
 with vref replaced by v.  Since evaluating (dlist-elem v i)
 gives v_i, the values in the two bindings are equal, and
 thus the binding of &j from the inductive assumption is preserved
 in the larger result.  This establishes the goal.[41]

Case 13.  td1 = (*cons td1car td1cdr) and td2 = (*cons td2car td2cdr)
 This is handled by ICONTAINED-IN analogously to Case 12, where
 the length of the *dlist is two, where *cons is the constructor
 instead of *dlist, where car and cdr are the accessors rather
 than dlist-elem, and the value v is a cons rather than a list
 of values.  The argument is exactly analogous.

Case 14.  td1 = (*cons td1car td2cdr) and (A-S-1 m td2) is a
  primitive descriptor.

  By definition,
  (icontained-in td1 (A-S-1 m td2) vref) = nil,
  falsifying H3.

Case 15.  td1 is a primitive descriptor and
  (A-S-1 m td2) = (*cons td2car td2cdr).

  By definition,
  (icontained-in td1 (*cons td2car td2cdr) vref) = nil,
  falsifying H3.

Case 16. td1 = $integer and (A-S-1 m td2) = some other
  primitive descriptor.

  By definition,
  (icontained-in td1 (A-S-1 m td2) vref) = nil,
  falsifying H3.

This completes an exhaustive case analysis.  QED.
```

41 It so happens that (make-binding-from-mapping m v b1 vref) is the merge of all the bindings (make-binding-from-mapping m v_i b1 (dlist-elem vref i)), but showing this is a stronger result than we need.

B-H Proof of Lemma ICONTAINED-IN-EQUAL-TDS

```
Lemma ICONTAINED-IN-EQUAL-TDS

 For any descriptors td1 and td2, simple mapping m containing no
 symbolic value references on the right hand side, symbolic value
 reference vref, Lisp value v, and type variable binding b1
 covering the variables in td1,

  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  (well-formed-mapping m vref b1)
H3  (I td1 v b1)
H4  (equal td1 (dapply-subst-list-1 m td2)) )
  =>
  (I td2 v (make-binding-from-mapping m v b1 vref))
```

As in the proof of Lemma ICONTAINED-IN-OK, we abbreviate the function DAPPLY-SUBST-LIST-1 with "A-S-1".

```
Proof of Lemma ICONTAINED-IN-EQUAL-TDS

 We will proceed by a computational induction on the length of
 the computation by I.

 Consider by cases the structure of td2.

 Case 1.  td2 = $integer   (Without loss of generality, the cases
   where td2 is one of $character, $nil, $non-integer-rational,
   $non-t-nil-symbol, $string $t, *empty, or *universal follow
   the same simple argument.)

  Since td2 is variable-free, (A-S-1 m td2) = td2, and the binding
  constructed by (make-binding-from-mapping m v b1 vref) has no
  effect on the value of
   (I td2 v (make-binding-from-mapping m v b1 vref)).
  H3 and H4 establish the conclusion.

 Case 2.  td2 = &i

  Consider by cases the possible values of (assoc &i m)

  Case 2.1.  (assoc &i m) = nil   (i.e., &i is not mapped in m)

   Then (A-S-1 m td2) = td2.  But by H4, td1 = td2, and by H1,
   the variables in td1 and td2 are distinct.  Since td2 is a
   variable, we have a contradiction.

  Case 2.2.  (assoc &i m) = (&i . $t)

   (A-S-1 m td2) = td2 = $t.  By H4, td1 = $t.  Expanding the
   definition of I in H3,
    (I td1 v b1) = (I $t v b1) = (equal v t).
   Thus, for any binding b’, (I td2 v b’) = (I $t t b’) = t,
   establishing the goal.

  Case 2.3.  (assoc &i m) = (&i . $nil)

   Exactly analogous to Case 2.2.

  Case 2.4   (assoc &i m) = (&i . &j)

   Then (A-S-1 m &i) = &j.  By H4, td1 = &j.  By H3,
    (I td1 v b1) = (I &j v b1)
```

= (by definition of I)
(equal v (cdr (assoc &j b1)))
By the definition of make-binding-from-mapping,
(make-binding-from-mapping m v b1 vref) includes the element
(&i . (cdr (assoc &j b1))). Therefore, in the conclusion,
(I &i v (make-binding-from-mapping m v b1 vref))
=
(equal v (cdr (assoc &i b1)))
which we established above.

Case 3. td2 = (*or td2$_1$ .. td2$_n$)

The following lemma is obviously true from the definition of A-S-1.

For any descriptors td1 .. tdn and mapping m,
(A-S-1 m (*or td1 .. tdn)) = (*or (A-S-1 m td1) .. (A-S-1 m tdn))

By this lemma,
(A-S-1 m td2)
=
(*or (A-S-1 m td2$_1$) .. (A-S-1 m td2$_n$))

Therefore, for some td1$_1$ .. td1$_n$,
td1 = (*or td1$_1$ .. td1$_n$), where for each i in 1..n,
(by H4) td1$_i$ = (A-S-1 m td2$_i$). In H3,

(I (*or td1$_1$ .. td1$_n$) v b1)
= (by definition of I)
(or (I td1$_1$ v b1) .. (I td1$_n$ v b1))

Using our lemma as an inductive hypothesis for whichever i
(I td1$_i$ v b1) holds,

(I td2$_i$ v (make-binding-from-mapping m v b1 vref))

This is one of the disjuncts in the conclusion, when we expand
by the definition of I.

(I (*or td2$_1$ .. td2$_n$)
v
(make-binding-from-mapping m v b1 vref))
=
(or (I td2$_1$ v (make-binding-from-mapping m v b1 vref))
..
(I td2$_n$ v (make-binding-from-mapping m v b1 vref)))

Case 4. td2 = (*dlist td2$_1$ .. td2$_n$)

The following lemma is obviously true from the definition of A-S-1.

For any descriptors td1 .. tdn and mapping m,
(A-S-1 m (*dlist td1 .. tdn))
= (*dlist (A-S-1 m td1) .. (A-S-1 m tdn))

By this lemma,
(A-S-1 m td2)
=
(*dlist (A-S-1 m td2$_1$) .. (A-S-1 m td2$_n$))

Therefore, for some td1$_1$ .. td1$_n$,
td1 = (*dlist td1$_1$ .. td1$_n$), where for each i in 1..n,
(by H4) td1$_i$ = (A-S-1 m td2$_i$). In H3,

(I (*dlist td1$_1$ .. td1$_n$) (v$_1$ .. v$_n$) b1)
= (by definition of I)
(and (I td1$_1$ v$_1$ b1) .. (I td1$_n$ v$_n$ b1))

```
We use our lemma as an inductive hypothesis for all i,
instantiating with td1 = td1_i, td2 = td2_i, v = v_i, m = m,
vref = the form (dlist-elem vref i), and b1 = b1.  This gives us

(and (I td2_1
        v_1
        (make-binding-from-mapping m v_1 b1 (dlist-elem vref 1)))
     ..
     (I td2_n
        v_n
        (make-binding-from-mapping m v_n b1 (dlist-elem vref n))))

(Here by the notation (dlist-elem vref i), we mean the form whose
car is the atom dlist-elem, whose cadr is vref, and whose caddr
is the integer i, rather than the value of applying the function
dlist-elem to the symbolic value reference vref.)

It is given that m contains no symbolic value references on its
right hand sides.  (If it had, (make-binding-from-mapping m v b1 vref)
would include a value reference.  This is not possible, since
(make-binding-from-mapping m v b1 vref) = td1, and td1 cannot
include a symbolic value reference.)  By examination of the definition
of make-binding-from-mapping, it is clear that

(make-binding-from-mapping m v_i b1 (dlist-elem vref i))
 =
(make-binding-from-mapping m v b1 vref)

since in the absence of symbolic value references in m, neither
the second (value) nor the fourth (value reference) parameters
play any role in the result.  Substituting this equality into
the result above gives us

(and (I td2_1 v_1 (make-binding-from-mapping m v b1 vref))
     ..
     (I td2_n v_n (make-binding-from-mapping m v b1 vref)))

This is identical to the conclusion with I expanded.

Case 5.  td2 = (*cons td2car td2cdr)

This case is analogous to Case 4, with *cons used as the constructor
rather than *dlist, car and cdr as the destructors rather than
dlist-elem, and the value v is a cons rather than a list of values.

Case 6.  td2 = (*rec foo foobody)

Use the inductive assumption, instantiated with
td1 = (open-rec-descriptor-absolute td1),
td2 = (open-rec-descriptor-absolute td2),
m = m, v = v, b1 = b1, and vref = vref.  The hypotheses are
relieved as follows. (We denote each Hi in the inductive assumption
with Hi').  Open-rec-descriptor-absolute introduces no
new variables, so H1 establishes H1'.  H2 is equal to H2'.
H3 establishes H3' by Canonicalization Rule 2.  Consider the lemma

Lemma A-S-1-OPEN-REC-COMMUTE
 For any *rec descriptor td and well-formed mapping m,

 (open-rec-descriptor-absolute (A-S-1 m td))
  =
 (A-S-1 m (open-rec-descriptor-absolute td))


From this lemma, we see H4 establishes H4'.  The conclusion,
then, gives us
```

```
  (I (open-rec-descriptor-absolute td2)
     v
     (make-binding-from-mapping m v b1 vref))

  which, by Canonicalization Rule 2 is equal to our goal.

 This completes an exhaustive case analysis.  QED.
```

B-I The ICONTAINED-IN *REC Rules

What follows is a list of all the containment rules used in ICONTAINED-IN to handle the case where both arguments are *REC descriptors. Each rule gives the definition of ICONTAINED-IN for the case where the arguments are *REC descriptors of a certain form (where some previous case in ICONTAINED-IN did not apply). Each rule coincides with a lemma stated in terms of the descriptor interpreter. The statement of these lemmas and their proofs are supplied for only some of the rules stated here, but these examples should provide adequate demonstration that the other rules are provable by the same techniques. Most of these rules are exactly analogous to the similarly numbered Contain-*Rec rules and are justified by the same means as those rules. The ones which are different in nature are those where type variables or Lisp variable references appear in the arguments, and these rules are the ones which are presented for proof. Some of the rules yield a NIL result for ICONTAINED-IN, and we note that the proofs of these rules are trivial, since a NIL result negates one of the antecedents to the lemma which would correspond to the rule.

Note that descriptors appearing within replicating components of the *REC descriptors are variable-free, by the definition of well-formedness for *REC descriptors. (For a discussion, see Section 5.2.) Thus, the rules frequently have an invocation of the CONTAINED-IN algorithm, rather than ICONTAINED-IN, on the right hand side. For example, see Rule IContain-*Rec2. For the same reason, descriptors appearing within replicating components are also free of symbolic value references, since such a reference could have only gotten there via replacement of a type variable in application of a mapping.

```
Rule IContain-*REC1
 Where the two *recs differ only in name,

 (icontained-in (*rec foo ( .. (*recur foo) .. ))
                (*rec bar ( .. (*recur bar) .. ))
                vref)
 = t

The proof of the following rule is like the proof for
Rule Contain-*REC2. (See Appendix B-F.)

Rule IContain-*REC2
 (icontained-in (*rec foo (*or $nil (*cons d1 (*recur foo))))
                (*rec bar (*or $nil (*cons d2 (*recur bar))))
                vref)
 =
 (contained-in d1 d2)

The proof of the following is also like the proof for
Rule Contain-*REC2, with the *CONS arguments reversed.

Rule IContain-*REC2'
 (icontained-in (*rec foo (*or $nil (*cons (*recur foo) d1)))
                (*rec bar (*or $nil (*cons (*recur bar) d2)))
                vref)
 =
 (contained-in d1 d2)

Rule IContain-*REC3
```

```
 Where d1 is either a primitive (non-cons) descriptor or an
 *OR of primitive non-cons descriptors, and similarly for d3,

 (icontained-in (*rec foo (*or d1 (*cons d2 (*recur foo))))
                (*rec bar (*or d3 (*cons d4 (*recur bar))))
                vref)
 =
 (and (contained-in d1 d3) (contained-in d2 d4))

Rule IContain-*REC3'
 Where d1 and d3 are either primitive non-cons descriptors or
 *ORs of primitive non-cons descriptors,

 (icontained-in (*rec foo (*or d1 (*cons (*recur foo) d2)))
                (*rec bar (*or d3 (*cons (*recur bar) d4)))
                vref)
 =
 (and (contained-in d1 d3) (contained-in d2 d4))

The following is self-evident.  (*rec bar (*or *universal .. )) is
equivalent to *universal.

Rule IContain-*REC40
 (icontained-in (*rec foo  .. )
                (*rec bar (*or *universal .. ))
                vref)
 = t

The following rule is proved below.

Rule IContain-*REC41
 (icontained-in (*rec foo .. )
                (*rec bar (*or vref (*cons d2 (*recur bar))))
                vref)
 = t

The proof of IContain-*REC42 follows easily in the same manner as
IContain-*REC41.

Rule IContain-*REC42
 (icontained-in (*rec foo .. )
                (*rec bar (*or (*or .. vref .. ) (*cons .. )))
                vref)
 = t

The following rule is proved below.

Rule IContain-*REC43
 Where d3 is a primitive descriptor or a disjunction of primitive
 descriptors,

 (icontained-in
   (*rec foo (*or d3 (*cons d4 (*recur foo))))
   (*rec bar (*or (rec-tail vref) (*cons d2 (*recur bar))))
   vref)
 =
 (contained-in d4 d2)

The following rule is proved below.

Rule IContain-*REC44

 (icontained-in (*rec bim (*or &i (*cons d1 (*recur bim))))
                (*rec bar (*or &i (*cons d2 (*recur bar))))
                vref)
 =
 (contained-in d1 d2)

The proof of the following is similar to that of IContain-*REC44.
```

```
Rule IContain-*REC45
 Where d1 and d2 contain no variables,

 (icontained-in
    (*rec bim (*or &i (*cons d1 (*recur bim))))
    (*rec bar (*or (*or .. &i .. ) (*cons d2 (*recur bar))))
    vref)
 =
 (contained-in d1 d2)
```

The proof of the following rule is trivial, since the NIL result negates the antecedent that the result is non-NIL.

```
Rule IContain-*REC46
 Where d2, d3, and d4 contain no variables,

 (icontained-in (*rec foo (*or &j (*cons d2 (*recur foo))))
                (*rec bar (*or d3 (*cons d4 (*recur bar))))
                vref)
 = nil
```

The proof of the following rule is exactly analogous to the proof for Rule *Contain5.

```
Rule IContain-*REC5
 When there are variables in either foo or bar,

 (icontained-in (*rec foo (*or d1 (*cons d2 (*recur foo))))
                (*rec bar (*or d3 (*cons d4 (*recur bar))))
                vref)
 =
 (or (and (contained-in d1 d3) (contained-in d2 d4))
     (contained-in foo d3)
     (and (contained-in d1 bar) (contained-in d2 d4)))

Rule IContain-*REC6
 There are no type variables anywhere and d1 and d4 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors.

 (icontained-in
    (*rec foo (*or d1 (*cons d2 (*cons d3 (*recur foo)))))
    (*rec bar (*or d4 (*cons d5 (*recur bar))))
    vref)
 =
 (and (contained-in d1 d4)
      (contained-in d2 d5)
      (contained-in d3 d5))
```

The proof of the following rule is trivial.

```
Rule IContain-*REC6'
 There are no type variables anywhere and d1 and d4 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors.

 (icontained-in
   (*rec bar (*or d4 (*cons d5 (*recur bar))))
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*recur foo)))))
   vref)
 =
 nil

Rule IContain-*REC7
 There are no type variables anywhere and d1 and d4 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in
```

```
   (*rec foo (*or d1 (*cons (*recur foo) (*recur foo))))
   (*rec bar (*or d2 (*cons (*recur bar) (*recur bar))))
   vref)
 =
 (contained-in d1 d2)
```

Rule IContain-*REC8

```
 There are no type variables anywhere and d1 and d5 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*recur bar))))
   vref)
 =
 (and (contained-in d1 d5)
      (contained-in d2 d6)
      (contained-in d3 d6)
      (contained-in d4 d6))
```

The proof of the following rule is trivial.

Rule IContain-*REC8’

```
 There are no type variables anywhere and d1 and d5 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in
   (*rec bar (*or d5 (*cons d6 (*recur bar))))
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   vref)
 =
 nil
```

The proof of the following rule is trivial.

Rule IContain-*REC9

```
 There are no type variables anywhere and d1 and d5 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*cons d7 (*recur bar)))))
   vref)
 =
 nil
```

The proof of the following rule is trivial.

Rule IContain-*REC9’

```
 There are no type variables anywhere and d1 and d5 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in
   (*rec bar (*or d5 (*cons d6 (*cons d7 (*recur bar)))))
   (*rec foo (*or d1 (*cons d2 (*cons d3 (*cons d4 (*recur foo))))))
   vref)
 =
 nil
```

Rule IContain-*REC10

```
 There are no type variables anywhere and d2 and d4 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in (*rec foo (*cons d1 (*or d2 (*recur foo))))
```

```
                (*rec bar (*cons d3 (*or d4 (*recur bar))))
                vref)
 =
 (and (contained-in d1 d3) (contained-in d2 d4))

Rule IContain-*REC11
 There are no type variables anywhere and d2 and d3 allow no
 *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in (*rec foo (*cons d1 (*or d2 (*recur foo))))
                (*rec bar (*or d3 (*cons d4 (*recur bar))))
                vref)
 =
 (and (contained-in d1 d4) (contained-in d2 d3))

The proof of the following rule is trivial.

Rule IContain-*REC11'
 There are no type variables in bar and d3 is a
 primitive descriptor or a disjunction of primitive descriptors,

 (icontained-in (*rec bar (*or d3 d4))
                (*rec foo (*cons d1 d2)))
 =
 nil

Rule IContain-*REC13
 There are no type variables anywhere and d1, d3, d5, and d7
 allow no *cons forms, i.e., they are primitive descriptors or
 disjunctions of primitive descriptors,

 (icontained-in
   (*rec foo (*or d1 (*cons d2 (*or d3 (*cons d4 (*recur foo))))))
   (*rec bar (*or d5 (*cons d6 (*or d7 (*cons d8 (*recur bar))))))
   vref)
 =
 (and (contained-in d1 d5)
      (contained-in d2 d6)
      (contained-in d3 d7)
      (contained-in d4 d8))

Rule IContain-*REC0 is a default, to be employed to administer failure
then the preconditions for all the other rules have not been satisfied.
The proof is trivial.

Rule IContain-*REC0
 (icontained-in foo bar vref) =  nil
```

Now for the soundness proofs of some of the rules above. Each rule being proved will be followed by statement of the lemma corresponding to the rule, and then the lemma will be proved. Lemmas following the same pattern could be stated for the other rules and proved using similar techniques.

Recall that the main theorem we are trying to prove is the following.

```
Lemma ICONTAINED-IN-OK:

 For any descriptors td1 and td2, simple mapping m, Lisp value v,
 symbolic value reference vref, and binding b1 covering the
 variables in td1,

  (and
H1  (disjoint (gather-variables-in-descriptor td1)
              (gather-variables-in-descriptor td2))
H2  (well-formed-mapping m vref b1)
```

```
H3  (icontained-in td1 (dapply-subst-list-1 m td2) vref)
H4  (I td1 v b1) )
  =>
  For some b2, (I td2 v b2)
```

Note that for each of our rules, the descriptor in the second argument to ICONTAINED-IN is *not* the TD2 in this theorem, but rather (DAPPLY-SUBST-LIST-1 M TD2). So in each of the following, we will use a shorthand "bar-td2" to represent the *REC descriptor TD2, we will use "bar" to represent the second argument in the icontained-in rule, i.e., (DAPPLY-SUBST-LIST M bar-td2). We will also abbreviate DAPPLY-SUBST-LIST-1 with "A-S-1".

```
Rule IContain-*REC41
 (icontained-in (*rec foo .. )
                (*rec bar (*or vref (*cons .. (*recur bar) ..)))
                vref)
 = t
```

For notational convenience, let

```
 "foo" denote (*rec foo .. )
 "bar" denote (*rec bar (*or vref (*cons .. (*recur bar) ..)))
 "bar-td2" denote (*rec bar (*or &i (*cons .. (*recur bar) ..)))
```

The lemma corresponding to this rule is:

```
Lemma ICONTAIN-*REC41-OK
 For any descriptors foo, bar, and bar-td2 of the form prescribed,
 and for any simply mapping m, Lisp value v, symbolic value reference
 vref, and binding b1 covering the variables in foo,

  (and
H1  (disjoint (gather-variables-in-descriptor foo)
              (gather-variables-in-descriptor bar-td2))
H2  (well-formed-mapping m vref b1)
H3  (equal bar (A-S-1 m bar-td2))
H4  (icontained-in foo (A-S-1 m bar-td2) vref)
H5  (I foo v b1))
  =>
  (I bar-td2 v (make-binding-from-mapping m v b1 vref))

Proof of Lemma ICONTAIN-*REC41-OK

 From H3, we know m maps type variable &i to vref.
 I.e., we know that bar-td2 is of the form:

  (*rec bar (*or &i (*cons .. (*recur bar) ..)))

 By H2 and the definition of make-binding-from-mapping,
 (make-binding-from-mapping m v b1 vref) maps &i to v.

 (I (*rec bar (*or &i (*cons .. (*recur bar) ..)))
    v
    (make-binding-from-mapping m v b1 vref))
 =   by the canonicalization function open-rec-descriptor-absolute
 (I (*or &i
         (*cons ..
                (*rec bar (*or &i (*cons .. (*recur bar) ..)))
                ..))
    v
    (make-binding-from-mapping m v b1 vref))
 =   by the definition of I
 (or (I &i v (make-binding-from-mapping m v b1 vref))
     (I (*cons ... ) v (make-binding-from-mapping m v b1 vref)))
 =   since (make-binding-from-mapping m v b1 vref) maps &i to v
```

```
 (or (equal v v) .. )
 = t

QED


Rule IContain-*REC43
 Where d3 is a primitive descriptor or a disjunction of primitive
 descriptors,

 (icontained-in
   (*rec foo (*or d3 (*cons d4 (*recur foo))))
   (*rec bar (*or (rec-tail vref) (*cons d2 (*recur bar))))
   vref)
 =
 (contained-in d4 d2)
```

For notational convenience, let

```
 "foo" denote (*rec foo (*or d3 (*cons d4 (*recur foo))))
 "bar" denote (*rec bar (*or (rec-tail vref)
                             (*cons d2 (*recur bar))))
 "bar-td2" denote (*rec bar (*or &i (*cons d2 (*recur bar))))
```

The lemma corresponding to this rule is:

```
Lemma ICONTAIN-*REC43-OK
 For any descriptors foo, bar, and bar-td2 of the form prescribed,
 and for any simple mapping m, Lisp value v, symbolic value reference
 vref, and binding b1 covering the variables in foo,

  (and
H1  (disjoint (gather-variables-in-descriptor foo)
              (gather-variables-in-descriptor bar-td2))
H2  (well-formed-mapping m vref b1)
H3  (equal bar (A-S-1 m bar-td2))
H4  (contained-in d4 d2)
H5  (I foo v b1))
  =>
C (I bar-td2 v (make-binding-from-mapping m v b1 vref))

Proof of Lemma ICONTAIN-*REC43-OK

 From H3 we know m maps type variable &i to the form
 (rec-tail vref) (i.e., the form whose car is the atom rec-tail
 and whose cadr is the symbolic value reference vref).
 From H2 and the definition of make-binding-from-mapping,
 we know the binding from (make-binding-from-mapping m v b1 vref)
 maps &i to the value (rec-tail v).  H5 expands by definition of
 I to
  (or (I d3 v b1) (I (*cons d4 foo) v b1)),
 which in turn expands to:

H5' (or (I d3 v b1)
        (and (consp v)
             (I d4 (car v) b1)
             (I foo (cdr v) b1)))

 The conclusion, (I bar-td2 v (make-binding-from-mapping m v b1 vref)),
 expands to

C' (or (I &i v (make-binding-from-mapping m v b1 vref))
       (and (consp v)
            (I d2 (car v) (make-binding-from-mapping m v b1 vref))
            (I bar-td2 (cdr v) (make-binding-from-mapping m v b1 vref))))

 Now consider the problem by cases, as suggested in H5'.
```

```
 Case 1  (I d3 v b1)

  A condition on the rule is that d3 is a primitive (non-cons)
  descriptor or a disjunction of primitive descriptors.  Therefore,
  (I d3 v b1) can only be true if the value v is an atomic value.
  This limits our options in the conclusion to establishing
  (I &i v (make-binding-from-mapping m v b1 vref)).  No problem.
  As previously stated, (make-binding-from-mapping m v b1 vref)
  maps &i to (rec-tail v).  By the definition of rec-tail, if
  v is an atom, (rec-tail v) = v.  So by the definition of I,
  (I &i v (make-binding-from-mapping m v b1 vref)) = (equal v v),
  since (make-binding-from-mapping m v b1 vref) maps &i to
  the value (rec-tail v).

 Case 2  (and (consp v)
              (I d4 (car v) b1)
              (I foo (cdr v) b1))

  It suffices to establish

C1   (and (consp v)
C2        (I d2 (car v) (make-binding-from-mapping m v b1 vref))
C3        (I bar-td2 (cdr v) (make-binding-from-mapping m v b1 vref)))

  C1 follows directly from the case assumption.
  C2 follows immediately from Lemma CONTAINED-IN-OK:

Lemma CONTAINED-IN-OK

 For any descriptors td1 and td2, Lisp value v, and binding b1,

  (and
H1  (null (gather-variables-in-descriptor td1))
H2  (null (gather-variables-in-descriptor td2))
H3  (contained-in td1 td2)
H4  (I td1 v b1))
  =>
  For some b2, (I td2 v b2)

Comment: The b2 used here is irrelevant, since the descriptors are
variable-free.


  We instantiate this lemma with td1 = d4, td2 = d2, v = (car v),
  and b1 = b1.  The well-formedness rule for *rec descriptors
  establishes that d2 and d4 have no variables, since both d2 and
  d4 appear in replicating components of their respective *rec
  descriptors.  H4 equals the third antecedent.  The fourth and
  final antecedent is established by our case assumption.  So we
  can use the conclusion.  As always, in the absence of variables,
  the bindings are irrelevant to the evaluation of I, so b2 may as
  well be the binding (make-binding-from-mapping m v b1 vref).  This
  establishes C2.

  To establish C3, we use ICONTAIN-*REC43-OK, as an inductive
  assertion on the CDRs, instantiating with m = m, vref = vref,
  b1 = b1, and v = (cdr v).  Our case assumption (I foo (cdr v) b1))
  matches H5 of the inductive assertion.  The other antecedents
  match hypotheses in our main goal.  Therefore, we can use the
  conclusion:

  (I bar-td2 (cdr v) (make-binding-from-mapping m (cdr v) b1 vref))

  Our final subgoal,

  C3      (I bar-td2 (cdr v) (make-binding-from-mapping m v b1 vref)))

  is directly implied by the conclusion of the inductive hypothesis.
  The binding (make-binding-from-mapping m v b1 vref) is equal
```

```
  to (make-binding-from-mapping m (cdr v) b1 vref), since
  (rec-tail (cdr v)) = (rec-tail v) under the (consp v) assumption.
  In either case it is the final atomic tail of the argument.
  QED.
```

```
Rule IContain-*REC44
 Where d1 and d2 contain no variables,
 (icontained-in (*rec foo (*or &i (*cons d1 (*recur foo))))
                (*rec bar (*or &i (*cons d2 (*recur bar))))
                vref)
 =
 (contained-in d1 d2)
```

For notational convenience, let

```
 "foo" denote (*rec foo (*or &i (*cons d1 (*recur foo))))
 "bar" denote (*rec bar (*or &i (*cons d2 (*recur bar))))
 "bar-td2" denote (*rec bar (*or &k (*cons d2 (*recur bar))))
```

The lemma corresponding to this rule is:

```
Lemma ICONTAIN-*REC44-OK
 For any descriptors foo, bar, and bar-td2 of the form prescribed,
 and for any simple mapping m, Lisp value v, symbolic value reference
 vref, and binding b1 covering the variables in foo,

  (and
H1  (disjoint (gather-variables-in-descriptor foo)
              (gather-variables-in-descriptor bar-td2))
H2  (well-formed-mapping m vref b1)
H3  (equal bar (A-S-1 m bar-td2))
H4  (contained-in d1 d2)
H5  (I foo v b1))
  =>
C (I bar-td2 v (make-binding-from-mapping m v b1 vref))

Proof of Lemma ICONTAIN-*REC44-OK
 Note that since H1 guarantees the disjointness of the variables in
 foo and bar-td2, the only way &i could appear in both foo and bar is
 if the mapping m had introduced it in place of some other variable
 not appearing in foo.  Let us denote this variable &k.  The
 mapping m, then, must contain (&k . &i).

 Now expand H5, giving

H5'  (or (I &i v b1)
         (and (consp v)
              (I d1 (car v) b1)
              (I foo (cdr v) b1)))

 and expand C, giving

C' (or (I &k v (make-binding-from-mapping m v b1 vref))
       (and (consp v)
            (I d2 (car v) (make-binding-from-mapping m v b1 vref))
            (I bar-td2 (cdr v) (make-binding-from-mapping m v b1 vref))))

 Split into cases, according to H5'.

 Case 1  (I &i v b1)

  By the definition of I, (I &i v b1) expands to
  (equal v (cdr (assoc &i b1))).

  Now consider (make-binding-from-mapping m v b1 vref).  By definition
  of  make-binding-from-mapping, when an entry (&k . &i) appears in
```

```
  the mapping m, the binding for &k in the result is the same as
  the binding of &i in b1.  The (well-formed-mapping m vref b1)
  predicate in H2 guarantees that &i is bound in b1.  So with this
  in mind, we see that (I &k v (make-binding-from-mapping m v b1 vref))
  expands by the definition of I to (equal v (cdr (assoc &i b1))),
  which we established above.

 Case 2  (and (consp v)
              (I d1 (car v) b1)
              (I foo (cdr v) b1))

  We will attempt to establish the second disjunct of C1':

  C1   (and (consp v)
  C2        (I d2 (car v) (make-binding-from-mapping m v b1 vref))
  C3        (I bar-td2 (cdr v) (make-binding-from-mapping m v b1 vref)))

  C1 appears in the case assumption.
  C2 follows immediately from Lemma CONTAINED-IN-OK, instantiated
  with td1 = d1, td2 = d2, v = (car v), and b1 = b1.  The
  well-formedness property for *rec descriptors establishes that
  neither d1 nor d2 contain any variables, since they appear within
  a replicating component of the *rec form.  Thus the variable-free
  hypotheses are satisfied.  H4 is equal to the third antecedent
  of CONTAINED-IN-OK.  Our case assumption (I d1 (car v) b1)
  establishes its final antecedent.  So we can use the conclusion.
  As always, in the absence of variables, the bindings are
  irrelevant to the evaluation of I, so b2 may as well be
  (make-binding-from-mapping m v b1 vref).  This establishes C2.

  To establish C3, we use our lemma as an inductive assumption on the
  CDRs, instantiating with m = m, b1 = b1, v = (cdr v), and
  vref = the form (cdr vref).

  H2 implies H2 of our inductive assertion, since
  well-formed-mapping only uses the root of the second argument, and
  (root-of-var-ref vref) = (root-of-var-ref (cdr vref)).
  Our case assumption (I foo (cdr v) b1)) matches H5 of the inductive
  assertion.  The other antecedents match hypotheses in our main goal.
  Therefore, we can use the conclusion:

  (I bar-td2 (cdr v) (make-binding-from-mapping m (cdr v) b1 (cdr vref)))

  Since bar-td2 has no symbolic value references and its only variable
  is &k, the only point at which the binding enters in the evaluation
  of these two calls of I is with respect to &k, and from what we
  know of m and from the definition of make-binding-from-mapping,
  we see that &k maps to the binding of &i in both
  (make-binding-from-mapping m (cdr v) b1 vref) and
  (make-binding-from-mapping m v b1 vref).  Hence, C3 follows from
  the conclusion of the inductive hypothesis
 QED.
```

B.10 Proof of Lemma UNIVERSALIZE-SINGLETON-VARS-1-OK

Lemma UNIVERSALIZE-SINGLETON-VARS-1-OK supports the proof of Lemma ICONTAINED-IN-INTERFACE-OK in Section 7.8.

```
Lemma TC-UNIVERSALIZE-SINGLETON-VARS-1-OK

 For all descriptors td, values v, type variable bindings b, and
 lists vlist containing all the type variables which appear only
 once within td,

 (I (tc-universalize-singleton-vars-1 td vlist) v b)
 =>
```

```
 For some b' (I td v b')
```

Let us suppose that TC-UNIVERSALIZE-SINGLETON-VARS-1 is defined as follows:

```
(DEFUN TC-UNIVERSALIZE-SINGLETON-VARS-1 (TD VLIST)
  (IF (NULL VLIST)
      NIL
      (SUBST '*UNIVERSAL
             (CAR VLIST)
             (TC-UNIVERSALIZE-SINGLETON-VARS-1 TD (CDR VLIST)))))
```

This is not quite the style in which TC-UNIVERSALIZE-SINGLETON-VARS is coded, but it is functionally equivalent. (Actually in the implementation, TC-UNIVERSALIZE-SINGLETON-VARS gathers a list of all variables in td, with each instance represented, and then calls TC-UNIVERSALIZE-SINGLETON-VARS-1 with this list. TC-UNIVERSALIZE-SINGLETON-VARS-1 substitutes *UNIVERSAL for each variable in TD which occurs exactly once in the list.)

```
Proof of Lemma TC-UNIVERSALIZE-SINGLETON-VARS-1-OK

The proof is by induction on vlist.

Base case: vlist = nil

 By definition,
  (tc-universalize-singleton-vars-1 td vlist) = td
 establishing the goal trivially, with b' = b.

Inductive case: vlist = (cons &i vlist')

 We assume our lemma inductively, establishing

 (I (tc-universalize-singleton-vars-1 td (cdr vlist)) v b)
 =>
 For some b'' (I td v b'')

 Thus, our goal is

 (I (subst *universal
           &i
           (tc-universalize-singleton-vars-1 td (cdr vlist)))
    v b)
 =>
 For some b' (I td v b')

 Consider the following functions, which will serve as
 witnesses.

 (DEFUN FIND-RIGHT-SUBSTS (VAR V TD)
   (IF (EQUAL VAR TD)
       (LIST V)
       (IF (ATOM TD)
           NIL
           (IF (OR-DESCRIPTORP TD)
               (FIND-ALL-RIGHT-SUBSTS VAR V (CDR TD))
               (IF (REC-DESCRIPTORP TD)
                   (FIND-RIGHT-SUBSTS
                     VAR V (OPEN-REC-DESCRIPTOR-ABSOLUTE TD))
                   ;; td is a *cons
                   (IF (ATOM V)
                       NIL
                       (APPEND (FIND-RIGHT-SUBSTS
                                 VAR (CAR V) (CADR TD))
                               (FIND-RIGHT-SUBSTS
                                 VAR (CDR V) (CADDR TD)))))))))
```

```
(DEFUN FIND-ALL-RIGHT-SUBSTS (VAR V TD-LIST)
  (IF (NULL TD-LIST)
      NIL
      (APPEND (FIND-RIGHT-SUBSTS VAR V (CAR TD-LIST))
              (FIND-ALL-RIGHT-SUBSTS VAR V (CDR TD-LIST)))))
```

Given a variable VAR, a Lisp value V, and a descriptor TD, FIND-RIGHT-SUBSTS returns all the possible bindings for VAR under which V could satisfy TD. Since we do not allow variables in a replicating component of a *REC, the variable will never be required to simultaneously be bound to more than one of these bindings. Furthermore, FIND-RIGHT-SUBSTS terminates because the size of V decreases when it recurs with a *CONS, and there is always a *CONS along any path (in the sense of recursive descent) through the body of a *REC descriptor to any *RECUR within the body.

The binding we need to demonstrate our conclusion is simply any b'' which satisfied our inductive hypothesis, modified to map &i to the correct value. Form a collection of bindings by extending b'' with the pair (&i . v_j) for each v_j returned by (find-right-substs &i v td). By "extending" we mean either adding the pair to b'' if there is no binding for &i in b'', or by replacing the binding for &i in b'' with v_j. There is no danger in replacing a binding for &i, because it will be replaced in one of our new bindings by every possible correct binding for &i. Thus, if a binding for &i is correct in b'', in one of our extended bindings, it will be preserved.

So wherever some component of v was required to satisfy the instance of *UNIVERSAL which replaced &i in the descriptor in our hypothesis, for one of our bindings b', constructed as above, v will satisfy &i. QED.

Appendix C
A Set-Based Semantics

In Chapter 5, we mentioned that several alternate semantic models were considered before we settled on the INTERP-SIMPLE semantics. This appendix describes one of those models, one in which type descriptor variables can be instantiated by other descriptors, rather than by singleton values. We abandoned this model because of a desire for a more computational style of semantics. As this was a work in progress when abandoned, it is no doubt imperfect.

This appendix consists of notes originating from a discussion with Matt Kaufmann about semantics, which were extended and elaborated as the ideas were developed. Some of the discussion reflects that the ideas were in flux. Kaufmann provided the first draft of the definition of Eval and the statement of the theorems proved.

Let TD be the set of type descriptors, and let CLU be the Common Lisp universe of objects. Let TA be the set of type assignments, i.e. functions mapping the set of type variables to the power set of CLU, P(CLU). For the purpose of communication, we may sometimes refer to TA as mapping from type variables to variable-free descriptors, as any variable-free descriptor is an exact representation of a set of Lisp values.

We define a function "Eval", to be a mapping from a type descriptor and a type assignment to a set of Common Lisp values.

```
Eval: TD x TA -> P(CLU)
```

defined by recursion as follows.

```
Eval(*empty,a) = empty set
Eval(*universal,a) = P(CLU)
Eval(&n,a) = a(&n) for &n a type variable.
Eval(td,a) = "the obvious set" if td is a basic type descriptor like
             *integer, *nil, etc.
Eval((*AND td1 td2),a) = Eval(td1,a) intersect Eval(td2,a)
 {This *AND descriptor is not the one implemented, which is only
  transitory and has a special relationship to partially defined
  recursive descriptors.  This *AND was introduced here because
  a possible extension to the descriptor language and the inference
  algorithm might make it possible to deal more accurately with
  variables in recursive forms.}
Eval((*OR td1 td2),a) = Eval(td1,a) union Eval(td2,a)
Eval(*CONS td1 td2,a) =
    {(cons v1 v2)|(member v1 Eval(td1,a)) and (member v2 Eval(td2,a))}
```

Some discussion leads up to the definition of Eval((*REC <rec-name> td),a). Consider an example of a recursive descriptor:

```
T = (*rec tl (*or $nil (*cons *universal (*recur tl))))
```

Then consider the descriptor T_o, derived from T by replacing the recursion point with a fresh variable, which we will call &*.

```
To = (*or $nil (*cons *universal &*))
```

We define the semantics of T as

```
Eval(T,a) = ∪^n P_a^n(phi)
```

i.e., the union over all naturals n of the function P_a^n with the initial argument of the empty set, where P_a is defined

```
P_a(s) = Eval(T_o,a ++ {<&*,s>})    ("++" is an appending operator)
```

The expansions of this function are as follows (where "x" is a Cartesian product employing the CONS constructor):

```
The base set is ∅.

P_a(∅) = Eval (T_o, a ++ {<&*,∅>})
        = (*or $nil (*cons *universal *empty)) = {nil}
so P(∅) = ∅ ∪ {nil} = {nil}

P_a({nil}) = Eval (T_o, a ++ {<&*,{nil}>})
          = {nil} ∪ (Universe x {nil})
so P(P(∅)) = {nil} ∪ (Universe x {nil}).
  Call the above set S1.
P_a(S1) = Eval (T_o, a ++ {<&*,S1>})
  = {nil} ∪ (Universe x S1)
  = {nil} ∪ (Universe x {nil}) ∪ (Universe x (Universe x {nil}))
which is P(P(P(∅)))

etc.
```

Notice that $\emptyset \subseteq P(\emptyset) \subseteq P(P(\emptyset))$ ...

Another notation for the big union "$\cup^n P_a^n(\emptyset)$" is the form

```
Union{P_a^n(∅): n ∈ w}
```

Here, "w" is omega, the set of all natural numbers, the "∈" is "member", so the set qualifier is a quantification over all natural numbers, with n being the quantified variable.

So in general, when we have a recursive descriptor

```
T = (*rec <recname> td)
```

we have

```
T_o = td/((*recur <recname>) . &*)
```

and we define the semantics of T as

```
Eval(T,a) = ∪^n P_a^n(∅)
```

where

```
P_a(s) = Eval(T_o,a ++ {<&*,s>})
```

A proof of the following rewrite rule would have been crucial to the proof of unification. I did not get around to this proof before abandoning this semantic model. With respect to the preceding fixed point semantics, we would have needed to prove the equivalence:

```
Eval((*REC <rec-name> td),a) =
```

```
    Eval(td((*rec <rec-name> td)/(*recur <rec-name>)),a)
```

Next, we will need a notion of descriptor substitution. A substitution is simply a function from a set of type variables into TD. The notion of application of substitution, written td/s (for td in TD and s a substitution) has a simple recursive definition, defined as follows. Note that we can view the identity substitution s_id as the empty function.

```
Definition of "/": (descriptor, substs) -> descriptor

 &i/s = s(&i)
 prim/s = prim, where "prim" is one of
   {$CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
              $STRING $T}
 *universal/s = *universal
 *empty/s = *empty
 (*cons t1 t2)/s = (*cons tl/s t2/s)
 (*or t1 t2)/s = (*or tl/s t2/s)
 (*and t1 t2)/s = (*and tl/s t2/s)
 (*rec <recname> td)/s = (*rec <recname> td/s)
```

We can define an ordering on substitutions as follows:

```
  s1 <= s2 iff
  for all a in TA and all type variables &n,
    Eval(s1(&n),a) ⊆ Eval(s2(&n),a).
```

Using these semantics and definitions we can derive a number of properties.

C.1 Monotonicity

```
If s1 <= s2, then for all td in TD and all a in TA,
  Eval(td/s1,a) ⊆ Eval(td/s2,a).

Proof:
Case 1  td = &i
 td/s1 = si(&i)    td/s2 = s2(&i)
 Eval (s1(&i), a) ⊆ Eval (s2(&i), a)      def of <=

Case 2  td simple, *empty, or *universal
 td/s1 = td/s2 = td   Eval(td,a) = Eval(td,a)

Case 3  td = (*cons td1 td2)
  Goal: all a in TA,
          Eval((*cons td1 td2)/s1,a) ⊆ Eval((*cons td1 td2)/s2,a)
  By definition of /:
  Eval((*cons td1/s1 td2/s1),a) ⊆ Eval((*cons td1/s2 td2/s2),a)
  By def of Eval:
  {(*cons v1 v2)|v1 ∈ Eval(td1/s1,a) and v2 ∈ Eval(td2/s1,a)}
     ⊆
  {(*cons v3 v4)|v3 ∈ Eval(td1/s2,a) and v4 ∈ Eval(td2/s2,a)}
  With 1) and 2) as base cases and the induction hypothesis:
    Eval(td/s1,a) ⊆ Eval(td/s2,a).
  Note that A ⊆ C and B ⊆ D
   => {(cons a b)| a ∈ A and b ∈ B}
       ⊆
      {(cons c d)| c ∈ C and d ∈ D}
  where A = Eval(td1/s1,a), B = Eval(td2/s1,a),
        C = Eval(td1/s2,a), and D = Eval(td2/s2,a)
  and we are done.

Case 4  td = (*or td1 td2)
```

```
  Goal: all a in TA,
          Eval((*or td1 td2)/s1,a) ⊆ Eval((*or td1 td2)/s2,a)
  By def of /:
  Eval((*or td1/s1 td2/s1),a) ⊆ Eval((*or td1/s2 td2/s2),a)
  By def of Eval:
    Eval(td1/s1,a) ∪ Eval(td2/s1,a)
       ⊆
    Eval(td1/s2,a) ∪ Eval(td2/s2,a)
  With 1) and 2) as base cases and the induction hypothesis:
    Eval(td/s1,a) ⊆ Eval(td/s2,a)
  we can assume Eval(td1/s1,a) ⊆ Eval(td1/s2,a)
          and   Eval(td2/s1,a) ⊆ Eval(td2/s2,a)
  and applying the general rule
     A ⊆ C and B ⊆ D => A ∪ B ⊆ C ∪ D
  we are done.

Case 5  td = (*and td1 td2)
  Same proof as *or, but with intersection instead of union.

Case 6  td = (*rec <rec-name> td)
  Prove: all a in TA,
    Eval((*rec <rec-name> td)/s1, a)
       ⊆
    Eval((*rec <rec-name> td)/s2, a)
  By def of /, this reduces to
    Eval((*rec <rec-name> td), a) ⊆ Eval((*rec <rec-name> td), a)
  which is trivially true.
QED.
```

C.2 Maximality of the *UNIVERSAL Substitution

```
Let su be a substitution mapping every type variable in its domain
to *universal.  Then for every substitution s with domain contained in
the domain of su, we have s <= su.  In particular, s_id <= su.

Proof:  Easy.
```

C.3 Elimination of Variables

```
Let su be a substitution mapping every type variable in its
domain to *universal, and suppose

 f |= (td1/su, td2/su,..., tdn/su) -> td

Then f |= (td1, td2, ..., tdn) -> td.

Proof:  By definition of "|=", we need to show that for all a in TA
and all (x1,...,xn) such that xi is a member of Eval(tdi,a) for
1<=i<=n, then f(x1,...,xn) is a member of Eval(td,a).  So, fix a in TA
and (x1,...,xn) such that xi is a member of Eval(tdi,a) for 1<=i<=n;
we need to show that f(x1,...,xn) is a member of Eval(td,a).
By Properties 1 and 2, we have that Eval(tdi,a) ⊆ Eval(tdi/su,a)
for 1<=i<=n.  It follows that xi ∈ Eval(tdi/su,a) for 1<=i<=n.
Now using the hypothesis that f |= (td1/su, td2/su, ..., tdn/su) -> td,
we have (by definition of "|=") that f(x1,...,xn) belongs to Eval(td,a),
which is what we needed to show.
```

C.4 Instantiation

If f |= (td1, td2, ..., tdn) -> td, and if s is a substitution,
then f |= (td1/s, td2/s, ..., tdn/s) -> td/s.

Proof:
Expand both the hypothesis and the conclusion using the definition of |=

```
all in TA, all (x1, .., xn) | xi in Eval(tdi,a) for 1 <= i <= n,
  f(x1, .., xn) in Eval(td,a)
=>
all a in TA, all (x1, .., xn) | xi in Eval(tdi/s,a) for 1 <= i <= n,
  f(x1, .., xn) in Eval(td/s,a)
```

Since the a in the conclusion is arbitrary, fix it and call it "ac". Now recall that a substitution is just a function from a set of type variables into TD, and that a type assignment is just a function from type variables into CLU. These functions can be composed. In particular, consider the composition of the function corresponding with s and the function from ac. Now choose the "a" for the hypothesis to be the assignment, or function, which produces the same result as s o ac, and call this assignment "ah". I.e., for every variable v in the domain of s, ah maps v to the value set produced by s(v) when all the variables in s(v) are mapped through ac, and for every variable v not in the domain of s, ah maps v the same as ac maps it.

So now consider:

Lemma L1: Eval(t,ah) = Eval(t/s,ac).

Using the lemma, we can equate the hypothesis and conclusion of our theorem. QED.

The proof of the lemma will use another lemma:

Lemma L2: If t is variable-free, Eval(t,a) = Eval(t,b).

Proof: Trivial, since Eval touches the context only when a variable is encountered in t.

Lemma L1: Eval(t,(s o ac)) = Eval(t/s,ac) where s is a substitution and ac a type assignment.

Proof of L1:
Let ah = s o ac. So the problem is to prove Eval(t,ah) = Eval(t/s,ac).
Do this by induction on the structure of type descriptors.

1. For t primitive, *empty, or *universal, the result is trivial.
2. For variables, it follows directly from the definiton of ah.
3. For t = (*cons t1 t2), we use 1) and 2) as the base case and for the induction step assume the proposition is true for t = t1 and t = t2. Then, by the definition of Eval, we have {(cons v1 v2) | v1 in Eval(t1,ah) and v2 in Eval(t2,ah)} = {(cons v3 v4) | v3 in Eval(t1/s,ac) and v4 in Eval(t2/s,ac)} and the induction hypothesis carries us through.
4. *and, as with *cons
5. *or, as with *cons
6. For t = (*rec <rname> td), From the definition of /, we have t = t/s. So our subgoal is: Eval(t,ah) = Eval(t,ac). Since recursive descriptors are variable-free, we invoke lemma L2 and we are done.

QED

C.5 The Semantics of Function Signatures

Given this semantics for type descriptors, we can now study the semantics of the function type signatures generated by the inference system.

Imagine that for each function F in a state S, S assigns a set to F which we'll call its *domain*. (We could say that this is the set recognized by the guard of F.) In some sense all functions are total, so this is not quite the standard notion of domain. But the semantic interpretation of a function applied to arguments not satisfying the guard is extremely weak. Our type inference system can certainly only generate a meaningful signature mapping arguments within the domain. So we will not consider mapping arguments outside of the domain.

We can consider a function signature to be a set of *segments*, where a segment is a syntactic entity of the form (TD1, TD2, ..., TDN) -> TD, where N is the arity of the function. Intuititively, such a segment says that if the ith parameter of the function is in the set prescribed by TDi, then the value of the function application *may be* in the set TD. We could say the specification for our inference algorithm TC, applied to a function F in a state S (where the state contains the signatures of previously defined functions) is:

```
TC(f,s) == {(td1...tdn) -> td | f |= (td1...tdn) -> td }
```

By "F |= (TD1...TDN) -> TD" we mean that "(TD1...TDN) -> TD" is a valid segment for the function F. By this we mean the following.

```
If f is a function of arity n then we might write the following
definition of |= :

f |= ((td1 ... tdn) -> td) ==
  for all a in TA,
    for all (x1 ... xn) such that xi in Eval(tdi,a) for 1<=i<=n,
      f(x1 ... xn) in Eval(td,a)
```

But perhaps this definition of |= is would not provide the most worthwhile signatures for our Lisp functions. Consider the function:

```
(DEFUN FOO (X)
  (IF (EQUAL X 3)
      10
      (IF (EQUAL X 5)
          'FOO
          T)))
```

Under the preceding definition of |=, we might wish for the signature:

```
($INTEGER) -> (*OR $INTEGER $NON-T-NIL-SYMBOL $T)
($CHARACTER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING $T)
 -> $T
```

But producing a signature like this goes against the grain of an inference algorithm which will analyze the function in Lisp's recursive descent evaluation order. Ultimately, the results returned by the function will be computed at the tips of its IF structure. At each tip we will have a context of the types of the formal variables, as determined by the various IF tests through which they have passed. This suggests that the orientation of the algorithm would be around generating a segment or a collection of segments at these tips. Since there can be tests like (EQUAL X 3), an $INTEGER parameter could factor into more than one of these tips. But the preceding definition of |= determines that each segment will have on its left hand side a collection of type descriptors totally distinct from any other segment. To derive such a collection would seem to require a complete inversion of the segments produced via recursive analysis. Such an inversion might be possible, but it would be tricky to preserve information about variables, as the scope of variables would then become the entire signature rather than being limited to one segment.

A signature for FOO more in harmony with recursive descent analysis would be:

```
($INTEGER) -> $INTEGER
($INTEGER) -> $NON-T-NIL-SYMBOL
(*UNIVERSAL) -> $T
```

One segment is produced for each tip of the IF tree. But clearly, we have a type assignment, i.e., the empty one, for which if X1 = 3, the second and the third segments do not conform to |=. So this style of signature is inconsistent with our definition of |=. We will need a definition of signatures wherein individual segments are neither wholly independent nor absolutely applicable. For any actual argument types, there may be more than one segment whose left hand side is satisfied by the arguments. So let us try to find a definition consistent with this alternate signature for FOO.

One of the pieces of information returned by the inference algorithm is a descriptor list characterizing the function guard. We can imagine two important components of our definition. One ensures that for any arguments which satisfy the function guard, at least one segment will be applicable, i.e., will have a left hand side which is also satisfied by the type of the arguments. Stated formally, this requirement is as follows:

```
for all (x1...xn) such that there exists a substitution a in TA such that
    for all i such that 1<=i<=n, xi in Eval(guardi,a)
      where (guard1..guardn) = the type descriptor list characterizing
                                  the function guard
  there exists a segment (td1..tdn) -> td in TC(f,s) such that
    there exists a substitution a in TA such that
      for all i such that 1<=i<=n, xi in Eval(tdi,a)
```

We can paraphrase the first part of this requirement states by saying: "for all (X1...XN) such that F(X1...XN) is defined (and N is the arity of F)". The second part says that for any arguments satisfying the function guard, there is some segment whose left hand side is satisfied by the types of the arguments.

Now we need to characterize the fact that one of the segments will yield the correct type for the arguments (X1...XN). There are several ways to phrase this requirement, all equivalent and each revealing a slightly different view of a signature and how it might be used. The most straightforward simply states, in our formal setting, that one of the segments applies.

```
  where S = ((td11...tdn1) -> td1) .. ((td1m...tdnm) -> tdm) is the
    collection of segments in TC(f,s) such that for all j, 1<=j<=m
      there exists a substitution a in TA such that
        for all i such that 1<=i<=n, xi in Eval(tdij,a)
   for some segment ((td1...tdn) -> td) in S,
     for some substitution a in TA such that
       for all i such that 1<=i<=n, xi in Eval(tdi,a),
         f(x1..xn) in Eval(td,a)
```

S is the set of segments whose left hand sides are satisfied by the (X1...XN) under some substitution for type variables. The statement is, then, that one of the segments, given the right substitution, supplies the correct result type.

The next statement, rather than stressing the selection of a single correct segment, focuses on an aggregate result which is also correct under some substitution. This is the interpretation which, we will see, suggests the principal rule used in formulating closed recursive forms for the results of recursive functions.

```
where S = ((td11...tdn1) -> td1) .. ((td1m...tdnm) -> tdm) is the
  collection of segments in TC(f,s) such that for all j, 1<=j<=m
    there exists a substitution a in TA such that
      for all i such that 1<=i<=n, xi in Eval(tdij,a)
 and where A is the collection of all such substitutions,
     for some a in A, f(x1..xn) in Eval((*or td1..tdm),a)
```

The following alternate statement is much like the previous one and perhaps more directly grounds in the notion of sets.

```
where S = ((td11...tdn1) -> td1) .. ((td1m...tdnm) -> tdm) is the
  collection of segments in TC(f,s) such that for all j, 1<=j<=m
    there exists a substitution a in TA such that
      for all i such that 1<=i<=n, xi in Eval(tdij,a)
 and where (((td11...tdn1) -> td1),a1) .. (((td1m...tdnm) -> tdm),am)
   is the collection of all such segments and their associated
     substitutions,
   f(x1..xn) in Union (Eval(td1,a1) .. Eval(tdm,am))
```

Appendix D
A Function-Based Semantics

In Chapter 5, we mentioned that several alternate semantic models were considered before we settled on the INTERP-SIMPLE semantics. This appendix describes one of those models, where the semantics is given in the vernacular of functions and meta-functions. Each descriptor corresponds to a function, and a descriptor containing type variables corresponds to a function with one parameter for each type variable. These parameters are to be instantiated with functions corresponding to the type descriptors instantiating the variables. This direction was abandoned because its meta-theoretic approach posed more potential problems than we would encounter with a more straightforwardly computational model. In particular, the semantic definition itself generates function definitions, a notion which could become problematic. As this was a work in progress when abandoned, it is no doubt imperfect.

The notion of well-formed descriptor is the precisely same in this semantic model as it is in the model finally adopted for the system. While working on this model, we also had a descriptor construction (*NOT <descriptor>), which represented the negation of the enclosed descriptor.

For any well-formed descriptor, there is an associated Lisp recognizer meta-function. A recognizer meta-function is very much like a recognizer function, except that, rather than having arity one, its arity is one plus the number of distinct variables which appear in the descriptor. The actuals supplied for these variables are the names of recognizer meta-functions which have exactly one argument, i.e., recognizers corresponding to variable-free type descriptors.

So let us define a generator function for these meta-functions. Given a descriptor, it generates a list of DEFUNs of meta-recognizer functions, with the first being the top level meta-recognizer for the entire descriptor. This generator provides the basis for a function-based semantics for the descriptor language. GEN-RECOGNIZERS takes a type descriptor as input and generates a list of meta-recognizer functions which, taken together, recognize objects conforming to the descriptor under the binding of type variables to variable-free meta-recognizers provided in the parameter list.

Here are several examples of calls to GEN-RECOGNIZERS, followed by the list of functions each call returns. Interleaved are examples of calls to the meta-recognizers on illustrative arguments and the results to those calls. A hideous thing about reading these results is the names of the functions generated, which are the string of characters composing the descriptor itself followed by "-RECOGNIZER". The names are ugly, but are guaranteed to be unique and replicable. The only exception to this convention, which is unnecessary but is a concession to readability, is the case of recursive meta-recognizers, which take on the name associated with the *REC form from which they were generated. This concession is safe under the requirement that identically named *REC descriptors must be identical.

When the descriptor is primitive, we generate a single function with the appropriate calls to primitive Lisp functions.

```
(GEN-RECOGNIZERS '$CHARACTER)

((DEFUN $CHARACTER-RECOGNIZER (THING)
   (CHARACTERP THING)))

($CHARACTER-RECOGNIZER #\a)
T
```

If the descriptor is an *OR, we generate an OR of the recognizers for the disjuncts.

```
(GEN-RECOGNIZERS '(*OR $INTEGER $NIL))

((DEFUN |(*OR $INTEGER $NIL)-RECOGNIZER| (THING)
     (OR (INTEGERP THING)
         (NULL THING))))
```

When it is a *CONS, we have a CONSP test plus appropriate tests applied to the CAR and CDR.

```
(GEN-RECOGNIZERS '(*CONS $INTEGER $NIL))

((DEFUN |(*CONS $INTEGER $NIL)-RECOGNIZER| (THING)
     (AND (CONSP THING)
          (INTEGERP (CAR THING))
          (NULL (CDR THING)))))

(|(*CONS $INTEGER $NIL)-RECOGNIZER| '(4))
T
```

When the descriptor is a variable, we generate a meta-recognizer with an extra formal parameter corresponding to the variable. The caller should supply a meta-recognizer function *of arity one* as the actual parameter for this formal. This function is then applied to the object occupying the position of the variable.

```
(GEN-RECOGNIZERS '&1)

((DEFUN &1-RECOGNIZER (THING VAR-&1)
   (APPLY VAR-&1 (LIST THING))))

(&1-RECOGNIZER '(4) '|(*CONS $INTEGER $NIL)-RECOGNIZER|)
T

(GEN-RECOGNIZERS '(*CONS &1 &2))

((DEFUN |(*CONS &1 &2)-RECOGNIZER| (THING VAR-&1 VAR-&2)
   (AND (CONSP THING)
        (APPLY VAR-&1 (LIST (CAR THING)))
        (APPLY VAR-&2 (LIST (CDR THING))))))

(|(*CONS &1 &2)-RECOGNIZER| '(#\a 4)
                            '$CHARACTER-RECOGNIZER
                            '|(*CONS $INTEGER $NIL)-RECOGNIZER|)
T

(|(*CONS &1 &2)-RECOGNIZER|
   '(1 . 2) '$INTEGER-RECOGNIZER '$NIL-RECOGNIZER)
NIL
```

When the descriptor is a *REC, we generate a recursive function where the recursive call is in the position corresponding to the *RECUR.

```
(GEN-RECOGNIZERS
  '(*REC INT-LISTP-1 (*OR $NIL (*CONS $INTEGER (*RECUR INT-LISTP-1)))))

((DEFUN INT-LISTP-1 (THING)
   (OR (NULL THING)
       (AND (CONSP THING)
            (INTEGERP (CAR THING))
            (INT-LISTP-1 (CDR THING))))))

(INT-LISTP-1 '(1 2 3))
T

(GEN-RECOGNIZERS
  '(*OR (*CONS *UNIVERSAL $T)
        (*REC TRUE-LISTP-1
```

```
              (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP-1))))))

((DEFUN |(*OR (*CONS *UNIVERSAL $T)
 (*REC TRUE-LISTP-1
 (*OR $NIL (*CONS *UNIVERSAL (*RECUR TRUE-LISTP-1)))))-RECOGNIZER|
        (THING)
   (OR (AND (CONSP THING)
            T
            (EQUAL (CDR THING) T))
       (TRUE-LISTP-1 THING)))
 (DEFUN TRUE-LISTP-1 (THING)
   (OR (NULL THING)
       (AND (CONSP THING)
            T
            (TRUE-LISTP-1 (CDR THING))))))
```

This example illustrates two things: first, that nested recursive descriptors are handled appropriately, with their variable parameters composed correctly, and second, that the variable parameter meta-recognizers are applied uniformly.

```
(GEN-RECOGNIZERS
  '(*CONS &2
          (*REC TRUE-LISTP-2
                (*OR (*REC TRUE-LISTP-3
                           (*OR $NIL (*CONS &2 (*RECUR TRUE-LISTP-3))))
                     (*CONS &1 (*RECUR TRUE-LISTP-2))))))

((DEFUN |(*CONS &2
 (*REC TRUE-LISTP-2
  (*OR (*REC TRUE-LISTP-3 (*OR $NIL (*CONS &2 (*RECUR TRUE-LISTP-3))))
   (*CONS &1 (*RECUR TRUE-LISTP-2)))))-RECOGNIZER| (THING VAR-&1 VAR-&2)
   (AND (CONSP THING)
        (APPLY VAR-&2 (LIST (CAR THING)))
        (TRUE-LISTP-2 (CDR THING) VAR-&1 VAR-&2)))
 (DEFUN TRUE-LISTP-2 (THING VAR-&1 VAR-&2)
   (OR (TRUE-LISTP-3 THING VAR-&2)
       (AND (CONSP THING)
            (APPLY VAR-&1 (LIST (CAR THING)))
            (TRUE-LISTP-2 (CDR THING) VAR-&1 VAR-&2))))
 (DEFUN TRUE-LISTP-3 (THING VAR-&2)
   (OR (NULL THING)
       (AND (CONSP THING)
            (APPLY VAR-&2 (LIST (CAR THING)))
            (TRUE-LISTP-3 (CDR THING) VAR-&2)))))

(|(*CONS &2
 (*REC TRUE-LISTP-2
  (*OR (*REC TRUE-LISTP-3 (*OR $NIL (*CONS &2 (*RECUR TRUE-LISTP-3))))
   (*CONS &1 (*RECUR TRUE-LISTP-2)))))-RECOGNIZER|
  '(FOO 1 2 3 FOO BAR)
  '$INTEGER-RECOGNIZER
  '$NON-T-NIL-SYMBOL-RECOGNIZER)
T

(|(*CONS &2
 (*REC TRUE-LISTP-2
  (*OR (*REC TRUE-LISTP-3 (*OR $NIL (*CONS &2 (*RECUR TRUE-LISTP-3))))
   (*CONS &1 (*RECUR TRUE-LISTP-2)))))-RECOGNIZER|
  '(FOO 1 2 3 BAR 4)
  '$INTEGER-RECOGNIZER
  '$NON-T-NIL-SYMBOL-RECOGNIZER)
NIL
```

Here are the functions defining the meta-recognizer generator. There is an assumption that the descriptors are well-formed.

```
(DEFUN GEN-RECOGNIZERS (DESCRIPTOR)
  ;; We generate the functions for all the *REC descriptors nested within
  ;; the body of DESCRIPTOR prior to generating the top level function
  ;; for DESCRIPTOR.  If DESCRIPTOR is a *REC form, we set up the state
  ;; a little differently for the call to GEN-RECOGNIZER-BODY.
  (LET* ((FNNAME (IF (REC-DESCRIPTORP DESCRIPTOR)
                     (CADR DESCRIPTOR)
                     (GEN-FNNAME DESCRIPTOR)))
         (VARS
           (SORT (FIND-VARS-IN-DESCRIPTOR DESCRIPTOR NIL) #'LEXORDER))
         (VAR-MAP (MAP-VARS-TO-GOOD-NAMES VARS))
         ;; REC-DEFUNS is an alist mapping *REC forms in the original
         ;; descriptor to the meta-recognizer functions which have been
         ;; generated for them.
         (REC-DEFUNS (IF (REC-DESCRIPTORP DESCRIPTOR)
                         (GEN-REC-RECOGNIZERS (CADDR DESCRIPTOR))
                         (GEN-REC-RECOGNIZERS DESCRIPTOR))))
    ;; The REMOVE-DUPLICATES is necessary to keep the compiler happy.
    ;; It is OK, so long as we maintain the requirement that all *REC
    ;; descriptors with the same name are identical.
    (REMOVE-DUPLICATES
      (CONS
        (LIST 'DEFUN
              FNNAME
              (CONS 'THING (ALL-CDRS VAR-MAP))
              (IF (REC-DESCRIPTORP DESCRIPTOR)
                  (GEN-RECOGNIZER-BODY
                    (CADDR DESCRIPTOR) 'THING REC-DEFUNS FNNAME VAR-MAP)
                  (GEN-RECOGNIZER-BODY
                    DESCRIPTOR 'THING REC-DEFUNS NIL VAR-MAP)))
        (ALL-CDRS REC-DEFUNS))
      :TEST #'EQUAL)))

(DEFUN GEN-RECOGNIZER-BODY
       (DESCRIPTOR BASE REC-DEFUNS RECUR-NAME VAR-MAP)
  ;; DESCRIPTOR is a type descriptor or some componenent of a type
  ;;   descriptor.
  ;; BASE is the formal associated with the data object, perhaps in a
  ;;   nest of calls to CAR and CDR.
  ;; REC-DEFUNS is an alist mapping *REC forms in the original descriptor
  ;;   to the functions which have been generated for them.
  ;; RECUR-NAME -- If the form presented to GEN-RECOGNIZERS is a *REC
  ;;   form, RECUR-NAME is the name associated with the *REC.  NIL
  ;;   otherwise.
  ;; VAR-MAP is an alist mapping type descriptor variables occurring in
  ;;   the outer form to names which can be used as parameters to the
  ;;   meta-recognizer without confusing the Lisp reader.  (&1 is
  ;;   interpreted as a keyword parameter tag with a typo.)
  (CASE DESCRIPTOR
    (*EMPTY NIL)
    (*UNIVERSAL T)
    ($CHARACTER `(CHARACTERP ,BASE))
    ($INTEGER `(INTEGERP ,BASE))
    ($NIL `(NULL ,BASE))
    ($NON-INTEGER-RATIONAL
      `(IF (RATIONALP ,BASE) (NULL (INTEGERP ,BASE)) NIL))
    ($NON-T-NIL-SYMBOL `(IF (SYMBOLP ,BASE)
                            (IF (NULL ,BASE)
                                NIL
                                (IF (EQUAL ,BASE T) NIL T))
                            NIL))
    ($STRING `(STRINGP ,BASE))
    ($T `(EQUAL ,BASE T))
    (OTHERWISE
      (IF (VARIABLE-NAMEP DESCRIPTOR)
          (LIST 'APPLY
                (CDR (ASSOC DESCRIPTOR VAR-MAP))
                (LIST 'LIST BASE))
          (CASE (CAR DESCRIPTOR)
```

```
            (*CONS
              `(AND (CONSP ,BASE)
                    ,(GEN-RECOGNIZER-BODY
                       (CADR DESCRIPTOR) `(CAR ,BASE) REC-DEFUNS
                       RECUR-NAME VAR-MAP)
                    ,(GEN-RECOGNIZER-BODY
                       (CADDR DESCRIPTOR) `(CDR ,BASE) REC-DEFUNS
                       RECUR-NAME VAR-MAP)))
            (*OR (CONS 'OR
                       (MAPCAR
                         (FUNCTION
                           (LAMBDA (DISJUNCT)
                             (GEN-RECOGNIZER-BODY DISJUNCT
                                                  BASE
                                                  REC-DEFUNS
                                                  RECUR-NAME
                                                  VAR-MAP)))
                         (CDR DESCRIPTOR))))
            (*NOT
              (LIST 'NOT
                    (GEN-RECOGNIZER-BODY (CADR DESCRIPTOR)
                                         BASE
                                         REC-DEFUNS
                                         RECUR-NAME
                                         VAR-MAP)))
            (*REC
              (LET ((REC-DEFN
                      (CDR (ASSOC DESCRIPTOR REC-DEFUNS :TEST #'EQUAL))))
                (CONS (CADR REC-DEFN)
                      (CONS BASE (CDR (CADDR REC-DEFN))))))
            ;; Here, we assume the *RECUR label matches the nearest
            ;; enclosing *REC.
            (*RECUR (CONS RECUR-NAME (CONS BASE (ALL-CDRS VAR-MAP))))
            (OTHERWISE NIL))))))

(DEFUN GEN-REC-RECOGNIZERS (DESCRIPTOR)
  ;; GEN-REC-RECOGNIZERS searches DESCRIPTOR for *REC forms and calls
  ;; GEN-RECOGNIZERS recursively for each one encountered.  It goes depth
  ;;   first.
  (IF (NULL DESCRIPTOR)
      NIL
      (IF (ATOM DESCRIPTOR)
          NIL
          (IF (REC-DESCRIPTORP DESCRIPTOR)
              (LET ((BODY-RECOGNIZERS
                      (GEN-REC-RECOGNIZERS (CADDR DESCRIPTOR))))
                (CONS (CONS DESCRIPTOR
                            (CAR (GEN-RECOGNIZERS DESCRIPTOR)))
                      BODY-RECOGNIZERS))
              (APPEND (GEN-REC-RECOGNIZERS (CAR DESCRIPTOR))
                      (GEN-REC-RECOGNIZERS (CDR DESCRIPTOR)))))))

(DEFUN GEN-FNNAME (DESCRIPTOR)
  (INTERN (FORMAT NIL "~A-RECOGNIZER" DESCRIPTOR)))

(DEFUN MAP-VARS-TO-GOOD-NAMES (VARS)
  (IF (NULL VARS)
      NIL
      (CONS (CONS (CAR VARS) (INTERN (FORMAT NIL "VAR-~A" (CAR VARS))))
            (MAP-VARS-TO-GOOD-NAMES (CDR VARS)))))

(DEFUN FIND-VARS-IN-DESCRIPTOR (DESCRIPTOR VARS)
  ;; FIND-VARS-IN-DESCRIPTOR just creates a list of all the type
  ;; variables in DESCRIPTOR, with each appearing only once.
  (IF (VARIABLE-NAMEP DESCRIPTOR)
      (ADJOIN DESCRIPTOR VARS :TEST #'EQL)
      (IF (ATOM DESCRIPTOR)
          VARS
          (FIND-VARS-IN-FORM-LIST DESCRIPTOR VARS))))
```

```
(DEFUN ALL-CDRS (L)
  (IF (NULL L)
      NIL
      (CONS (CDR (CAR L))
            (ALL-CDRS (CDR L)))))
```

Now let us apply this semantics for type descriptors in the setting of the type signatures generated by the inference algorithm. A simple example will be a good vehicle for communicating the relevant ideas.

Consider a function:

```
(DEFUN FOO (X)
  (DECLARE (XARGS :GUARD (CONSP X)))
  (IF (EQUAL (CAR X) 0)
      0
      (CONS (CAR X) NIL)))
```

The inference tool gives this function the following signature:

```
  Guards: (*CONS *UNIVERSAL *UNIVERSAL)

  The guards are complete.

  The function calls only functions whose guards are complete.

  Signature segments:
    ((*CONS $INTEGER *UNIVERSAL)) -> $INTEGER
    ((*CONS &5 *UNIVERSAL)) -> (*CONS &5 $NIL)

  The function is not a recognizer.
```

There are two "segments" in this signature. The first says that if the argument Y to FOO is a CONS whose CAR is an integer, then the result of evaluating (FOO Y) can be an integer. In the second segment, &5 is a "type variable". This segment says that if the argument Y is a CONS whose CAR is an object of some arbitrary type T, then the result can be a CONS whose CAR is an object of type T and whose CDR is NIL.

Taken as a whole, this signature means that, on any call (FOO Y), where Y is some data object which satisfies FOO’s guard, there is some segment of this signature and some binding of all type variables appearing in the segment such that Y satisfies the descriptor on the left hand side of the segment under the binding and (FOO Y) satisfies the descriptor on the right hand side of that segment under the same binding.

In the vernacular of meta-recognizer functions, we would say this signature means that, on any call (FOO Y), where Y is some data object which satisfies FOO’s guard, there is some segment of this signature and some binding of each type variable appearing in the segment to some unary meta-recognizer function such that Y satisfies the meta-recognizer function for the descriptor on the left hand side of the segment under the binding and (FOO Y) satisfies the meta-recognizer for the descriptor on the right hand side of that segment under the same binding.

For convenience, I will call this interpretation "OR semantics", the intuition being that for at least one, but not necessarily all, of the segments for which the actual parameters satisfy the left hand side (or parameter descriptors), the function result will satisfy the right hand side. This differs from "AND semantics", which would claim that for any segment whose left hand side is satisfied by the actuals, the right hand side would be satisfied by the result.

We can state this interpretation as a collection of lemmas which should be true for the function. The lemma which characterizes the signature is as follows:

```
   (IMPLIES
     (AND (|($CONS *UNIVERSAL *UNIVERSAL)-RECOGNIZER| X) ; from the guard
          (META-RECOGNIZER-P VAR-FN-1))
     (OR (AND (|(*CONS $INTEGER *UNIVERSAL)-RECOGNIZER| X)
              (|$INTEGER-RECOGNIZER| (FOO X)))
         (AND (|(*CONS &5 *UNIVERSAL)-RECOGNIZER| X VAR-FN-1)
              (|(*CONS &5 $NIL)-RECOGNIZER| (FOO X) VAR-FN-1))))

where:

(DEFUN |($CONS *UNIVERSAL *UNIVERSAL)-RECOGNIZER| (THING)
  NIL)

(DEFUN |(*CONS $INTEGER *UNIVERSAL)-RECOGNIZER| (THING)
  (AND (CONSP THING)
       (INTEGERP (CAR THING))
       T))

(DEFUN $INTEGER-RECOGNIZER (THING)
  (INTEGERP THING))

(DEFUN |(*CONS &5 *UNIVERSAL)-RECOGNIZER| (THING VAR-&5)
  (AND (CONSP THING)
       (APPLY VAR-&5 (LIST (CAR THING)))
       T))

(DEFUN |(*CONS &5 $NIL)-RECOGNIZER| (THING VAR-&5)
  (AND (CONSP THING)
       (APPLY VAR-&5 (LIST (CAR THING)))
       (NULL (CDR THING))))
```

There is one disjunct in the conclusion of the theorem for each segment in the signature. Only one need be true, though more than one can be. Each disjunct is just the conjunction of all the predicates for the left hand side of the segment (In this example, there is only one, because FOO is unary.) with the predicate for the right hand, or result, side of the segment, with the meta-recognizer function VAR-FN-1 providing the glue to bind the interpretation of the type variables.

The task of the checker would be to generate and prove these theorems.

There is also an excruciatingly tedious chain of lemmas which capture the notion of the guard being complete and verified. To get a flavor for these, consider a slightly different function BAR.

```
(DEFUN BAR (X)
  (DECLARE (XARGS :GUARD (IF (CONSP X) (INTEGERP (CAR X)) NIL)))
  (IF (EQUAL (CAR X) 0)
      (CONS (CAR X) T)
      (CONS (CAR X) NIL)))
```

The inference tool returns the result of analyzing BAR:

```
Guards: (*CONS $INTEGER *UNIVERSAL)

The guards are complete.

The function calls only functions whose guards are complete.

Signature segments:
  ((*CONS $INTEGER *UNIVERSAL)) -> (*CONS $INTEGER (*OR $NIL $T))

The function is not a recognizer.
```

Note that we have the functions CAR, CONSP, and INTEGERP already resident in our state. Their signatures are:

For CONSP:

```
Guards: *UNIVERSAL

The guards are complete.

The function calls only functions whose guards are complete.

Signature segments:
  ((*CONS *UNIVERSAL *UNIVERSAL)) -> $T
  ((*OR $CHARACTER $INTEGER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T))
    -> $NIL

The function is a recognizer.  It recognizes forms characterized
by the descriptor:
  (*CONS *UNIVERSAL *UNIVERSAL)
```

For CAR:

```
Guards: (*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL))

The guards are complete.

The function calls only functions whose guards are complete.

Signature segments:
  ((*CONS &1 *UNIVERSAL)) -> &1
  ($NIL) -> $NIL

The function is not a recognizer.
```

For INTEGERP:

```
Guards: *UNIVERSAL

The guards are complete.

The function calls only functions whose guards are complete.

Signature segments:
  ($INTEGER) -> $T
  ((*OR $CHARACTER $NIL $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL
        $STRING $T (*CONS *UNIVERSAL *UNIVERSAL)))
    -> $NIL

The function is a recognizer.  It recognizes forms characterized
by the descriptor:
  $INTEGER
```

Now consider the verification of guards for functions called within the guard expression itself,

```
(IF (CONSP X) (INTEGERP (CAR X)) NIL)
```

We will proceed in evaluation order. To verify the guards for BAR, we would also have to do this analysis on the body of BAR, but it goes the same way.

On entry, we know nothing of the type of X. We need to verify the guard of CONSP is satisfied.

Fortunately, this is easy, since the guard is *UNIVERSAL, i.e., any object is acceptable. We must prove the lemma:

```
Guard Lemma 1.
(IMPLIES ($UNIVERSAL-RECOGNIZER X)    ; the predicate characterizing X
         ($UNIVERSAL-RECOGNIZER X))   ; the predicate for CONSP's guard

where:

(DEFUN $UNIVERSAL-RECOGNIZER (THING)
  T)
```

Easy. Next we evaluate (CAR X), so we need to check its guard, which is

```
(*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL))
```

To do this, we need help from the context. Fortunately, we have it. We have the context provided by the (CONSP X), which gives us a type-alist entry of

```
(X . (*CONS *UNIVERSAL *UNIVERSAL))
```

Thus, we generate the lemma:

```
Guard Lemma 2.
(IMPLIES (|(*CONS *UNIVERSAL *UNIVERSAL)-RECOGNIZER| X)
         (|(*OR $NIL (*CONS *UNIVERSAL *UNIVERSAL))-RECOGNIZER| X))
```

This is also trivially true. Now for (INTEGERP (CAR X)). The guard descriptor for INTEGERP is *UNIVERSAL. X is still of type

```
(*CONS *UNIVERSAL *UNIVERSAL)
```

The algorithm has determined that (CAR X) under this context has type *UNIVERSAL. Thus, we generate the lemma:

```
Guard Lemma 3.
(IMPLIES
  (AND (|(*CONS *UNIVERSAL *UNIVERSAL)-RECOGNIZER| X)  ; from the context
       ($UNIVERSAL-RECOGNIZER (CAR X)))  ; surmised by the inference tool
  ($UNIVERSAL-RECOGNIZER (CAR X)))    ; required by the guard of INTEGERP
```

Again, this is trivially true.

Moving along, NIL requires no guard verification. IF is OK if its three arguments are OK. We are almost done verifying the guards in the guard of BAR. The "top level" lemma that claims we are OK is:

```
(AND (VALID-FS FUNCTION-SIGNATURES)
     ;; COMPLETEP is a predicate which says that the guard is complete
     ;; and  the function calls only functions whose guards are complete.
     (COMPLETEP 'CONSP FUNCTION-SIGNATURES)
     (COMPLETEP 'CAR FUNCTION-SIGNATURES)
     (COMPLETEP 'INTEGERP FUNCTION-SIGNATURES)
     <the conjunction of the three lemmas above>)
```

Also, there is the syntax check that BAR is not called in the guard of BAR, which I will omit.

All this supports the notion that if the world embodied in the initial state consists only functions whose guard descriptors are "complete", and if we add to that world only functions which we determine are complete, then we can guarantee that for any new function which calls only complete functions, if the inference tool decides there are no guard violations, then in fact there can be no guard violations as a result of applying the new function to any arguments which satisfy its guards.

Proof of the soundness of the inference system in the presence of incompletely guarded functions, i.e., functions whose guards are not determined by the system to be complete, is beyond the scope of this exercise. We will confine ourselves to a world in which all functions have guards which are complete and verified.

Finally, there is the issue of the well-formedness of a signature. If we are to generate a theorem whose proof represents the soundness of a function signature, we first need to ensure that the signature is well-formed. This builds fairly directly upon the well-formedness of descriptors. For our purposes, the signature is composed of the GUARD, the GUARD-COMPLETE flag, the function calls complete functions flag, and the SEGMENTS. A signature is well-formed if:

1. The arity of the guard is the same as the arity of the function.
2. The guard is a list of well-formed descriptors.
3. The GUARD-COMPLETE flag is T.
4. The function calls only functions whose guards are complete.
5. The left hand side of each segment is a list of well-formed descriptors whose arity is the same as that of the function.
6. The right hand side of each segment is a well-formed descriptor.
7. All *REC descriptors throughout the signature with the same name must be identical.

Appendix E
A Semantics with Composed Substitutions

This appendix describes an attempt to formulate a semantics for function signatures which was computational in style and allowed for a notion of substitutions for variables which were not necessarily ground, i.e., a variable could be replaced by a descriptor containing a variable. The difficulty we had in formulating this model revealed the painful complexity of a semantics for our function signatures in which type variables represented types. This presentation, in fact, includes the discovery that one of the notions being used was flawed. As such, this discussion provides a real, pragmatic motivation for the switch to value-instantiated variables, a change which was made as the effort to further develop this model began to deteriorate.

We first imagine three functions, two of which relate ground Lisp values to descriptors and the other which relates two descriptors.

```
TCO (TD O A)
(for TYPE-CHECK-OBJECT) determines whether a data object O satisfies a
descriptor TD under some substitution A of variable-free descriptors for
all the variables in the descriptor.

TCOV (TDV OV A)
(for TYPE-CHECK-OBJECT-VECTOR) determines whether a vector of data
objects OV satisfies a vector (of the same arity) of descriptors TDV
under the complete substitution A.

TCTD (TD1 TD2 A)
(for TYPE-CHECK-TYPE-DESCRIPTOR) determines whether a type descriptor
TD1 contains TD2 under some substitution A, i.e., whether the set of
Lisp objects represented by TD2 under A is a subset of those represented
by TD1 under A.
```

Now imagine that we have a function F for which the inference tool returns the following:

```
Guard: (GD1..GDn) == GDV     (guard descriptors are variable-free)

Segments:
 (TD11 .. TDn1) -> TD1
 ..
 (TD1k .. TDnk) -> TDk
```

The semantics of this signature are as follows.

```
For all substitutions A and object vectors (X1.. Xn) = XV,
  (TCOV &iV XV A) and (TCOV GDV XV NIL)
 ->
  for some segment (TD1i .. TDni -> TDi)
    there exists a substitution A’ such that
       (P A A’ (TD1i .. TDni -> TDi))
     and
       (TCOV (TD1i .. TDni TDi) (X1.. Xn (F X1.. Xn)) AA’)

where
 &iV represents a vector of type variables, one for each formal argument
 P is a relation which indicates that all the substitutions in A’
   are consistent with those in A.  By this we mean (TCTD &j TDji AA’),
   i.e., that the substitutions in A’ are such that when applied to
   our original assignment of variables to formal and to the descriptor
   characterizing the formal, the subset relation holds.
 AA’ is the composition of substitutions A and A’.
```

The purpose of complicated relationships among A, A’, and P is to enable us to universally quantify our entire signature over A, but yet to extend A to A’ in order to cover variables nested within a segment’s descriptors which are not in &iV. These extensions are peculiar to each segment, but must be consistent with A. We were facing the situation where we needed an existential quantifier on the right hand side of the main implication, a situation which should raise a red flag, and we wanted to use P to qualify our existentially quantified variable in the strongest way possible.

For example, consider the APPEND function, which might have the signature:

```
Guard: ((*rec tl (*or $nil (*cons *universal (*recur tl))))
        *universal)

Segments:
 ($nil &2) -> &2
 ((*cons &3 (*rec tl (*or $nil (*cons *universal (*recur tl))))) &2)
    -> (*cons &3 (*rec app (*or &2 (*cons *universal (*recur tl)))))
```

We choose &iV = (&1 &2) and let XV = (X Y), and thus we have the following:

```
For all substitutions A grounding &1 and &2 and all values x and y,
   (TCOV (&1 &2) (x y) A)
  and
   (TCOV ((*rec tl (*or $nil (*cons *universal (*recur tl)))) *universal)
         (x y)
         nil)
 ->
   (TCOV ($nil &2 &2) (x y (APPEND x y)) A)
  or
   there exists a substitution A’ grounding &3 such that
    (TCTD &1
          (*cons &3 (*rec tl (*or $nil (*cons *universal (*recur tl)))))
          AA’)
    and
    (TCOV ((*cons &3 (*rec tl (*or $nil (*cons *universal (*recur tl)))))
           &2
           (*cons &3 (*rec app (*or &2 (*cons *universal (*recur tl))))))
          (x y (append x y))
          AA’)
```

Now for example, suppose X has the value (CONS 2 (CONS 3 NIL)) and Y = T. Consider the trivial A = ((&1 . *UNIVERSAL) (&2 . *UNIVERSAL)). Clearly,

```
(TCOV (&1 &2) (x y) A)
```

Also, we can see that

```
(TCOV ((*rec tl (*or $nil (*cons *universal (*recur tl)))) *universal)
      (x y)
      nil)
```

The first segment does not apply to these values of x and y, so the second one must. Let A’ be ((&3 . $INTEGER)). We can see that

```
(TCTD &1
      (*cons &3 (*rec tl (*or $nil (*cons *universal (*recur tl)))))
      AA’)
```

since AA’ maps &1 to *UNIVERSAL. Also

```
(TCOV ((*cons &3 (*rec tl (*or $nil (*cons *universal (*recur tl)))))
       &2
```

```
        (*cons &3 (*rec app (*or &2 (*cons *universal (*recur tl))))))
       (x y (append x y))
       AA')
```

which becomes clearer if we apply the substitutions and fill in the values:

```
(TCOV ((*cons $integer
              (*rec tl (*or $nil (*cons *universal (*recur tl)))))
       *universal
       (*cons $integer
              (*rec app
                    (*or *universal (*cons *universal (*recur tl))))))
       ((2 3) T (2 3 . t))
       nil)
```

But if we had chosen a more useful A, for instance

```
((&1 . (*rec ilist (*or $nil (*cons $integer (*recur ilist)))))
 (&2 . $t))
```

we would have seen that

```
(TCTD &1
      (*cons &3 (*rec tl (*or $nil (*cons *universal (*recur tl)))))
      AA')
```

does not hold. But do need the TCTD at all? Why not streamline the specification as follows?

```
For all substitutions A grounding &1 and &2 and all values x and y,
   (TCOV (&1 &2) (x y) A)
  and
   (TCOV ((*rec tl (*or $nil (*cons *universal (*recur tl)))) *universal)
         (x y)
         nil)
 ->
   (TCOV ($nil &2 &2) (x y (APPEND x y)) A)
  or
   there exists a substitution A' grounding &3 such that
    (TCOV ((*cons &3 (*rec tl (*or $nil (*cons *universal (*recur tl)))))
           &2
           (*cons &3 (*rec app (*or &2 (*cons *universal (*recur tl))))))
          (x y (append x y))
          AA')
```

So the generalization is:

```
For all substitutions A grounding &iV and all object vectors
    (X1.. Xn) = XV,
  (TCOV &iV XV A) and (TCOV GDV XV NIL)
 ->
  for some segment (TD1i .. TDni -> TDi) with variables &jV
    there exists a substitution A' grounding &jV - &iV such that
     (TCOV (TD1i .. TDni TDi) (X1.. Xn (F X1.. Xn)) AA')

where
 &iV represents a vector of type variables, one for each formal argument
 AA' is the composition of substitutions A and A'.
```

Appendix F
Thoughts on the Termination of DUNIFY-DESCRIPTORS

This appendix summarizes an approach to arguing the termination of DUNIFY-DESCRIPTORS. In it, we sketch a definition for a measure which will always decrease on recursive calls, thus allowing for an argument by computational induction.

Our termination argument for DUNIFY-DESCRIPTORS is ridiculously difficult. There are no fewer than four separate factors which contribute to the composite count for the function. None of these factors are simple, and we have not worked them out fully. But here is an overview of them, why they are necessary components, where they come into play, how they relate to one another, and generally why we think they work.

1) The size of the two descriptor arguments. This is the most commonly decreased measure, and the most obvious choice. For instance, we decrease the size when we case split on the disjuncts of an *OR, or when we take the first args of a pair of *CONS forms. It is not a straightforward measure, however, because of complications introduced in several of the *REC rules. Here, for example (from Rule *Rec5), where there are no variables anywhere in the descriptors, is the rule:

```
 (dunify-descriptors (*rec foo (*or d1 (*cons d2 (*recur foo))))
                     (*rec bar (*or d3 (*cons d4 (*recur bar))))
                     s)
  =
  ((*rec bim (*or d13 d14bar d32foo (*cons d24 (*recur bim)))) . s)
```

When we compute d14bar, we recur with d1, which is definitely smaller than foo, but the other argument is the *CONS disjunct of bar, with the recursion opened, so the size of second argument is definitely larger. If the size measure were simply the number of tokens involved, we could lose, because d24 may be larger than foo. Therefore, we would need to give some special premium to the fact that we liquidated the recursive form foo. But even the number of recursive forms is not good enough, because there may be multiple recursive forms in d24, and we could even end up with more than we started. So, it appears we would need to assign a weight to *RECs which nest other *RECs, or even worse (as seems to be required in this case), to *RECs being unified with other *RECs nesting *RECs. One possible way out is to use the TERM-RECS measure discussed below, which seems reasonable given that we open a *REC. Unfortunately, the TERM-RECS mechanism is not employed in the *REC rules as implemented. It would not be hard to do, though.

2) The TERM-RECS measure. This currently comes into play only once, though it may well be the key to reducing the complexity in some other quarters as well, the above-mentioned being only one. The TERM-RECS mechanism is used in DUNIFY-REC-WITH-NON-REC to ensure that we do not get caught in overlapping recursions. For example:

```
(dunify-descriptors
  '(*rec foo (*or $nil (*cons *universal
                              (*cons *universal (*recur foo)))))
  '(*cons *universal
          (*rec foo (*or $nil (*cons *universal
                                     (*cons *universal (*recur foo))))))
  nil nil)
```

would open up the first *REC, then on recursion open up the second, then the first, etc. But DUNIFY-DESCRIPTORS maintains a stack of pairs of operands, so that when it is recursively called with a pair which it has already seen, it can recognize that it is in an endless recursive chain and return a default result. It is possible that a TERM-RECS count could alleviate the problem with the size count, and even eliminate the var-equalities count, to be discussed. But there is a technical problem. Just what is the count corresponding to the TERM-RECS mechanism? Counts must be monotonically decreasing and always non-negative, and the TERM-REC stack grows. So the length of the stack must be subtracted from something, but what is that something? I would guess it to be something like the number of descriptor pairs which could be generated by pairwise combining each of the *REC descriptors in the problem with all descriptors in the full disjunction of both arguments, wherein *REC descriptors are opened once and then treated atomically. Thus, there would only be finitely many pairs, and once we generated them all and put them into TERM-RECS, the next time occasion we had to add one, we would certainly find it already present, which would force termination. Incidentally, I believe the TERM-RECS count would have to be the most dominant count, in that when it is used (i.e., when a *REC is opened on recursion), the size measure is clearly increased, and in doing so we might also increase the variable closure measure yet to be discussed (depending on how it is defined).

3) The variable equalities measure is necessary because of a single case where a variable is being dunified with a *REC in which some possible disjunct is another variable. Examples:

```
(dunify-descriptors &i (*rec foo (*or &j .. (*cons .. (*recur foo)))) s)

(dunify-descriptors
  &i (*rec foo (*or (*rec bar (*or &j .. (*cons .. (*recur bar))))
                    .. (*cons .. (*recur foo))))
  s)
```

Here, we case split, recurring on &i and &j in one case, and on &i and the other possibilities in the other. The reason for this has to do with wanting to prevent substitutions of the form

```
(&i . (*rec foo (*or &j .. (*cons .. (*recur foo)))))
```

where $i < j$, thus ensuring that, even in the case of narrowing a binding, all our substitutions are downward directed on the lexical ordering of variable names. This is important to our method for avoiding loops. The problem, of course, is that when we recur in the second case, the second argument is larger in size, there may be replicated variables, (there are certainly no fewer) and so no natural variable measure is decreasing, and nothing else decreases either. But what we ARE eliminating is the possibility that the binding of one variable is another, and if we treat the number of instances where this possibility is evident in the descriptors as a count, then we would in fact be reducing it. Again, the difficulty is to say exactly how to compute this count, and again, the trick is possibly to treat *REC descriptors specially and do an exhaustive counting as follows. If we unify two different variables, the count is 1. If we unify with a term not containing a variable, the count is 0. If we unify something with an *OR, the count is the sum of the counts of unifying with the disjuncts. If we unify with two *CONSes, the count is the sum of the counts of the CARs and the CDRs. If we unify a non-*CONS (other than a *REC) with a *CONS, the count is 0. If we unify a variable with a *REC, the count is the number of possible occurrences of a different var as a disjunct of the *REC. If we unify a non-variable with a *REC, the count is the count from recurring with the opened *REC, but subject to the TERM-RECS termination mechanism for opening *RECs. If we unify two *RECs, the count is 0 (we do not need to use the count from their exposed guts). Thus, the count never increases, and we reduce it in two cases. The first, when we unify two variables, is a terminal case. The other is the case for which we need it. So for example, when we unify

```
(*CONS &1 &2)
```

and

```
(*OR &3 (*CONS &4 (*REC FOO (*OR &5 (*CONS *UNIVERSAL (*RECUR FOO))))))
```

the count is 2, one for the pairing of &1 and &4, and the other for the pairing of &2 with &5. What seems odd is that, on the one case which matters, we had to inject the TERM-RECS mechanism to keep the counter from running away. This seems to suggest we could substitute the TERM-RECS count, since it is maximally significant.

4) Finally, we need some measure on the variables in our problem, which I will call the variable closure measure. I do not know just what this measure should be. In the termination argument for classical unification, this is just the number of unbound variables in the terms, under the binding list. Thus, the measure goes down every time a variable is bound. But, for our algorithm, we need a measure which will decrease whenever we proceed from unifying a variable with some descriptor to recursively unifying the descriptor with the previous binding of a variable in the substitution list. We know that the variable will not appear in either term, nor can there be a loop through the binding list back to the variable. But a couple of examples will illustrate the difficulties of finding the right measure. First, an unbound variables measure will not do. Consider:

```
(dunify-descriptors
  '&4 '(*cons &2 $nil) ((&4 . (*cons &3 $nil)) (&2 . &1) (&1 . $nil)))
```

The recursion, then, is to unify (*CONS &2 $NIL) with (*CONS &3 $NIL). In taking this step, we are not reducing the number of unbound variables in our problem, since &4 was already bound. We could be said to be reducing the length of the substitution chains down to unbound variables, since we are knocking &4 out of consideration. But neither the length of such chains, nor the sum of their lengths, nor the maximum of their lengths, nor any other measure I can think of, is strictly monitonically decreasing. In the previous example, we can see that the unification will bind &3 to &2, thus linking to the chain down to &1, which is longer chain than what we started with. Furthermore, we are not always taking the variable whose binding we are considering out of future consideration. Consider:

```
(dunify-descriptors
  '&2
  '(*cons &3 $integer)
  '((&2 . &1)
    (&1 . (*rec bar (*or $integer (*cons $integer (*recur bar)))))
    (&3 . &2)))
```

This results in unifying (*CONS &3 $INTEGER) with &1, then recusively unifying (*CONS &3 $INTEGER) with

```
(*REC BAR (*OR $INTEGER (*CONS $INTEGER (*RECUR BAR))))
```

&3 with $INTEGER, &2 with $INTEGER, and &1 with $INTEGER. Thus, we revisited both &2 and &1, albeit with shorter arguments. (Only upon emerging from this recursion do we discover we have an ill-formed substitution caused by the inconsistent bindings of &1, but this does not bear on the termination question.) So all I can say is that I believe there is a measure which goes down when we unify a new binding for a variable with its old one, but I have not found it yet. It is probable that this measure is itself a composite, perhaps involving the number of *REC forms in the substitution list, but that is speculation. This speculation lends itself to the further speculation that TERM-RECS might play yet another role.

Assuming these four measures exist, however, and depending on their final definitions, I would probably claim that the TERM-RECS is the most dominant, followed by the variable equalities and the variable closure measures, and finally the term size, and that none increases except in cases where some more dominant measure decreased, and that at least one decreases in all cases, as noted in the source code.

Appendix G
Selected Functions from the Implementation

This appendix contains the definitions of a few selected functions from the system implementation which seem worthy of inclusion in this document. All the code in the implementation is available via anonymous ftp from ftp.cli.com. Appendix H provides instructions on how to retrieve the code and other information relevant to this thesis.

G.1 WELL-FORMED-DESCRIPTOR

```
;; -------------------------------------------------------------------
;; WELL-FORMED-DESCRIPTOR
;; -------------------------------------------------------------------

(DEFUN WELL-FORMED-DESCRIPTOR (DESCRIPTOR)
  ;; Note that our parameter to NEAREST-REC is NIL.  We use this to say
  ;; that there is no nearest enclosing *REC.  This suggests that we do
  ;; not allow NIL to be a *REC name, since if we did, this way of
  ;; characterizing our assumption would not be valid.  In fact, NIL
  ;; cannot be a *REC name, since *REC names can be only the names of
  ;; recognizer functions or atoms whose initial characters are "!REC"
  ;; followed by a natural number, and NIL is not allowed as the name
  ;; of a function.
  (AND (WFF-DESCRIPTOR DESCRIPTOR NIL NIL NIL)
       (ALL-SAME-NAME-RECS-IDENTICAL (ALL-REC-FORMS DESCRIPTOR))))

(DEFUN WFF-DESCRIPTOR (DESCRIPTOR NEAREST-REC INSIDE-OR INSIDE-CONS)
  (IF (MEMBER DESCRIPTOR
              '(*EMPTY *UNIVERSAL $CHARACTER $INTEGER $NIL
                       $NON-INTEGER-RATIONAL $NON-T-NIL-SYMBOL $STRING
                       $T))
      T
      (IF (VARIABLE-NAMEP DESCRIPTOR)
          T
          (IF (ATOM DESCRIPTOR)
              NIL
              (CASE (CAR DESCRIPTOR)
                (*OR (WFF-DESCRIPTOR-LIST
                       (CDR DESCRIPTOR) NEAREST-REC T INSIDE-CONS))
                (*CONS
                  (AND (CONSP (CDR DESCRIPTOR))
                       (WFF-DESCRIPTOR
                         (CADR DESCRIPTOR) NEAREST-REC INSIDE-OR T)
                       (CONSP (CDDR DESCRIPTOR))
                       (WFF-DESCRIPTOR
                         (CADDR DESCRIPTOR) NEAREST-REC INSIDE-OR T)
                       (NULL (CDDDR DESCRIPTOR))))
                (*NOT
                  (AND (CONSP (CDR DESCRIPTOR))
                       (WFF-DESCRIPTOR (CADR DESCRIPTOR)
                                       NEAREST-REC
                                       INSIDE-OR
                                       INSIDE-CONS)
                       (NULL (CDDR DESCRIPTOR))))
                (*REC
                  (AND (CONSP (CDR DESCRIPTOR))
                       (SYMBOLP (CADR DESCRIPTOR))
                       (CONSP (CDDR DESCRIPTOR))
                       (WFF-DESCRIPTOR
                         (CADDR DESCRIPTOR) (CADR DESCRIPTOR) NIL NIL)
                       (NO-REPLICATING-VARS (CADDR DESCRIPTOR)
```

```
                                            (CADR DESCRIPTOR))
                       (NULL (CDDDR DESCRIPTOR))))
                (*RECUR
                  (AND (CONSP (CDR DESCRIPTOR))
                       (EQUAL (CADR DESCRIPTOR) NEAREST-REC)
                       (NULL (CDDR DESCRIPTOR))
                       (AND INSIDE-OR INSIDE-CONS)))
                (OTHERWISE NIL))))))

(DEFUN WFF-DESCRIPTOR-LIST (DESCRIPTOR-LIST NEAREST-REC INSIDE-OR
                            INSIDE-CONS)
  (IF (NULL DESCRIPTOR-LIST)
      T
      (IF (ATOM DESCRIPTOR-LIST)
          NIL
          (AND (WFF-DESCRIPTOR
                 (CAR DESCRIPTOR-LIST) NEAREST-REC INSIDE-OR INSIDE-CONS)
               (WFF-DESCRIPTOR-LIST (CDR DESCRIPTOR-LIST)
                                    NEAREST-REC
                                    INSIDE-OR
                                    INSIDE-CONS)))))

(DEFUN ALL-REC-FORMS (DESCRIPTOR)
  (IF (ATOM DESCRIPTOR)
      NIL
      (IF (REC-DESCRIPTORP DESCRIPTOR)
          (APPEND (LIST DESCRIPTOR)
                  (ALL-REC-FORMS (CADDR DESCRIPTOR)))
          (APPEND (ALL-REC-FORMS (CAR DESCRIPTOR))
                  (ALL-REC-FORMS (CDR DESCRIPTOR))))))

(DEFUN ALL-SAME-NAME-RECS-IDENTICAL (ALL-RECS)
  (IF (NULL ALL-RECS)
      T
      (AND (HEAD-IS-SAME-NAME-OK (CAR ALL-RECS) (CDR ALL-RECS))
           (ALL-SAME-NAME-RECS-IDENTICAL (CDR ALL-RECS)))))

(DEFUN HEAD-IS-SAME-NAME-OK (REC REC-LIST)
  (IF (NULL REC-LIST)
      T
      (AND (IF (EQUAL (CADR REC) (CADR (CAR REC-LIST)))
               (EQUAL REC (CAR REC-LIST))
               T)
           (HEAD-IS-SAME-NAME-OK REC (CDR REC-LIST)))))

(DEFUN NO-REPLICATING-VARS (REC-BODY REC-NAME)
  (LET ((OR-LIFTED-BODY (LIFT-ORS REC-BODY)))
    (IF (OR-DESCRIPTORP OR-LIFTED-BODY)
        (NO-REPLICATING-VARS-1 OR-LIFTED-BODY REC-NAME)
        (NO-REPLICATING-VARS-1 (LIST OR-LIFTED-BODY) REC-NAME))))

(DEFUN NO-REPLICATING-VARS-1 (DISJUNCTS REC-NAME)
  (IF (NULL DISJUNCTS)
      T
      (IF (NOT (INP-EQUAL (LIST '*RECUR REC-NAME) (CAR DISJUNCTS)))
          (NO-REPLICATING-VARS-1 (CDR DISJUNCTS) REC-NAME)
          (IF (NULL (GATHER-VARS-IN-DESCRIPTOR (CAR DISJUNCTS)))
              (NO-REPLICATING-VARS-1 (CDR DISJUNCTS) REC-NAME)
              NIL))))
```

G.2 WELL-FORMED-SIGNATURE

```
;; -------------------------------------------------------------------
;; WELL-FORMED-SIGNATURE
;; -------------------------------------------------------------------

;; A signature is well-formed iff
;;
;; 1 -- The arity of the guard is the same as the arity of the function.
;; 2 -- The guard is a list of well-formed descriptors.
;; 3 -- The GUARD-COMPLETE flag is T.
;; 4 -- The left hand side of each segment is a list of well-formed
;;      descriptors whose arity is the same as that of the function.
;; 5 -- The right hand side of each segment is a well-formed descriptor.
;; 6 -- All *REC descriptors throughout the signature with the same name
;;      must be identical.

(DEFUN WELL-FORMED-SIGNATURE (SIGNATURE FUNCTION-ARITY)
  (LET ((GUARD (SIGNATURE-BEST-GUARD SIGNATURE))
        ;;(GUARD-COMPLETE (SIGNATURE-GUARD-COMPLETE SIGNATURE))
        (SEGMENTS (INSTANTIATED-SIGNATURE-SEGMENTS SIGNATURE)))
    (WELL-FORMED-SIGNATURE-1 GUARD SEGMENTS FUNCTION-ARITY)))

(DEFUN WELL-FORMED-SIGNATURE-1 (GUARD SEGMENTS FUNCTION-ARITY)
    (AND (EQUAL (LENGTH GUARD) FUNCTION-ARITY)
         (ALL-WELL-FORMED-DESCRIPTORS GUARD)
         (NULL (TC-GATHER-VARS-IN-FORM GUARD))
         (ALL-WELL-FORMED-SEGMENTS SEGMENTS FUNCTION-ARITY)
         (ALL-SAME-NAME-RECS-IDENTICAL
           (ALL-SIGNATURE-RECS GUARD SEGMENTS))))

(DEFUN ALL-WELL-FORMED-DESCRIPTORS (DESCRIPTOR-LIST)
  (IF (NULL DESCRIPTOR-LIST)
      T
      (AND (WELL-FORMED-DESCRIPTOR (CAR DESCRIPTOR-LIST))
           (ALL-WELL-FORMED-DESCRIPTORS (CDR DESCRIPTOR-LIST)))))

(DEFUN ALL-WELL-FORMED-SEGMENTS (SEGMENTS FUNCTION-ARITY)
  (IF (NULL SEGMENTS)
      T
      (AND (EQUAL (LENGTH (SCHEMA-CONTEXT (CAR SEGMENTS)))
                  FUNCTION-ARITY)
           (ALL-WELL-FORMED-DESCRIPTORS (SCHEMA-CONTEXT (CAR SEGMENTS)))
           (WELL-FORMED-DESCRIPTOR (SCHEMA-RESULT (CAR SEGMENTS)))
           (ALL-WELL-FORMED-SEGMENTS (CDR SEGMENTS) FUNCTION-ARITY))))

(DEFUN ALL-SIGNATURE-RECS (GUARD SEGMENTS)
  (APPEND (ALL-RECS-FROM-DESCRIPTOR-LIST GUARD)
          (ALL-RECS-FROM-SEGMENTS SEGMENTS)))

(DEFUN ALL-RECS-FROM-SEGMENTS (SEGMENTS)
  (IF (NULL SEGMENTS)
      NIL
      (APPEND
        (ALL-RECS-FROM-DESCRIPTOR-LIST (SCHEMA-CONTEXT (CAR SEGMENTS)))
        (ALL-REC-FORMS (SCHEMA-RESULT (CAR SEGMENTS)))
        (ALL-RECS-FROM-SEGMENTS (CDR SEGMENTS)))))

(DEFUN ALL-RECS-FROM-DESCRIPTOR-LIST (DESCRIPTOR-LIST)
  (IF (NULL DESCRIPTOR-LIST)
      NIL
      (APPEND (ALL-REC-FORMS (CAR DESCRIPTOR-LIST))
              (ALL-RECS-FROM-DESCRIPTOR-LIST (CDR DESCRIPTOR-LIST)))))
```

G.3 WELL-FORMED-STATE-ASSUMING-ARITY

```
;; ------------------------------------------------------------------
;; WELL-FORMEDNESS OF THE STATE
;; ------------------------------------------------------------------

(DEFUN WELL-FORMED-STATE-ASSUMING-ARITY (FUNCTION-SIGNATURES)
  ;; Without having the function definition in hand, it is impossible to
  ;; know if the arity embodied in the signature is correct, so we have
  ;; to assume it is.
  ;; Here we assume FUNCTION-SIGNATURES is a list of forms for which CADR
  ;; is defined.
  (IF (NULL FUNCTION-SIGNATURES)
      T
      (IF (ATOM FUNCTION-SIGNATURES)
          NIL
          (AND (WELL-FORMED-SIGNATURE
                 (CAR FUNCTION-SIGNATURES)
                 (LENGTH (SIGNATURE-GUARD (CAR FUNCTION-SIGNATURES))))
               (WELL-FORMED-STATE-ASSUMING-ARITY
                 (CDR FUNCTION-SIGNATURES))))))
```

G.4 Guard Predicates for E

```
;; ------------------------------------------------------------------
;; Well-formedness predicates for the arguments to E
;; ------------------------------------------------------------------

(DEFUN WELL-FORMED-VALUE (VALUE)
  (OR (CHARACTERP VALUE)
      (RATIONALP VALUE)
      (SYMBOLP VALUE)
      (STRINGP VALUE)
      (AND (CONSP VALUE)
           (WELL-FORMED-VALUE (CAR VALUE))
           (WELL-FORMED-VALUE (CDR VALUE)))))

(DEFUN WELL-FORMED-ENV (ENV)
  (IF (NULL ENV)
      T
      (AND (CONSP (CAR ENV))
           (SYMBOLP (CAR (CAR ENV)))
           (WELL-FORMED-VALUE (CDR (CAR ENV)))
           (WELL-FORMED-ENV (CDR ENV)))))

(DEFUN WELL-FORMED-WORLD (WORLD)
  << embodies the aforementioned specification >> )

(DEFUN WELL-FORMED-CLOCK (CLOCK)
  (AND (INTEGERP CLOCK)
       (>= CLOCK 0)))

(DEFUN WELL-FORMED-FORM (FORM ENV WORLD)
  (OR (CHARACTERP FORM)
      (RATIONALP FORM)
      (STRINGP FORM)
      (EQUAL FORM T)
      (EQUAL FORM NIL)
      (ASSOC FORM ENV)                      ; a variable in ENV
      (AND (CONSP FORM)
           (TRUE-LISTP FORM)
           (OR (AND (EQUAL (CAR FORM) 'QUOTE)
                    (EQUAL (LENGTH FORM) 2)
                    (WELL-FORMED-VALUE (CADR FORM)))
```

```
               (AND (EQUAL (CAR FORM) 'IF)
                    (EQUAL (LENGTH FORM) 4)
                    (WELL-FORMED-FORM (CADR FORM))
                    (WELL-FORMED-FORM (CADDR FORM))
                    (WELL-FORMED-FORM (CADDDR FORM)))
               (LET ((WORLD-ENTRY (ASSOC (CAR FORM) WORLD)))
                 (AND WORLD-ENTRY
                      (EQUAL (LENGTH (CDR FORM))
                             (LENGTH (CADR WORLD-ENTRY)))
                      (ALL-WELL-FORMED-FORMS (CDR FORM) ENV WORLD))))))

(DEFUN ALL-WELL-FORMED-FORMS (FORMS ENV WORLD)
  (IF (NULL FORMS)
      T
      (AND (WELL-FORMED-FORM (CAR FORMS) ENV WORLD)
           (ALL-WELL-FORMED-FORMS (CDR FORMS) ENV WORLD))))
```

G.5 Guard Predicates for INTERP-SIMPLE

```
;; ------------------------------------------------------------------
;; Well-formedness predicates for the arguments to INTERP-SIMPLE
;; ------------------------------------------------------------------

(DEFUN ALL-WELL-FORMED-VALUES (VALUES)
  (IF (NULL VALUES)
      T
      (IF (ATOM VALUES)
          NIL
          (AND (WELL-FORMED-VALUE (CAR VALUES))
               (ALL-WELL-FORMED-VALUES (CDR VALUES))))))

(DEFUN WELL-FORMED-BINDINGS (BINDINGS DESCRIPTORS)
  (LET ((DESCRIPTOR-VARS (FIND-VARS-IN-FORM DESCRIPTORS NIL)))
    (AND (ALL-BINDINGS-WELL-FORMED BINDINGS)
         (ALL-VARS-BOUND DESCRIPTOR-VARS BINDINGS))))

(DEFUN ALL-BINDINGS-WELL-FORMED (BINDINGS)
  (IF (NULL BINDINGS)
      T
      (IF (ATOM BINDINGS)
          NIL
          (AND (CONSP (CAR BINDINGS))
               (VARIABLE-NAMEP (CAR (CAR BINDINGS)))
               (WELL-FORMED-VALUE (CDR (CAR BINDINGS)))
               (ALL-BINDINGS-WELL-FORMED (CDR BINDINGS))))))

(DEFUN ALL-VARS-BOUND (VARS BINDINGS)
  (IF (NULL VARS)
      T
      (AND (ASSOC (CAR VARS) BINDINGS)
           (ALL-VARS-BOUND (CDR VARS) BINDINGS))))
```

G.6 TC-PREPASS

```
;; ------------------------------------------------------------------
;; TC-PREPASS
;; ------------------------------------------------------------------

(DEFUN TC-PREPASS (FORM FUNCTION-SIGNATURES)
  ;; Here we expect FORM to conform to the syntactic requirements of
  ;; WELL-FORMED-FORM, but we are not concerned with the names of any
```

```
  ;; symbols.  FUNCTION-SIGNATURES should be well-formed according
  ;; to WELL-FORMED-STATE-ASSUMING-ARITY.
  (IF (ATOM FORM)
      FORM
      (IF (IS-QUOTED-FORM FORM)  ;; **** defined in recognizer.lisp
          FORM
          (IF (EQL (CAR FORM) 'IF)
              (LET ((PREPASSED-PRED
                      (TC-PREPASS-IF-PRED
                        (TC-PREPASS (CADR FORM) FUNCTION-SIGNATURES)
                        FUNCTION-SIGNATURES)))
                (IF (NULL PREPASSED-PRED)
                    (TC-PREPASS (CADDDR FORM) FUNCTION-SIGNATURES)
                    (IF (EQUAL PREPASSED-PRED T)
                        (TC-PREPASS (CADDR FORM) FUNCTION-SIGNATURES)
                        (LIST 'IF
                              PREPASSED-PRED
                              (TC-PREPASS (CADDR FORM)
                                          FUNCTION-SIGNATURES)
                              (TC-PREPASS (CADDDR FORM)
                                          FUNCTION-SIGNATURES)))))
              (CONS (CAR FORM)
                    (MAPCAR (FUNCTION
                              (LAMBDA (ARG)
                                (TC-PREPASS ARG FUNCTION-SIGNATURES)))
                            (CDR FORM)))))))

(DEFUN TC-PREPASS-IF-PRED (PRED FUNCTION-SIGNATURES)
  ;; This function returns a form which can only evaluate to T or NIL.
  (IF (NULL PRED)
      NIL
      (IF (IS-QUOTED-FORM PRED)
          (LET ((DESCRIPTOR (DESCRIPTOR-FROM-QUOTE PRED)))
            (IF (EQL DESCRIPTOR '$NIL)
                NIL
                T))
          (IF (ATOM PRED)
              `(NULL (NULL ,PRED))
              (IF (EQL (CAR PRED) 'IF)
                  (LET ((PREPASSED-PRED
                          (TC-PREPASS-IF-PRED (CADR PRED)
                                              FUNCTION-SIGNATURES))
                        ;; The THEN and ELSE arms are predicates, too,
                        ;; since they will be the results of this
                        ;; predicate
                        (PREPASSED-THEN
                          (TC-PREPASS-IF-PRED (CADDR PRED)
                                              FUNCTION-SIGNATURES))
                        (PREPASSED-ELSE
                          (TC-PREPASS-IF-PRED (CADDDR PRED)
                                              FUNCTION-SIGNATURES)))
                    (LIST
                      'IF PREPASSED-PRED PREPASSED-THEN PREPASSED-ELSE))
                  (LET ((PREPASSED-ARGS
                          (MAPCAR
                            (FUNCTION
                              (LAMBDA (FORM)
                                (TC-PREPASS FORM FUNCTION-SIGNATURES)))
                            (CDR PRED))))
                    (IF (TC-ONLY-T-NIL-RESULTS-FN
                          (CAR PRED) FUNCTION-SIGNATURES)
                        (CONS (CAR PRED) PREPASSED-ARGS)
                        `(NULL (NULL ,(CONS (CAR PRED)
                                            PREPASSED-ARGS))))))))))

(DEFUN TC-ONLY-T-NIL-RESULTS-FN (FNNAME FUNCTION-SIGNATURES)
  (LET ((SIGNATURE (GET-SIGNATURE FNNAME FUNCTION-SIGNATURES)))
    (IF (NULL SIGNATURE)
        NIL
```

```
        (TC-ONLY-T-NIL-RESULTS-FN-1
          (CDR (SIGNATURE-SEGMENTS SIGNATURE))))))

(DEFUN TC-ONLY-T-NIL-RESULTS-FN-1 (SEGMENTS)
  (IF (NULL SEGMENTS)
      T
      (IF (MEMBER (SCHEMA-RESULT (CAR SEGMENTS))
                  '($T $NIL (*OR $NIL $T))
                  :TEST #'EQUAL)
          (ONLY-T-NIL-RESULTS-FN-1 (CDR SEGMENTS))
          NIL)))
```

G.7 DESCRIPTOR-FROM-QUOTE

```
(DEFUN IS-QUOTED-FORM (FORM)
  (OR (IS-EXPLICITLY-QUOTED-FORM FORM)
      (INTEGERP FORM)
      (IF (MEMBER FORM '(T NIL)) T NIL)
      (RATIONALP FORM)
      (CHARACTERP FORM)
      (STRINGP FORM)
      (KEYWORDP FORM)))

(DEFUN IS-EXPLICITLY-QUOTED-FORM (FORM)
  (AND (CONSP FORM)
       (EQUAL (CAR FORM) 'QUOTE)
       (CONSP (CDR FORM))
       (NULL (CDDR FORM))))

(DEFUN DESCRIPTOR-FROM-QUOTE (FORM)
  ;; (IS-QUOTED-FORM FORM) is assumed.
  ;; This function takes a quoted form, which may be a self-evaluating
  ;; literal or a (QUOTE foo) and passes it along to
  ;; DESCRIPTOR-FROM-QUOTED-FORM with the outside QUOTE, if there is one,
  ;; stripped off.
  (IF (IS-EXPLICITLY-QUOTED-FORM FORM)
      (DESCRIPTOR-FROM-QUOTED-FORM (CADR FORM))
      (DESCRIPTOR-FROM-QUOTED-FORM FORM)))

(DEFUN DESCRIPTOR-FROM-QUOTED-FORM (FORM)
  ;; This function takes a form which is either a self-evaluating
  ;; literal or a cons structure which is understood to have been quoted.
  (IF (NULL FORM)
      '$NIL
      (IF (EQUAL FORM T)
          '$T
          (IF (INTEGERP FORM)
              '$INTEGER
              (IF (RATIONALP FORM)
                  '$NON-INTEGER-RATIONAL
                  (IF (CHARACTERP FORM)
                      '$CHARACTER
                      (IF (STRINGP FORM)
                          '$STRING
                          (IF (SYMBOLP FORM)
                              '$NON-T-NIL-SYMBOL
                              (LIST '*CONS
                                    (DESCRIPTOR-FROM-QUOTED-FORM
                                      (CAR FORM))
                                    (DESCRIPTOR-FROM-QUOTED-FORM
                                      (CDR FORM))))))))))))
```

G.8 WELL-FORMED-SUBSTS

```
;; ----------------------------------------------------------------------
;; The predicates for well-formed substitutions
;; ----------------------------------------------------------------------

(DEFUN WELL-FORMED-SUBSTS (SUBSTS)
  ;; Here we assume SUBSTS is an alist whose keys are type variables and
  ;; whose associated values are all well-formed descriptors.  (Actually,
  ;; we do not rely here on the full well-formedness requirement for
  ;; descriptors, but it should hold if the world is sane, anyway.
  (WELL-FORMED-SUBSTS-1 SUBSTS SUBSTS))

(DEFUN WELL-FORMED-SUBSTS-1 (CANDIDATES SUBSTS)
  (IF (NULL CANDIDATES)
      T
      (AND (WELL-FORMED-ROOT (CAR CANDIDATES) SUBSTS)
           (WELL-FORMED-SUBSTS-1 (CDR CANDIDATES) SUBSTS))))

(DEFUN WELL-FORMED-ROOT (CANDIDATE SUBSTS)
  (AND (ORDERED-IF-VAR CANDIDATE)
       (WELL-FORMED-ROOT-1
         (LIST (CAR CANDIDATE)) (CDR CANDIDATE) SUBSTS)))

(DEFUN ORDERED-IF-VAR (SUBST)
  ;; If the subst maps to a variable, make sure the descriptor-ord of the
  ;; range variable is less than that of the domain.
  (IF (VARIABLE-NAMEP (CDR SUBST))
      (DESCRIPTOR-ORDER (CDR SUBST) (CAR SUBST))
      T))

(DEFUN WELL-FORMED-ROOT-1 (VARS FORM SUBSTS)
  (IF (ANY-VARS-IN-FORM VARS FORM)
      NIL
      (STEP-VARS-IN-FORM (FIND-VARS-IN-FORM FORM NIL) VARS SUBSTS)))

(DEFUN ANY-VARS-IN-FORM (VARS FORM)
  (IF (NULL VARS)
      NIL
      (OR (INP (CAR VARS) FORM)
          (ANY-VARS-IN-FORM (CDR VARS) FORM))))

(DEFUN STEP-VARS-IN-FORM (VARS-IN-FORM VARS SUBSTS)
  (IF (NULL VARS-IN-FORM)
      T
      (IF (WELL-FORMED-ROOT-1 (CONS (CAR VARS-IN-FORM) VARS)
                              (CDR (ASSOC (CAR VARS-IN-FORM) SUBSTS))
                              SUBSTS)
          (STEP-VARS-IN-FORM (CDR VARS-IN-FORM) VARS SUBSTS)
          NIL)))
```

G.9 DMERGE-NEW-SUBST

```
(DEFUN DMERGE-NEW-SUBST (VARIABLE DESCRIPTOR SUBSTS)
  ;; VARIABLE should be a type variable.
  ;; DESCRIPTOR should be a well-formed descriptor.
  ;; SUBSTS should be an alist whose keys are type variables and
  ;;  whose associated values are well-formed descriptors.
  ;;
  ;; We are merging the substitution (variable . descriptor) into
  ;; the substitution SUBSTS.  We return a list of substitutions.
  ;; We do a bit of runtime validation with the well-formed-substs
  ;; predicate.  Substs are formed only through this function, so
  ;; checking here is good enough to guarantee that no ill-formed
```

```
  ;; substitutions are formed.
  ;; Originally, I would break cold if I found an ill-formed subst.
  ;; But now I believe that I will never form an ill-formed subst
  ;; when any valid subst could be found, and that when I form an
  ;; ill-formed subst, it signifies a result which should not be
  ;; returned.  This can legitimately happen:
  ;; (dunify-descriptors
  ;;   '&2
  ;;   '(*rec foo (*or &3 (*cons *universal (*recur foo))))
  ;;   '((&2 . &1)
  ;;     (&1 . (*rec bar (*or $nil (*cons $integer (*recur bar)))))
  ;;     (&3 . &2))
  ;;   nil)
  ;; forms, among other valid substs,
  ;;  ((&2 . &1)
  ;;   (&1 *CONS $INTEGER
  ;;             (*REC !REC2 (*OR &3 (*CONS $INTEGER (*RECUR !REC2)))))
  ;;   (&3 . &2))
  ;; Wrong solution.  Consider
  ;;  (dunify-descriptors '(*cons &1 &1)
  ;;                      '(*CONS &2 (*CONS (*OR $T &2) $NIL))
  ;;                      NIL
  ;;                      NIL)
  ;; This creates the substitution
  ;;  ((&1 *CONS (*OR $T &2) $NIL) (&2 . &1))
  ;; which contains a loop, but can satisfy I-S with the binding
  ;;  ((&1 . (cons t nil)) (&2 . (cons t nil)))
  ;; So, we could surgically cut the loop, removing the irrelevant
  ;; occurrence of &2 in the binding for &1.  But in general, that
  ;; could be hard.  So instead we just forget about removing looping
  ;; substs, and spot them in dunify-descriptors-interface prior to
  ;; using dapply-subst-list.
  ;; The saving grace is that this will probably never happen.
  ;; At least, I can't figure out how to make this unification
  ;; occur in a type inference scenario.       26-Aug-93
  (LET ((CURRENT-SUBST (CDR (ASSOC VARIABLE SUBSTS))))
    (IF (NULL CURRENT-SUBST)
        (LIST (DINSERT-SUBST VARIABLE DESCRIPTOR SUBSTS))
        ;; Termination: Down on the number of variables ahead
        (LET ((DUNIFIED-FORMS
               (DUNIFY-DESCRIPTORS DESCRIPTOR CURRENT-SUBST SUBSTS NIL)))
          (DMERGE-DUNIFIED-FORMS-INTO-SUBSTS
            DUNIFIED-FORMS VARIABLE SUBSTS)))))

(DEFUN DMERGE-DUNIFIED-FORMS-INTO-SUBSTS (DUNIFIED-FORMS VARIABLE SUBSTS)
  (IF (NULL DUNIFIED-FORMS)
      NIL
      (APPEND (DMERGE-DUNIFIED-FORM-INTO-SUBSTS
                (CAR DUNIFIED-FORMS) VARIABLE SUBSTS)
              (DMERGE-DUNIFIED-FORMS-INTO-SUBSTS
                (CDR DUNIFIED-FORMS) VARIABLE SUBSTS))))

(DEFUN DMERGE-DUNIFIED-FORM-INTO-SUBSTS (DUNIFIED-FORM VARIABLE SUBSTS)
  ;; DUNIFIED-FORM is one of the results produced by DUNIFY-DESCRIPTORS
  ;; when we merge a new binding for VARIABLE into a SUBSTS list which
  ;; already binds it.
  (LET ((DIFFERENT-SUBSTS
          (OTHER-SUBSTS-DIFFERENT VARIABLE
                                  SUBSTS
                                  (DUNIFIED-FORM-SUBSTS DUNIFIED-FORM))))
    (IF (NULL DIFFERENT-SUBSTS)
        (LIST (DREPLACE-SUBST VARIABLE
                              (DUNIFIED-FORM-DESCRIPTOR DUNIFIED-FORM)
                              SUBSTS))
        (LET ((SECOND-ORDER-SUBSTS
                (DMERGE-SECOND-ORDER-SUBSTS DIFFERENT-SUBSTS
                                            DUNIFIED-FORM
                                            (LIST SUBSTS))))
          (DREPLACE-SUBST-IN-SECOND-ORDER-SUBSTS
```

```
            VARIABLE
            (DUNIFIED-FORM-DESCRIPTOR DUNIFIED-FORM)
            SECOND-ORDER-SUBSTS)))))

(DEFUN OTHER-SUBSTS-DIFFERENT (VARIABLE OLD-SUBSTS NEW-SUBSTS)
  ;; Accumulates the variable names from new-substs whose bindings
  ;; are not equal in the corresponding old-substs.  This includes
  ;; new substs which  are not present in old-substs.
  (IF (NULL NEW-SUBSTS)
      NIL
      (IF (EQUAL VARIABLE (CAR (CAR NEW-SUBSTS)))
          (OTHER-SUBSTS-DIFFERENT VARIABLE OLD-SUBSTS (CDR NEW-SUBSTS))
          (IF (EQUAL (CDR (CAR NEW-SUBSTS))
                     (CDR (ASSOC (CAR (CAR NEW-SUBSTS)) OLD-SUBSTS)))
              (OTHER-SUBSTS-DIFFERENT VARIABLE
                                      OLD-SUBSTS
                                      (CDR NEW-SUBSTS))
              (CONS (CAR (CAR NEW-SUBSTS))
                    (OTHER-SUBSTS-DIFFERENT VARIABLE
                                            OLD-SUBSTS
                                            (CDR NEW-SUBSTS)))))))

(DEFUN DMERGE-SECOND-ORDER-SUBSTS (DIFFERENT-SUBSTS DUNIFIED-FORM
                                   SUBSTS-LIST)
  ;; SUBSTS-LIST is a list of subst lists.  On the initial call,
  ;;  this is a singleton list.
  ;; DIFFERENT-SUBSTS is a list of variable names whose substitutions
  ;;  in DUNIFIED-FORM are different than the initial SUBSTS-LIST.
  ;; We merge each of these different substs, in turn, into the
  ;; SUBSTS from SUBSTS-LIST.
  ;; We have SUBSTS-LIST be a list of subst list to accommodate the
  ;; recursive call, since merging can create multiple subst lists,
  ;; and we have to merge succeeding different substs into each.
  (IF (NULL DIFFERENT-SUBSTS)
      SUBSTS-LIST
      (LET ((NEXT-SUBSTS
              (DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST
                (CAR DIFFERENT-SUBSTS) DUNIFIED-FORM SUBSTS-LIST)))
        (DMERGE-SECOND-ORDER-SUBSTS (CDR DIFFERENT-SUBSTS)
                                    DUNIFIED-FORM
                                    NEXT-SUBSTS))))

(DEFUN DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST
       (DIFFERENT-SUBST-VARIABLE DUNIFIED-FORM SUBSTS-LIST)
  (IF (NULL SUBSTS-LIST)
      NIL
      (APPEND (DMERGE-NEW-SUBST
                DIFFERENT-SUBST-VARIABLE
                (CDR (ASSOC DIFFERENT-SUBST-VARIABLE
                            (DUNIFIED-FORM-SUBSTS DUNIFIED-FORM)))
                (CAR SUBSTS-LIST))
              (DMERGE-DIFFERENT-SUBST-INTO-SUBSTS-LIST
                DIFFERENT-SUBST-VARIABLE
                DUNIFIED-FORM
                (CDR SUBSTS-LIST)))))

(DEFUN DREPLACE-SUBST-IN-SECOND-ORDER-SUBSTS
       (VARIABLE DESCRIPTOR SECOND-ORDER-SUBSTS)
  (IF (NULL SECOND-ORDER-SUBSTS)
      NIL
      (CONS
        (DREPLACE-SUBST VARIABLE DESCRIPTOR (CAR SECOND-ORDER-SUBSTS))
        (DREPLACE-SUBST-IN-SECOND-ORDER-SUBSTS
          VARIABLE DESCRIPTOR (CDR SECOND-ORDER-SUBSTS)))))

(DEFUN DINSERT-SUBST (VARIABLE DESCRIPTOR SUBSTS)
  (IF (NULL SUBSTS)
      (LIST (CONS VARIABLE DESCRIPTOR))
      (IF (DESCRIPTOR-ORDER VARIABLE (CAR (CAR SUBSTS)))
```

```
          (CONS (CONS VARIABLE DESCRIPTOR) SUBSTS)
          (IF (EQUAL VARIABLE (CAR (CAR SUBSTS)))
              (INTERNAL-ERROR
                (FORMAT NIL "INSERT-SUBST -- VARIABLE is not supposed t~
o have a binding in SUBSTS~% VARIABLE = ~A~% SUBSTS = ~A~%"
                        VARIABLE SUBSTS)
                NIL)
              (CONS (CAR SUBSTS)
                    (DINSERT-SUBST VARIABLE DESCRIPTOR (CDR SUBSTS)))))))

(DEFUN DREPLACE-SUBST (VARIABLE DESCRIPTOR SUBSTS)
  (IF (NULL SUBSTS)
      (INTERNAL-ERROR
        "DREPLACE-SUBST -- Can't replace a binding that's not there"
        VARIABLE)
      (IF (EQUAL VARIABLE (CAR (CAR SUBSTS)))
          (CONS (CONS VARIABLE DESCRIPTOR) (CDR SUBSTS))
          (CONS (CAR SUBSTS)
                (DREPLACE-SUBST VARIABLE DESCRIPTOR (CDR SUBSTS))))))
```

G.10 SCREEN-VAR-FROM-DESCRIPTOR

```
(DEFUN DUNIFY-VAR-WITH-CONTAINING-DESCRIPTOR
       (VARIABLE DESCRIPTOR SUBSTS TERM-RECS)
  ;; VARIABLE should be a type variable.
  ;; DESCRIPTOR should be a well-formed descriptor.
  ;; SUBSTS should be an alist whose keys are type variables and
  ;;  whose associated values are well-formed descriptors.
  ;; TERM-RECS should be a list of pairs of well-formed descriptors.
  ;;
  ;; This function implements the case of dunify-descriptors
  ;; corresponding to Case 5.3.2 of the proof.
  (LET ((SCREENED-DESCRIPTOR
          (SCREEN-VAR-FROM-DESCRIPTOR VARIABLE DESCRIPTOR)))
    (IF (NOT (EQUAL SCREENED-DESCRIPTOR '*FAIL-SCREEN))
        ;; Termination: Down on size
        (DUNIFY-DESCRIPTORS
          VARIABLE SCREENED-DESCRIPTOR SUBSTS TERM-RECS)
        NIL)))

(DEFUN SCREEN-VAR-FROM-DESCRIPTOR (VARIABLE DESCRIPTOR)
  ;; VARIABLE should be a type variable.
  ;; DESCRIPTOR should be a well-formed descriptor.
  (IF (EQUAL DESCRIPTOR VARIABLE)
      '*FAIL-SCREEN
      (IF (ATOM DESCRIPTOR)
          DESCRIPTOR
          (IF (CONS-DESCRIPTORP DESCRIPTOR)
              (LET ((ARGS (SCREEN-VAR-FROM-DESCRIPTOR-LIST
                            VARIABLE (CDR DESCRIPTOR))))
                (IF (OR (EQL (CAR ARGS) '*FAIL-SCREEN)
                        (EQL (CADR ARGS) '*FAIL-SCREEN))
                    '*FAIL-SCREEN
                    (CONS '*CONS ARGS)))
              (IF (OR-DESCRIPTORP DESCRIPTOR)
                  (LET ((ARGS (REMOVE '*FAIL-SCREEN
                                      (SCREEN-VAR-FROM-DESCRIPTOR-LIST
                                        VARIABLE (CDR DESCRIPTOR)))))
                    (PCANONICALIZE-DESCRIPTOR (CONS '*OR ARGS)))
                  (IF (REC-DESCRIPTORP DESCRIPTOR)
                      (LET ((BODY (SCREEN-VAR-FROM-DESCRIPTOR
                                    VARIABLE (CADDR DESCRIPTOR))))
                        (IF (NOT (EQUAL BODY (CADDR DESCRIPTOR)))
                            (LET ((REC-NAME (MAKE-REC-NAME)))
                              (CANONICALIZE-REC-FOR-DUNIFY
```

```
                                (LIST '*REC
                                      REC-NAME
                                      (SUBST REC-NAME
                                             (CADR DESCRIPTOR)
                                             BODY))))
                            DESCRIPTOR))
                      (IF (RECUR-DESCRIPTORP DESCRIPTOR)
                          DESCRIPTOR
                          (INTERNAL-ERROR
             "SCREEN-VAR-FROM-DESCRIPTOR -- Unexpected descriptor"
                            DESCRIPTOR)))))))

(DEFUN SCREEN-VAR-FROM-DESCRIPTOR-LIST (VARIABLE DESCRIPTOR-LIST)
  (IF (NULL DESCRIPTOR-LIST)
      NIL
      (CONS (SCREEN-VAR-FROM-DESCRIPTOR VARIABLE (CAR DESCRIPTOR-LIST))
            (SCREEN-VAR-FROM-DESCRIPTOR-LIST VARIABLE
                                             (CDR DESCRIPTOR-LIST)))))
```

Appendix H
Supplementary Material Available Via FTP

This appendix provides a complete listing of files supplementary to this report which are available via ftp from server ftp.cli.com and includes instructions on how to retrieve the files.

This postscript for this thesis document is in the file:

```
/pub/akers/inference/report/final-draft.ps
```

The log of a test suite run is in the file:

```
/pub/akers/inference/test/new-function-test-medium-sparc.out
```

The Common Lisp source files for the implementation are:

```
/pub/akers/inference/code/read-me
/pub/akers/inference/code/load-system.lisp
/pub/akers/inference/code/tc-basis.lisp
/pub/akers/inference/code/unify-descriptors.lisp
/pub/akers/inference/code/canonicalize-descriptor.lisp
/pub/akers/inference/code/recognizer.lisp
/pub/akers/inference/code/derive-equations.lisp
/pub/akers/inference/code/solve-equations.lisp
/pub/akers/inference/code/top-level.lisp
/pub/akers/inference/code/theorem-generator.lisp
/pub/akers/inference/code/pcanonicalize-descriptor.lisp
/pub/akers/inference/code/dunify-descriptors.lisp
/pub/akers/inference/code/singleton-interpreter.lisp
/pub/akers/inference/code/contained-in.lisp
/pub/akers/inference/code/checker.lisp
```

The following log illustrates the retrieval of these files from the ftp server.

```
axiom:akers[15]% ftp ftp.cli.com
Connected to jingles.
220 jingles FTP server (Version 5.90 Sat Apr 17 13:50:05 CDT 1993) ready.
Name (ftp.cli.com:akers): anonymous
331 Guest login ok, send ident as password.
Password:john@my.com

230 Guest login ok, access restrictions apply.
ftp> pwd
257 "/" is current directory.
ftp> cd pub/akers/inference
250 CWD command successful.
ftp> get report/final-draft.ps final-draft.ps
200 PORT command successful.
150 Opening ASCII mode data connection for report/final-draft.ps
  (1843972 bytes).
226 Transfer complete.
local: final-draft.ps remote: report/final-draft.ps
1921454 bytes received in 41 seconds (46 Kbytes/s)
ftp> get code/load-system.lisp load-system.lisp
200 PORT command successful.
150 Opening ASCII mode data connection for code/load-system.lisp
  (3695 bytes).
226 Transfer complete.
local: load-system.lisp remote: code/load-system.lisp
3769 bytes received in 0.021 seconds (1.8e+02 Kbytes/s)
```

```
ftp> get code/tc-basis.lisp tc-basis.lisp
200 PORT command successful.
150 Opening ASCII mode data connection for code/tc-basis.lisp
  (74752 bytes).
226 Transfer complete.
local: tc-basis.lisp remote: code/tc-basis.lisp
76718 bytes received in 1.3 seconds (58 Kbytes/s)

...

ftp> disconnect
221 Goodbye.
ftp> bye
axiom:akers[16]%
```

Bibliography

[Abadi 89] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin.
Dynamic Typing in a Statically Typed Language.
In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 213-227. Association for Computing Machinery, Austin, Texas, January, 1989.

[Aho 86] A. Aho, R. Sethi, J. Ullman.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley, 1986.

[Aiken 92] A. Aikin, E. Wimmers.
Solving Systems of Set Constraints.
In *Proceedings of the 1992 IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 1992.

[Akers 86] Robert L. Akers.
Using the Exception Suppression VC Mechanism.
Internal Note 7, Computational Logic, Inc., December, 1986.

[Appel 89] Andrew W. Appel.
Runtime Tags Aren't Necessary.
Lisp and Symbolic Computation 2(2):153-162, June, 1989.

[Backus 78] John Backus.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Communications of the ACM 21(8):613-641, August, 1978.

[Baker 90] Henry Baker.
The Nimble Type Inferencer for Common Lisp 84.
Technical Report, Nimble Computer Corporation, 1990.

[Baker 92] Henry Baker.
A Decision Procedure for Common Lisp's SUBTYPEP Predicate.
Lisp and Symbolic Computation 5:157-190, 1992.

[Beech 70] D. Beech.
A Structural View of PL/I.
Computing Surveys 2(1):33-64, 1970.

[Boehm 85] Hans-J. Boehm.
Partial Polymorphic Type Inference is Undecidable.
In *Proceedings of the 26th Annual Symposium on the Foundations of Computer Science*, pages 339-345. IEEE, October, 1985.

[Boehm 89] Hans-J. Boehm.
Type Inference in the Presence of Type Abstraction.
In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 192-206. Association for Computing Machinery, Portland, Oregon, June, 1989.

[Boyer 72] R.S. Boyer, J Moore.
The Sharing of Structure in Theorem Proving Programs.
Machine Intelligence 7:110-116, 1972.

[Boyer 75] R. S. Boyer and J S. Moore.
Proving Theorems about LISP Functions.
Journal of the Association for Computing Machinery 22(1):129-144, January, 1975.

[Boyer 90] R. S. Boyer and J S. Moore.
A Theorem Prover for a Computational Logic.
In *Proceedings of the 10th Conference on Automated Deduction, Lecture Notes in Computer Science 449*, pages 1-15. Springer-Verlag, 1990.

[Boyer & Moore 79]
R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer & Moore 88]
R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[Burstall 80] R. Burstall, D. MacQueen, D. Sannella.
HOPE: an Experimental Applicative Language.
In *Conference Record of the 1980 LISP Conference*, pages 136-143. Association for Computing Machinery -- SIGPLAN, Stanford, August, 1980.

[Cardelli 84a] Luca Cardelli.
Basic Polymorphic Typechecking.
Technical Report 112, AT&T Bell Laboratories, September, 1984.

[Cardelli 84b] Luca Cardelli.
A Semantics of Multiple Inheritance.
In *Semantics of Data Types, International Symposium, Lecture Notes in Computer Science No. 173*, pages 51-68. Springer-Verlag, 1984.

[Cardelli 85] L. Cardelli, P. Wegner.
On Understanding Types, Data Abstraction, and Polymorphism.
ACM Computing Surveys 17(4):471-522, December, 1985.

[Cardelli 89] Luca Cardelli.
Typeful Programming.
Technical Report 45, DEC Systems Research Center, May, 1989.

[Cartwright 76] Robert Cartwright, Jr.
A Practical Formal Semantic Definition and Verification System for Typed Lisp.
PhD thesis, Stanford, 1976.

[Church 40] Alonzo Church.
A Formulation of the Simple Theory of Types.
The Journal of Symbolic Logic 5(2):56-68, June, 1940.

[Cointe 88] Pierre Cointe.
Towards the Design of a CLOS Metaobject Kernel: ObjVlisp as a First Layer.
In *Lisp Evolution and Standardization*, pages 31-37. IOS, 1988.

[Constable 80] Robert L. Constable.
Programs and Types.
In *21st Annual Symposium on Foundations of Computer Science*, pages 118-128. IEEE Computer Society, Syracuse, New York, October, 1980.

[Constable 86] Robert Constable, et al.
Implementing Mathematics with the Nuprl Proof Development System.
Prentice-Hall, 1986.

[Coppo 80a] M. Coppo, M. Dezani-Ciancaglini, B. Venneri.
Principal Types Schemes and Lambda-Calculus Semantics.
In J.P. Seldin, J.R. Hindley (editor), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 535-560. Academic Press, 1980.

[Coppo 80b] M. Coppo, M. Dezani-Ciancaglini, B. Venneri.
An Extended Polymorphic Type System for Applicative Languages.
In P. Dembinsky (editor), *Mathematical Foundations of Computer Science, Lecture Notes in Computer Science No. 88*, pages 194-204. Springer Verlag, 1980.

[Cormack 90] G.V. Cormack, A.K. Wright.
Type-Dependent Parameter Inference.
In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 127-136. Association for Computing Machinery, White Plains, New York, June, 1990.

[Dahl 66] O. Dahl, K. Nygaard.
SIMULA -- an ALGOL-Based Simulation Language.
Communications of the ACM 9(9):671-678, September, 1966.

[Damas 82] L. Damas, R. Milner.
Principal Type-Schemes for Functional Programs.
In *Proceedings, 1982 ACM Conference on Principles of Programming Languages*, pages 207-212. Association for Computing Machinery, Albuquerque, New Mexico, January, 1982.

[Demers 78] A. Demers, J. Donahue, G. Skinner.
Data Types as Values: Polymorphism, Type-Checking, Encapsulation.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 23-30. Association for Computing Machinery, Tuscon, Arizona, January, 1978.

[Demers 79] A. Demers, J. Donahue.
Revised Report on Russell.
Technical Report TR-79-389, Computer Science Department, Cornell University, 1979.

[Demers 80a] A.J. Demers, J.E. Donahue.
Data Types, Parameters, and Type Checking.
In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 12-23. Association for Computing Machinery, Las Vegas, Nevada, January, 1980.

[Demers 80b] A.J. Demers, J.E. Donahue.
Type-Completeness as a Language Principle.
In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 234-244. Association for Computing Machinery, Las Vegas, Nevada, January, 1980.

[Dietrich 88] R. Dietrich, F. Hagl.
A Polymorphic Types System with Subtypes for Prolog.
In *ESOP '88, 2nd European Symposium on Programming, Lecture Notes in Computer Science No. 300*, pages 131-144. Springer-Verlag, Nancy, France, 1988.

[DoD 83] *Reference Manual for the Ada Programming Language*
United States Department of Defense, 1983.
ANSI/MIL-STD-1815 A.

[Donahue 85] J. Donahue, A. Demers.
Data Types Are Values.
ACM Transactions on Programming Languages and Systems 7(3):426-445, July, 1985.

[Fairbairn 82] J. Fairbairn.
Ponder and Its Type System.
Technical Report 31, University of Cambridge, November, 1982.

[Fairbairn 86] Jon Fairbairn.
A New Type-Checker for a Functional Language.
Science of Computer Programming 6:273-290, 1986.

[Fortune 83] S. Fortune, D. Leivant, M. O'Donnell.
The Expressiveness of Simple and Second-Order Type Structures.
Journal of the Association for Computing Machinery 30(1):151-185, January, 1983.

[Frank 81] Geoffrey A. Frank.
Specification of Data Structures for FP Programs.
In *Proceedings from the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 221-228. Association for Computing Machinery, Portsmouth, New Hampshire, October, 1981.

[Fuh 88] Y. Fuh, P. Mishra.
Type Inference with Subtypes.
In *ESOP '88, 2nd European Symposium on Programming, Lecture Notes in Computer Science No. 300*, pages 94-114. Springer-Verlag, Nancy, France, 1988.

[Giannini 85] Paola Giannini.
Type Checking and Type Deduction Techniques for Polymorphic Programming Languages.
Technical Report CMU-CS-85-187, Carnegie-Mellon University, December, 1985.

[Girard 72] Jean-Yves Girard.
Interpretation Fonctionelle et Elimination des Coupures dans L'arithmetique D'Ordre Superieur.
PhD thesis, University of Paris, 1972.

[Goguen 78] J.A. Goguen, J.W. Thatcher, E.G. Wagner.
An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.
In Raymond T. Yeh (editor), *Current Trends in Programming Methodology, Volume 4*, pages 80-149. Prentice Hall, 1978.

[Good 86a] Donald I. Good, Robert L. Akers, Lawrence M. Smith.
Report on Gypsy 2.05
Computational Logic Inc., 1986.
Revised March 27, 1992.

[Good 86b] Michael K. Smith, Donald I. Good, Benedetto L. DiVito.
Using the Gypsy Methodology
Computational Logic Inc., 1986.
Revised January, 1988.

[Gordon 79] M.J. Gordon, R. Milner, C.P. Wadsworth.
Edinburgh LCF, Lecture Notes in Computer Science, No. 78.
Springer-Verlag, 1979.

[Guttag 81] J. Guttag, J. Horning, J. Williams.
FP with Data Abstraction and Strong Typing.
In *Proceedings from the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 11-24. Association for Computing Machinery, Portsmouth, New Hampshire, October, 1981.

[Hancock 87] Peter Hancock.
Polymorphic Type Checking.
In Simon L. Peyton-Jones (editor), *The Implementation of Functional Programming Languages*. Prentice/Hall International, 1987.

[Henderson 77] Peter Henderson.
An Approach to Compile-Time Type Checking.
In *Information Processing 77 -- Proceedings of IFIP Congress 77*, pages 523-525. North Holland Publishing Company, Toronto, Canada, 1977.

[Hindley 69] R. Hindley.
The Principal Type Scheme of an Object in Combinatory Logic.
Transactions of the American Mathematical Society 146:29-60, December, 1969.

[Hoare 69] C.A.R. Hoare.
An Axiomatic Basis for Computer Programming.
CACM 12-10, 1969.

[Hoare 71] C.A.R. Hoare.
Notes on the Theory and Practice of Data Structuring.
1971.

[Hoare 72] C.A.R. Hoare, D.C.S. Allison.
Incomputability.
Computing Surveys 4(3):169-178, September, 1972.

[Hoare 73] C.A.R. Hoare.
Recursive Data Structures.
Technical Report A.I. Memo 223, Computer Sciences Department, Stanford University, October, 1973.

[Horowitz 84] Ellis Horowitz.
Fundamentals of Programming Languages, Second Edition.
Computer Science Press, 1984.

[Hudak 90] P. Hudak, P. Wadler, et al.
Report on the Programming Language Haskell, Version 1.0
The Haskell Committee, 1990.

[Johnson] Philip Johnson.
Type Flow Analysis for Robust Exploratory Software.
Technical Report, Department of Information and Computer Science, University of Hawaii at Manoa, .

[Kaes 88] Stefan Kaes.
Parametric Overloading in Polymorphic Programming Languages.
In *ESOP '88, 2nd European Symposium on Programming, Lecture Notes in Computer Science No. 300*, pages 131-144. Springer-Verlag, Nancy, France, 1988.

[Kamin 80] Samuel Kamin.
Final Data Type Specifications: A New Data Type Specification Method.
In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 131-138. Association for Computing Machinery, Las Vegas, Nevada, January, 1980.

[Kanellakis 89] P. Kanellakis, J. Mitchell.
Polymorphic Unification and ML Typing.
In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 105-115. Association for Computing Machinery, Austin, Texas, January, 1989.

[Kaplan 80] M.A. Kaplan, J.D. Ullman.
A Scheme for the Automatic Inference of Variable Types.
Journal of the Association for Computing Machinery 27(1):128-145, January, 1980.

[Karttunen 84] Lauri Karttunen.
Features and Values.
In *Proceedings of Coling84, the 10th International Conference on Computational Linguistics*, pages 28-33. Association for Computational Linguistics, Stanford University, California, July, 1984.

[Kasper 86] Robert T. Kasper, W. C. Rounds.
A Logical Semantics for Feature Structures.
In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 257-266. Columbia University, New York, June, 1986.

[Kasper 87] Robert T. Kasper.
A Unification Method for Disjunctive Feature Descriptions.
In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 235-242. USC/Information Sciences Institute, 1987.

[Katayama 84] Takuya Katayama.
Type Inference and Type Checking for Functional Programming Languages --- A Reduced Computation Approach.
In *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, pages 263-272. Association for Computing Machinery, Austin, Texas, August, 1984.

[Kay 79] Martin Kay.
Functional Grammar.
In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistic Society*, pages 142-158. Berkeley Linguistic Society, Berkeley, California, February, 1979.

[Knight 89] Kevin Knight.
Unification: A Multidisciplinary Survey.
ACM Computing Surveys 21(1):93-124, March, 1989.

[Lampson 76] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, G.J. Popek.
Report on the Programming Language Euclid.
Technical Report, Xerox Research Center, August, 1976.

[Landin 64] Peter. J. Landin.
The Mechanical Evaluation of Expressions.
Computer Journal 6(4):308-320, January, 1964.

[Landin 65] Peter. J. Landin.
A Correspondence Between ALGOL 60 and Church's Lambda Notation.
Communications of the ACM 8:89-101,158-165, 1965.

[Landin 66] P. J. Landin.
The Next 700 Programming Languages.
Communications of the ACM 9(3):157-166, March, 1966.

[Laski 68] John Laski.
Sets and Other Types.
ALGOL Bulletin 27:41-48, 1968.

[Leivant 83a] Daniel Leivant.
Structural Semantics for Polymorphic Data Types.
In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 155-166. Association for Computing Machinery, Austin, Texas, January, 1983.

[Leivant 83b] Daniel Leivant.
Polymorphic Type Inference.
In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 88-98. Association for Computing Machinery, Austin, Texas, January, 1983.

[Lewis 73] C.H. Lewis, B. K. Rosen.
Recursively Defined Data Types.
In *Conference Record of ACM Symposium on Principles of Programming Languages*, pages 125-138. Association for Computing Machinery, Boston, Massachusetts, October, 1973.

[Liskov 76] Barbara H. Liskov.
An Introduction to CLU.
Technical Report Computation Structures Group Memo 136, Massachusetts Institute of Technology Laboratory for Computer Science, February, 1976.

[Liskov 78] B. Liskov, E. Moss, C. Schaffert, B. Scheifler, A. Snyder.
CLU Reference Manual, Computational Structures Group Memo 161
MIT Laboratory for Computer Science, 1978.

[Liskov 81] Barbara Liskov.
CLU Reference Manual, Lecture Notes in Computer Science No. 114.
Springer-Verlag, 1981.

[Ma 90] Kwan-Liu Ma, R.R. Kessler.
TICL -- A Type Inference System for Common Lisp.
Software Practice and Experience 20(6):593-622, June, 1990.

[MacQueen 82] D.B. MacQueen, R. Sethi.
A Semantic Model of Types for Applicative Languages.
In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 243-252. Association for Computing Machinery, Pittsburgh, August, 1982.

[MacQueen 84a] David MacQueen.
Modules for Standard ML.
In *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, pages 198-207. Association for Computing Machinery, Austin, Texas, August, 1984.

[MacQueen 84b] D.B. MacQueen, R. Sethi, G.D. Plotkin.
An Ideal Model for Recursive Polymorphic Types.
In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165-174. Association for Computing Machinery, Salt Lake City, January, 1984.

[Martin-Lof 79] Per Martin-Lof.
Constructive Mathematics and Computer Programming.
In *6th International Congress for Logic, Methodology, and Philosophy of Science*. Preprint, University of Stockholm, Department of Mathematics, Hanover, August, 1979.

[Mathlab 82] The Mathlab Group.
Introductory MACSYMA Documentation: A Collection of Papers
Laboratory for Computer Science, MIT, Cambridge, Massachusetts, 1982.

[Mathlab 83] The Mathlab Group.
MACSYMA Reference Manual, Version 10
Laboratory for Computer Science, MIT, Cambridge, Massachusetts, 1983.

[McCarthy 62] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, M.I. Levin.
LISP 1.5 Programmer's Manual
MIT Press, Cambridge, Massachusetts, 1962.

[McCarthy 63] John McCarthy.
A Basis for a Mathematical Science of Computation.
In P. Braffort, D. Hirschberg (editors), *Computer Programming and Formal Systems*, pages 33-70. North-Holland, Amsterdam, 1963.

[McHugh 83] John McHugh.
Towards the Generation of Efficient Code from Verified Programs.
PhD thesis, University of Texas at Austin, December, 1983.

[McPhee 89] Nic McPhee.
Type Checking Lisp, A Preliminary Report.
1989.
private communication.

[Meertens 83] Lambert Meertens.
Incremental Polymorphic Type Checking in B.
In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 265-275. Association for Computing Machinery, Austin, Texas, January, 1983.

[Milner 78] Robin Milner.
A Theory of Type Polymorphism in Programming.
Journal of Computer and System Sciences 17:348-375, 1978.

[Milner 84] Robin Milner.
A Proposal for Standard ML.
In *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, pages 184-197. Association for Computing Machinery, Austin, Texas, August, 1984.

[Mishra 85] P. Mishra, U. Reddy.
Declaration-free Type Checking.
In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7-21. Association for Computing Machinery, New Orleans, January, 1985.

[Mitchell 84] John C. Mitchell.
Coercion and Type Inference (Summary).
In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175-185. Association for Computing Machinery, Salt Lake City, January, 1984.

[Moon 85] D. Moon.
Architecture of the Symbolics 3600.
In *The 12th Annual Symposium on Computer Architecture*. IEEE Computer Society, June, 1985.

[Morris 68] James H. Morris.
Lambda-Calculus Models of Programming Languages.
PhD thesis, MIT, 1968.
Published as MAC-TR-57.

[Morris 73] James H. Morris.
Types Are Not Sets.
In *Proceedings of the 1973 ACM Symposium on Principles of Programming Languages*, pages 120-124. Association for Computing Machinery, Boston, 1973.

[Mycroft 84] A. Mycroft, R.A. O'Keefe.
A Polymorphic Type System for Prolog.
Artificial Intelligence 23:295-307, 1984.

[O'Toole 89] J. O'Toole, D. Gifford.
Type Reconstruction with First-Class Polymorphic Values.
In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 207-217. Association for Computing Machinery, Portland, Oregon, June, 1989.

[Pase 89] Bill Pase and Mark Saaltink.
Formal Verification in m-EVES.
In G. Birtwistle and P.A. Subramanyam (editors), *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 268-302. Springer-Verlag, New York, 1989.

[Pozefsky 78] Mark Pozefsky.
Programming in Reduction Languages.
PhD thesis, University of North Carolina at Chapel Hill, 1978.

[Reynolds 74] John C. Reynolds.
Towards a Theory of Type Structure.
In *Programming Symposium, Proceedings, Colloque sur la Programmation, LNCS 19*, pages 408-425. Springer-Verlag, Paris, 1974.

[Robinson 65] J.A. Robinson.
A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the Association for Computing Machinery 12(1):23-41, January, 1965.

[Rounds 86] W. C. Rounds, Robert T. Kasper.
A Complete Logical Calculus for Record Structures Representing Linguistic Information.
In *Symposium on Logic in Computer Science*, pages 38-43. IEEE Computer Society, Cambridge, Massachusetts, June, 1986.

[Russell 08] Bertrand Russell.
Mathematical Logic as Based on the Theory of Types.
American Journal of Mathematics , 1908.

[Scott 76] Dana Scott.
Data Types as Lattices.
SIAM Journal on Computing 5(3):522-587, September, 1976.

[Shamir 77] A. Shamir, W. Wadge.
Data Types as Objects.
In *Lecture Notes in Computer Science - Automata, Languages and Programming*, pages 465-479. Springer-Verlag, Finland, 1977.

[SmithL 80] Lawrence Mark Smith.
Compiling from the Gypsy Verification Environment.
Master's thesis, The University of Texas at Austin, 1980.
Technical Report 20, Institute for Computing Science.

[Stallman 77] Richard Stallman.
GNU Emacs Manual, Sixth Edition
Free Software Foundation, 1977.

[Steele 84] Guy L. Steele Jr.
Common LISP: The Language.
Digital Press, 1984.

[Steele 90] Guy L. Steele Jr.
Common LISP: The Language, Second Edition.
Digital Press, 1990.

[Steensgaard-Madsen 89]
J. Steensgaard-Madsen.
Typed Representation of Objects by Functions.
ACM Transactions on Programming Languages and Systems 11(1):67-89, January, 1989.

[Strachey 67] C. Strachey.
Fundamental Concepts in Programming Languages.
Lecture Notes: International Summer School in Computer Programming, Copenhagen.
August, 1967

[Symbolics 88] Symbolics, Inc.
Genera Concepts
Symbolics, Inc., 1988.

[Turner 76] D. A. Turner.
SASL Language Manual
Computer Laboratory, University of Kent, 1976.

[Turner 81] D.A. Turner.
The Semantic Elegance of Applicative Languages.
In *Proceedings from the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 85-92. Association for Computing Machinery, Portsmouth, New Hampshire, October, 1981.

[Turner 85] David A. Turner.
Miranda: A Non-Strict Functional Language with Polymorphic Types.
In *Functional Programming Languages and Computer Architecture*, pages 1-16. Springer-Verlag, Nancy, France, 1985.

[VanWijngaarden 69]
A. Van Wijngaarden (ed.), M. Mailloux, J. Peck, C. Koster.
Report on the Algorithmic Language ALGOL 68.
Numerische Mathematik 14(2):79-218, 1969.

[Wadler 89] P. Wadler, S. Blott.
How to Make *Ad Hoc* Polymorphism Less *Ad Hoc*.
In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 60-76. Association for Computing Machinery, Austin, Texas, January, 1989.

[Wand 87] Mitchell Wand.
Complete Type Inference for Simple Objects.
In *Proceedings of the Symposium on Logic in Computer Science*, pages 37-44. Computer Society of the IEEE, June, 1987.

[Wegbreit 74] Ben Wegbreit.
The Treatment of Data Types in EL1.
CACM 17(5):251-264, May, 1974.

[Wirth 71] N. Wirth.
The Programming Language Pascal.
Acta Informatica 1(1):35-63, 1971.

[Wirth 88] Nicholas Wirth.
Type Extensions.
ACM Transactions on Programming Languages and Systems 10(2):204-214, April, 1988.

[Wulf 78] W.A. Wulf, ed., P. Hilfinger, G. Feldman, R. Fitzgerald, I. Kimura, R.L. London, K.U.S. Prasad, V.R. Prasad, J. Rosenberg.
An Informal Description of Alphard.
Technical Report CMU-CS-78-105, Carnegie-Mellon University, 1978.

[Young 90] William D. Young, Robert L. Akers, Bret Hartman, Lawrence M. Smith, Tad Taylor.
Gypsy Verification Environment User's Manual
Computational Logic Inc., 1990.

[Yuasa 85] T. Yuasa, M. Hagiya.
Kyoto Common Lisp Report
IBUKI, 1985.

Index

Table of Contents

List of Figures

List of Tables