A Sound Theorem-Prover for a Higher-order Functional Language

Technical Report
ARC 86-01

Matt Kaufmann

Burroughs Corporation
Austin Research Center
12201 Technology Blvd.
Austin, Texas 78727

January 6, 1986

A SOUND THEOREM-PROVER FOR A HIGHER-ORDER FUNCTIONAL LANGUAGE

MATT KAUFMANN
Burroughs Austin Research Center

Abstract:

The Boyer-Moore theorem-prover has been adapted to the functional language SASL. Examples are presented which range from a primer on this Prover to a summary of a mechanically-checked correctness proof for a SASL unification program. An equally important focus of this paper is on soundness of this Prover. Basic model-theoretic notions are employed to state and prove a soundness theorem for the logic underlying the mechanical Prover. The underlying goal is to show that a non-trivial mechanized system for verifying programs can have a rigorous mathematical underpinning.

No previous acquaintence with SASL or with the Boyer-Moore system is assumed, except for a few isolated technical details which may be skipped by the reader.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification -- *correctness proofs*; D.3.2 [**Programming Languages**]: Language Classifications -- *applicative languages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs -- *logics of programs, mechanical verification*; F.3.1 [**Logics and Meanings of Programs**]: Semantics of Programming Languages -- *denotational semantics*; F.4.1 [**Mathematical Logic and Formal Languages**] -- Mathematical Logic -- *computational logic, lambda calculus and related systems, mechanical theorem proving*.

General terms: Languages, Theory, Verification

Additional key words and phrases: Soundness, directed completeness, pattern-matching, unification

Author's address: Burroughs Corp., Austin Research Center, 12201 Technology Blvd., Austin, TX 78727.

This paper is a report on our work in automated (mechanical) theorem-proving for a functional language. The language is a version of SASL, originally introduced by David Turner and since elaborated into its current version, cf. Richards [14], [15]. Moreover, this language has entered the realm of practicability with the introduction of a prototype NORMA[1] machine at the Austin Research Center (ARC) of Burroughs Corporation. The main objectives of this paper are (1) to report our experiences with a SASL Prover and (2) to demonstrate the soundness of our approach via a rigorous mathematical underpinning. The underlying purpose is to demonstrate the potential for a non-trivial mechanical system for proving theorems about programs which has a rigorous mathematical underpinning.

In November 1984, Robert Boyer consulted for about a month with ARC in adapting the Boyer-Moore theorem-prover (cf. [2]) to SASL. The verification efforts presented herein use the resulting system, which we refer to as the *SASL Prover*. This paper is divided into three parts. The first part is descriptive. It begins with an introduction to SASL, and continues with an introduction to the SASL Prover by way of examples. Part 1 also includes a discussion of the LIFTing principle which enables us to reason more effectively about infinite data structures, and closes with an outline of some of our successful verifications using the SASL Prover. The second part is theoretical. We prove a soundness theorem which demonstrates the validity of the theorems proved by the SASL Prover (assuming that it correctly implements its logic). The proof goes by way of contructing a model appropriate for the SASL Prover, and then defining an appropriate isomorphism between it and an appropriate submodel of the standard SASL domain. Finally, in the third part we consider an alternate approach which eliminates some of the complications in reasoning about the error value "bottom" and also avoids the LIFTing principle. As this approach differs slightly from that used in the first two parts, we also indicate modifications required from the proof in Part 2 in order to prove the corresponding soundness theorem. We illustrate this alternate approach by briefly discussing our proofs of correctness of SASL pattern-matching and unification programs. (See Kaufmann [5], [6] for detailed proofs of correctness of these programs.)

A passing familiarity with Pure Lisp notation is assumed in this paper. Also, some of the more technical details rely on the presentation of a Computational Logic in Boyer-Moore [2], though these are not necessary for a first reading of this paper. Other than these exceptions, this paper is intended to be self-contained for the reader familiar with the rudiments

of logic. In particular, we do *not* assume familiarity with SASL.

A preliminary report on this (and other) work was presented at a talk briefly abstracted in Kaufmann [10].

Acknowledgements. We thank Bob Boyer for a most pleasant collaboration. We also thank our colleagues at Burroughs, especially Carl Pixley, Ed Schneider, and Douglas Surber, for their input on this work.

PART 1: DESCRIPTION OF SASL PROVER AND RESULTS

Here is an outline of this part.

A. Introduction to SASL
B. Introduction to the SASL prover
C. Introduction to the logic of the SASL prover
D. The LIFTing principle
E. Some illustrative examples
F. The correctness of a higher-order mergesort

1.A **Introduction to SASL**

Let us begin with a simple and familiar example: a definition of the factorial function, which we name fact1. (We'll define an alternate version fact2 later.) Notice that the juxtaposition "fact1 x" represents function application.

```
fact1 x = (x=0) -> 1;
          x * fact1(x-1)
```

The meaning of the construct "a -> b; c" is "if a equals TRUE, then return b; if a equals FALSE, then return c; otherwise return ERROR". (Yes, there is actually a value ERROR in SASL, usually called "BOTTOM".) SASL also has list values. For example, the following function returns the last element of a given list. "[]", pronounced "nil", denotes the empty list; hd ("head") returns the first element of a given list; tl ("tail") returns all but the first element of a given list. (Thus hd and tl are analogous to Lisp's car and cdr, respectivelly.)

```
last L = (tl L = []) -> hd L;
         last (tl L)
```

Notice that SASL is really just a sugared (and untyped) lambda calculus with pairing (and with constants for numbers, booleans, and characters)[2]. In fact, in the spirit of untyped lambda calculus, functions can take functions as arguments and return functions as values. For example, the following basic function takes arguments f (generally a function) and L (generally a list) and returns the result of applying f to every member of

L. (Here list is the recognizer for lists.)

```
map f L = (L=[]) -> [];
          (list L) -> f(hd L) : map f (tl L);
          BOTTOM
```

Functions in SASL are actually *curried*: that is, they are all really functions of one argument. For example, if we define

```
plus x y = x+y
```

and let L be the list [3,4,5] (i.e. the list with elements 3, 4, and 5 in that order), then

```
map (plus 7) L
```

denotes the list [10,11,12], since (plus 7) denotes the function which adds 7 to any argument.

SASL allows infinite lists, such as the list of integers from n on, defined as follows. Here, ":" is the list construction operation, analogous to Lisp's "cons" (and pronounced the same): thus, hd(x:y) = x, and for lists y we also have tl(x:y) = y. Unlike Lisp, the tail of (x:y) is BOTTOM if y is not a list; such a list is said to *terminate with BOTTOM*, rather than *with NIL*. Notice that ":" binds more loosely than juxtaposition. Thus the list of integers from n on is simply (from n), where:

```
from n = n : from(n+1) .
```

Here is a group of SASL definitions, the first of which defines the (infinite) list of primes, in increasing order.[4]

```
primes = sieve (from 2)
from n = n : from(n+1)
sieve L = hd L : sieve (filter (hd L) (tl L))
filter a L = ((hd L) REM a = 0) -> filter a (tl L);
             hd L : filter a (tl L)
```

Infinite lists such as primes are useful in finite computations. For example, the expression "primes 100" denotes the 100th prime in the context of the definitions above, as juxtaposition denotes list

subscription rather than function application when the first expression denotes a list. (The paper Hughes [4] gives a nice account of how to use infinite data structures, and more generally of applicative languages such as SASL.)

As the reader may have guessed by now, the following principle holds: *to every set of SASL definitions corresponds SASL objects for which those definitions hold* ([8, §8, Lemma D6]).

1.B **Introduction to the SASL prover**

The Prover begins with an initial database of facts about SASL, numbers, booleans, and so on. The user can extend this database by adding definitions and by instructing the Prover to (attempt to) prove theorems. There are two ways to add definitions. The DEFN command has the following syntax:

```
(DEFN function_name argument_list
  body)
```

Every function symbol occurring in **body** must be in the initial SASL database or have already been defined by the user, except that **function_name** itself may occur in **body**, i.e. recursion is permitted. If the function is recursive, then the Prover will attempt to prove that the function is total, i.e. it always returns a value (for all choices of arguments), by proving that some function of the arguments always decreases according to some well-founded relation. (See Chapter III of Boyer-Moore [2] for details.) If the Prover cannot do so, then it rejects this definition. Now many recursively defined SASL functions are not total, such as from defined above. However, there is another command that one may use for SASL functions, which is SASL-DEFN. This command has the same syntax as DEFN. The difference is that it checks that every function called in **body** (except the one being defined) is a SASL function, i.e. either belongs to a given predefined set of known SASL functions (the SASL primitives, defined in Subsection 2.B(iii)) or else is one which was previously defined using SASL-DEFN. (One added technical point: the function symbol RENAME gets special treatment, namely, if the term (RENAME 'f) occurs in **body**, then the function symbol f must either be the one being defined or must be a SASL function in the above sense.) If that check succeeds, then this definition is added to the database. The rationale here is that the semantics of SASL guarantee that every such definition has a solution, as mentioned above. Finally, we note that for

both the DEFN and SASL-DEFN commands, the function being defined may not occur in the initial SASL database nor may it have been previously defined by the user; otherwise the Prover will reject the definition.

The command to prove a theorem has the following syntax:

```
(PROVE-LEMMA theorem_name theorem_types
   theorem_statement
   hints)
```

where **hints** is optional, and **theorem_types** is "(REWRITE)" if the user wants the system to use this theorem as a rewrite lemma in the future, and otherwise is generally "NIL".

In Appendix 1, which we suggest as the reader's next stop, these commands are used to prove that for all non-negative integers n, fact1 n = fact2 1 n where fact1 is defined above and fact2 is defined using an accumulating parameter:

```
fact2 acc n = (n=0) -> acc;
              fact2 (acc*n) (n-1)
```

This and all definitions, however, must be given to the prover in Lisp syntax, without infix operators.[3] Our convention has been to prefix SASL functions with "SASL-"; for example, the subterm "acc*n" appears as "(SASL-TIMES acc n)".

1.C **Introduction to the logic of the SASL prover**

In this section we say more about the initial SASL database of facts. The commands used to create this database are to be found in Appendix 2. We allow ourselves to speak informally about the basic Boyer-Moore Computational Logic, as details may be found in Chapter III of [2].

The logic allows for so-called *shells*, which are really types and may be defined inductively. The intuitive idea can be given informally as follows. The NUMBERP shell contains all the non-negative integers, while the NEGATIVEP shell contains the rest. The TRUEP and FALSEP shells each contain one (boolean) value. The FINLISTP shell contains every hereditarily finite list, i.e. every list whose head is either a hereditarily finite list or is not a list at all and whose tail is a hereditarily finite list or is BOTTOM, henceforth spelled BTM and put by itself in the BTMP shell. The LISTP shell contains the rest of the lists. Finally, the

LITATOM shell holds functions. (There is no shell for characters; not every object needs to belong to a shell.) Let us now get a bit more precise.

A shell comes with a *recognizer*; a *constructor*; an (optional) *bottom object*; and for each argument of the constructor an *accessor* (roughly, inverse to the constructor), a *type restriction*, and a *default value*. For example, one built-in shell is the one with recognizer NUMBERP which contains the non-negative integers. The *constructor* is ADD1, which means that NUMBERP returns **T** (TRUE) on exactly those objects of the form (ADD1 x) together with the optional *bottom object*, which for this shell is **0**. The *accessor* is (not unexpectedly!) SUB1. The *type restriction*[5] for the argument to ADD1 is that the argument be a NUMBERP; if not, then that argument is coerced to the *default value*, which happens to be the "bottom object" **0** of the NUMBERP shell. For example, **(ADD1 T)** = **(ADD1 0)**, usually denoted **1**. (Similarly, **(ADD1 1)** is abbreviated by **2**, **(ADD1 2)** by **3**, and so on.) Here then are the built-in shells. We refer to them by their recognizers. Notice our convention of using boldface for terms.

- NUMBERP, as defined above.

- NEGATIVEP, with constructor MINUS, no bottom object, accessor NEGATIVE-GUTS, type restriction NUMBERP, and default value 0. (Note: shells are disjoint, and hence (MINUS 0)≠0, which is of course unfortunate. We'll partially rescue ourselves from this anomaly in Part 2. However, notice that the function SASL-DIFFERENCE defined in Appendix 2 never returns the value **(MINUS 0)**.)

- TRUEP, with constructor TRUE, no bottom object, and no accessors, type restrictions or default values since TRUE takes no arguments. As in Boyer-Moore [2], we abbreviate **(TRUE)** by **T**.

- FALSEP, defined analogously to TRUEP.

- LITATOM, which takes one argument, has bottom object NIL, and has no type restriction -- and that's all we need to know for now about this shell.

- LISTP, with constructor CONS, no bottom object, accessors CAR and CDR and with no type restrictions and default value of NIL for both accessors.

WARNING:

CAR and CDR are not used by the SASL Prover -- they have *nothing* to do with SASL's hd and tl functions.

We also add the following shells to create the SASL world. The reader may wish to compare these descriptions with the corresponding ADD-SHELL events in the beginning of Appendix 2. (By "events" we mean commands to the Prover.)

- BTMP, with constructor BTM taking no arguments, and that's that (as for TRUEP).

- FINLISTP, with constructor FINPAIR, bottom object SASL-NIL, accessors FINHD and FINTL, type restriction of non-LISTP and default value of BTM for the first argument, and type restriction that the second argument must be a BTMP or a FINLISTP with default value of BTM.

Notice that if the second argument of a FINPAIR is not a list, it is coerced to BTM; for example, **(FINPAIR 2 3) = (FINPAIR 2 BTM)**. This accords with the behavior of SASL "cons", which is not strict in either of its arguments; that is, a:b is a list for all a, b (and is never BOTTOM).

Henceforth we refer to the set of all shells defined above as *the set of basic SASL shells*. The sense in which these shells correspond to sets of SASL values will be made precise in Part 2.

A number of SASL facts are also included in the initial SASL library (Appendix 2). The DCL command is used to declare a function; that is, DCL is like DEFN except that the body is omitted. The ADD-AXIOM command is like the PROVE-LEMMA command, except that the theorem-prover accepts the given statement without proof. Hence, it is important to verify such statements by hand, as we do (at least in outline) in Part 2. Working through Appendix 2, one sees that the SASL functions (SASL-) cons, hd, and tl are defined using the functions associated with the FINLISTP shell unless one of the arguments is an infinite list structure (i.e. a LISTP), in which case the declared functions CONS-INF, HD-INF, and TL-INF are used. Subsequent axioms explain the relationships among all these functions. As remarked above, CAR and CDR are not used here at all. Rather, the LISTP shell is merely used as a convenient receptacle for the infinite list structures, since it has to be used for something anyhow (as it's built into the guts of the Prover).

Functions are currently handled in a rather awkward way. Recall that

the LITATOM shell is used to hold the functions (in a nonconstructive way, much as the LISTP shell holds the infinite list structures). Early on in Appendix 2 we declare a function AP, which is intended to be function application. The behavior of AP is explained by the axioms suffixed "REP", such as:

```
(ADD-AXIOM SASL-HD-REP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-HD) X)
                  (SASL-HD X)))
```

(Here, 'SASL-HD is actually a LITATOM obtained by continually applying the constructor for LITATOM to the ASCII characters of "SASL-HD", but that's not important.) The idea of this axiom would be clear if "(RENAME 'SASL-HD)" were simply replaced by "'SASL-HD". The function RENAME is introduced for the following reason. First we note that such axioms are added whenever a function is defined or declared by the user. Now suppose that the user defines his own function, say FOO, which also happens to be the same function as SASL-HD. Intrinsic to the Boyer-Moore logic is the assumption that the shell constructors are one-to-one (if there are no type restrictions), and hence 'SASL-HD ≠ 'FOO. However, because we wish to prove function equality in some cases, we have an extensionality axiom near the end of Appendix 2 which says that if two functions act the same on all arguments, then they are equal. Hence we conclude 'SASL-HD = 'FOO, contradicting the conclusion just reached above. The correct conclusion is that **(RENAME 'SASL-HD) = (RENAME 'FOO)**, which is proved using extensionality together with the axiom for 'SASL-HD displayed above along with the corresponding axiom for 'FOO.

Axioms such as the above "REP" axioms are also added automatically whenever the user invokes the SASL-DEFN command. Suppose the user adds a definition by:

```
(SASL-DEFN FOO (X)
  (BAR X))
```

The the system adds the following additional axiom automatically.

```
(FOO X) = (AP (RENAME 'FOO) X)
```

If there are two arguments, then the added axiom reflects the currying mentioned in Section A: given

```
(SASL-DEFN FOO2 (X Y)
   (BAR2 X Y))
```

the system adds

```
(FOO2 X Y) = (AP (AP (RENAME 'FOO2) X) Y)
```

In addition, the system also adds axioms saying that partial applications are functions. That is, given a definition

(SASL-DEFN NAME (X_1 X_2 ... X_n)
   BODY)

the system adds axioms

(LITATOM (AP (AP (... (AP (RENAME 'NAME) X_1) ...) X_{i-1}) X_i)

for $1 \le i < n$. (Recall that LITATOM is the recognizer for functions.)

The reader is welcome to read through Appendix 2, though the interesting parts are commented on above already. We'll return to Appendix 2 in Section 2.B, where we will consider soundness. One final remark: the function symbol AP-OUTER corresponds to juxtaposition, so that (AP-OUTER x y) denotes the application of x to y (i.e. (AP x y)) if x is a function, but denotes (SUBSC x y), i.e. list subscription, if x is a list.

As mentioned at the start of this section, a comprehensive account of the basic logic can be found in Boyer-Moore [2]. We remark however that (6) and (7) on page 39 of [2] (establishing a well-founded relation for each shell) are ignored in this paper, since there is a fixed set of well-founded relations used in the implementation, namely, the standard order on the non-negative integers and its two-fold and three-fold lexicographic products.

1.D **The LIFTing principle**

We first illustrate this principle by way of an example. Suppose one wants to prove the associativity of the SASL function append, which can be defined as follows:

```
append x y = ~list x -> BOTTOM;
             x=[] -> (list y -> y; BOTTOM);
             hd x : append (tl x) y
```

We have found that the following approach often works fairly well. The idea is to prove the given theorem first under the assumption that certain of the variables do not denote infinite data structures. Here is the statement of that theorem:

```
append (append x y) z = append x (append y z)
    for x, y, z not infinite data structures (i.e., not LISTPs)
```

It's easy to see from the definition of append that this theorem holds if x is not a list, i.e. if (list x) ≠ T. So let us informally prove the result for x a finite list, by induction on the structure of x. If x=[], then the theorem clearly holds. Assuming the theorem holds for x, we prove the result for a:x as follows:

```
  append (append (a:x) y) z)
= append (a : append x y) z
= a : append (append x y) z
= a : append x (append y z)     (by the inductive hypothesis)
= append (a:x) (append y z)
```

Now that we know that the theorem holds for non-LISTPs, we may use the LIFTing principle to conclude that the theorem holds without this restriction. A formal statement of the LIFTing principle can be found in Section 2.C. For now, we simply note that it includes the following. Let L be a list of variables. Suppose that t1 and t2 are SASL terms and that the equation [t1 = t2] is valid under the assumption that the variables in L are not LISTPs. Then [t1 = t2] is valid without that assumption. Intuitively, any disagreement between these terms would show up after some finite amount of computation which only refers to finite parts of the variables in L. Again, we'll be precise about this in Part 2 (specifically, in Lemma 2.4).

Let us now show the commands given to the Prover in order to verify the associativity of append. The strategy that we use here is fairly typical of several proofs that we have carried out on the Prover. We define not only the SASL append function, SASL-APPEND, but we also define a "finite analogue" APPEND which is intended to agree with the

SASL version on all x, y, and z which are not LISTPs. The advantage of this approach is that since the Prover is presented with the finite version APPEND using the DEFN command, it generates an induction principle in the course of accepting that definition which is used to prove the associativity of APPEND automatically. In conjunction with the theorem APPEND-IS-APPEND which asserts the equivalence of the SASL append function with its finite analogue (when the variables are not LISTPs), it is then able to conclude the associativity of SASL-APPEND (see "SASL-APPEND-ASSOC") under this same assumption. Finally, it LIFTs the result to apply to all x, y, z. Here, then, are the events given to the Prover. First, the definitions of the SASL append function and its "finite analog":

```
(SASL-DEFN SASL-APPEND (X Y)
      (SASL-IF (SASL-NOT (SASL-LIST X))
      (BTM)
      (SASL-IF (SASL-EQUAL X (SASL-NIL))
               (SASL-IF (SASL-LIST Y) Y (BTM))
               (SASL-CONS (SASL-HD X) (SASL-APPEND (SASL-TL X) Y)))))

(DEFN APPEND (X Y)
      (IF (FINLISTP X)
         (IF (EQUAL X (SASL-NIL))
            (IF (FINLISTP Y) Y (BTM))
            (FINPAIR (FINHD X)
                     (APPEND (FINTL X) Y)))
         (BTM)))
```

Now our plan is to first prove that SASL append is associative when restricted to non-infinite structures. To that end, we prove a lemma which asserts that under this restriction, the SASL append function is the same as its finite analogue.

```
(PROVE-LEMMA APPEND-IS-APPEND (REWRITE)
      (IMPLIES (AND (NOT (LISTP X))
                    (NOT (LISTP Y)))
               (EQUAL (SASL-APPEND X Y)
                      (APPEND X Y))))
```

The Prover proves this automatically, choosing to use the induction principle generated when the definition of APPEND was accepted. Next, it

uses that induction to prove the associativity of the finite analogue, followed by easy rewriting (with the lemma just given) to yield the result for the SASL append function restricted to non-infinite structures.

```
(PROVE-LEMMA ASSOC-APPEND (REWRITE)
      (EQUAL (APPEND (APPEND X Y) Z)
             (APPEND X (APPEND Y Z))))

(PROVE-LEMMA SASL-APPEND-ASSOC (REWRITE)
  (IMPLIES (NLISTP X)
           (IMPLIES (NLISTP Y)
                    (IMPLIES (NLISTP Z)
     (EQUAL (SASL-APPEND (SASL-APPEND X Y) Z)
            (SASL-APPEND X (SASL-APPEND Y Z)))))))
```

Finally, we would like to remove the assumption that X, Y, and Z are not infinite structures (LISTPs), as described above. The following LIFT command is the one that carries out the final step, i.e. the removal of the finiteness assumption. It has the following syntax:

```
(LIFT lifted_theorem_name theorem_types source_theorem_name
      variables)
```

where **lifted_theorem_name** is the name of the resulting theorem, **theorem_types** is as before (usually "(REWRITE)" or "NIL"), **source_theorem_name** is the name of the finite version that has already been proved, and **variables** is a list of variables which have been assumed to be non-**LISTP**s in the source theorem. A syntax check is made to assure that the source theorem has a syntactic form appropriate for the application of LIFTing, as described in Section 2.C. In particular, equations between terms, all of whose functions are SASL functions, are allowed. In the present example, then, we give the command

```
(LIFT SASL-APPEND-ASSOC-LIFTED (REWRITE) SASL-APPEND-ASSOC
      (X Y Z))
```

This command then adds a theorem named SASL-APPEND-ASSOC-LIFTED to the database, with statement

```
     (EQUAL (SASL-APPEND (SASL-APPEND X Y) Z)
            (SASL-APPEND X (SASL-APPEND Y Z)))
```

and the proof is complete.

The LIFTing principle would not be valid if applied to arbitrary theorems. Consider for example the following statement:

(NOT (EQUAL X (1:X)))

This statement is true for all non-LISTPs. However, the infinite list of ones, defined by

X = 1:X ,

shows that the statement above can fail for infinite lists. In fact, we'll see in the proof of Lemma 2.4 that the "NOT" above is the source of the problem, and our syntactic restrictions for use of the LIFTing principle will treat "NOT" very carefully.

1.E **Some illustrative examples**

Appendix 3 contains a sequence of events (commands to the Prover) which ran successfully and contain several theorems. These were essentially reported in Boyer-Kaufmann [1]; the only difference is that the first six events of Appendix 2 in [1] have been replaced in our latest run by the six events presented above (which use a *correct* version of SASL append).

The first group of events in Appendix 3 culminates in THEOREM-SASL-GLUE-LIFTED, which (informally) asserts that if L is a finite tree with character strings at the leaves, then the appropriate flattening of that tree is also a character string. Since the SASL function tree (which recognizes trees) is defined in terms of the higher-order function all, and since the Boyer-Moore system is not designed explicitly to handle such functions, the finite analogue of tree (namely, AUX-TREE) is defined using an extra parameter "FLAG", which indicates whether one is looking at the head or the tail of the tree. The flattening function, SASL-GLUE, also has such a finite analogue with an extra parameter "FLAG". In fact, as the lemma AUX-GLUE-IS-GLUE-AND-MAP-GLUE shows, **(AUX-GLUE T)** is really SASL-GLUE while **(AUX-GLUE F)** is really **(MAP SASL-GLUE)**.

The next group of events is a mechanical proof of a theorem mentioned in Turner [19], namely, if L is an infinite list then append L K = L for all K. The function REAL-LENGTH is defined in the initial database of SASL

facts (near the end of Appendix 2), and is intended to give the "real length" of an arbitrary list (where an infinite list has real length BTM, but a list of the form $(x_1 : \ldots : x_n : \mathbf{BTM})$ has real length n rather than (SASL) length BTM). The function INF-LIST is defined after REAL-LENGTH, and is intended to recognize infinite lists. The proof uses these two functions along with a finite analogue LEN of the SASL length function.

The third group of events culminates in a statement that every list whose length does not equal BTM is the reverse of its reverse.

Unlike the first three groups, the final group of events does not involve the lifting principle. It does however involve infinite lists, as it culminates in the statement that $(\text{from } n)_K = n+k-1$. (**from** is defined in section A.)

A more substantial example is a proof of correctness of a SASL quicksort function, as reported in Kaufmann [7]. There were few surprises in that effort, however, so let us move on to consider a sorting program which takes the ordering as a parameter. We happened to choose mergesort for this purpose.

1.F **The correctness of a higher-order mergesort**

To save space we omit the detailed list of events leading to the proof of correctness of a higher-order mergesort, but instead summarize that proof. First, here are the main SASL definitions. Notice that the first parameter lt of mergesort is intended to be a binary, boolean-valued function -- usually, the characteristic function of a total order. However, the main theorem will be general enough to allow lt to be an arbitrary binary function. The auxiliary function merge takes two lists which are already sorted according to its parameter lt and merges them together to produce a sorted list.

```
mergesort lt L =
  ~(list L) -> BOTTOM;
  L=[] -> [];
  tl L = [] -> L;
  merge lt (mergesort lt (odds L))
           (mergesort lt (odds (tl L)))
```

```
merge lt L M =
  ~(list L & list M) -> BOTTOM;
  L=[] -> M;
  M= [] -> L;
  lt (hd L) (hd M) -> hd L : merge lt (tl L) M;;
  hd M : merge lt L (tl M)

odds L =
  L=[] -> [];
  tl L = [] -> L;
  hd L : odds (tl (tl L))
```

Next, we present the main functions given to the Prover with the DEFN command. They are used to state the theorems. The first of these is a rather unusual definition of what it means for a list to be sorted with respect to LT. The idea is that for consecutive members x, y of the list, either [x LT y] is true or else at least [y LT x] is false. (That is, it's OK and perhaps even unavoidable that both [x LT y] and [y LT x] are false, if LT is not a total order.) It's easy to see that if LT is indeed a total order, then this definition agrees with the usual one. (In fact a lemma has been proved to this effect for the cases that LT is ordinary '<' or '≤'.)

```
(DEFN SORTED (LT L)
 (IF (NOT (FINLISTP L))
     T
     (IF (EQUAL L (SASL-NIL))
         T
         (IF (EQUAL (FINTL L) (SASL-NIL))
         T
         (AND (OR (EQUAL (AP-OUTER (AP-OUTER LT (FINHD L))
                                   (FINHD (FINTL L)))
                         T)
                  (EQUAL (AP-OUTER (AP-OUTER LT (FINHD (FINTL L)))
                                   (FINHD L))
                         F))
              (SORTED LT (FINTL L)))))))
```

The next definition is for a function PFINLISTP (*Proper* FINLISTP), which returns **T** on any FINLISTP which terminates with NIL, i.e. any list of the form a1:a2:...:ai:[], and **F** on everything else.

```
(DEFN PFINLISTP (L)
  (IF (NOT (FINLISTP L))
  F
  (IF (EQUAL L (SASL-NIL))
      T
      (PFINLISTP (FINTL L)))))
```

The function OCCURRENCES checks to see how many times some X occurs in a given finite list L.

```
DEFN OCCURRENCES (X L)
  (IF (NOT (FINLISTP L))
      0
      (IF (EQUAL L (SASL-NIL))
          0
          (IF (EQUAL X (FINHD L))
              (ADD1 (OCCURRENCES X (FINTL L)))
                    (OCCURRENCES X (FINTL L))))))
```

There are also finite analogues defined for MERGESORT, MERGE, and ODDS; as before, we will refer to the SASL versions by prefixing them with "SASL-". Here are the main theorems proved about MERGESORT. As both of these results involve non-SASL functions, we do not attempt to lift them in any way; besides, mergesort doesn't terminate on infinite lists anyhow.

1. Mergesort returns a sorted list:

```
(IMPLIES (PFINLISTP L)
         (EQUAL (SORTED LT (SASL-MERGESORT LT L)) T))
```

2. Mergesort returns a permutation of the given list, if it "halts":

```
(IMPLIES (AND (NLISTP L)
                  (EQUAL (SASL-PFINLISTP (SASL-MERGESORT LT L)) T))
         (EQUAL (OCCURRENCES X (SASL-MERGESORT LT L))
                (OCCURRENCES X L)))
```

The second result may appear weak because of the second hypothesis, but the idea is to remove it in particular cases. We tried this out for the case of finite lists of nonnegative numbers (NUMBERPs) ordered by ordinary integer '<'. Let NUMLIST be the recognizer for such lists:

```
(DEFN NUMLIST (L)
  (IF (NOT (FINLISTP L))
      F
      (IF (EQUAL L (SASL-NIL))
          T
          (AND (NUMBERP (FINHD L))
               (NUMLIST (FINTL L))))))
```

The following results were then proved, using the general results and their lemmas.

3. Mergesort on a NUMLIST returns a NUMLIST:

```
(IMPLIES (NUMLIST L)
         (NUMLIST (MERGESORT (RENAME 'SASL-LT) L)))
```

4. Mergesort returns a permutation of a NUMLIST:

```
(IMPLIES (NUMLIST L)
     (EQUAL (OCCURRENCES X (SASL-MERGESORT (RENAME 'SASL-LT) L))
            (OCCURRENCES X L)))
```

PART 2: SOUNDNESS

In this part we provide a theoretical basis for the Prover discussed in Part 1. In Section A we construct term models for the Prover, in a general setting not specific to the SASL modification of the original Boyer-Moore system. Moreover, we state a useful lemma showing soundness in a general sense, and we indicate how to extend interpretations (as defined below) when presented with a DEFN event. In Section B we specialize to our (SASL) Prover, showing how to build interpretations corresponding to sequences of events. Section C contains a proof of the validity of LIFT events for directedly complete propositions. In Section D we consider the type set computations which take place at the time of SASL-DEFN events. Finally, everything is in place to conclude with the soundness theorem in Section E. Throughout this part, we emphasize what is new about this Prover relative to the original version described in [2]. However, we do give definitions and details for that version as well, when appropriate.

The logic we use is simply ordinary first-order logic, restricted to universal (and quantifier-free) sentences, embellished as described below (with LIFT events, for example). In particular, there is no need for any flavor of Hoare triples or the like, as often used for reasoning about programs in imperative languages.

2.A **A "term model" for the Prover; interpretations**

In this section we construct a *term model* for the Prover which is to be used to prove a soundness theorem demonstrating the validity of our approach. Indeed, this section is independent of SASL; it provides a model for the original Boyer-Moore Theorem-Prover as well (relative to any given set of shells). Some familiarity with Boyer-Moore [2] may be helpful in this section for a few of the technical details.

Fix a sequence S of shells and a set AT of *atoms*, and let B be the union of AT with the set of bottom objects of S. We define the *term model* M to be the set of all *proper terms* built up from B using the constructors of S: that is, every element of B is a proper term, and if **f** is a constructor of S taking n arguments and $t_1, ..., t_n$ are proper terms which meet the respective type restrictions of f, then $(\mathbf{f}\ t_1\ \ldots\ t_n)$ is a proper term. (It is clear what it means for a type restriction *tr* to hold of t: a type restriction corresponds to a collection of shells, one of whose constructors or bottom objects must be the leading function symbol of t in order for *tr* to hold of t. As in [2], we think of the elements of

AT as lying in a shell OTHER.) The class of *terms* is defined similarly, but without the type restrictions and with variables allowed as well. In particular, a variable is a term.

Now M as defined above is a universe corresponding to a set of shells, but we still need to consider a setting in which one has functions on that universe which correspond to the DEFN events submitted to the Prover. An *interpretation for* (an arbitrary set) A is a mapping from a set of function symbols (each tagged with an arity) to functions on A (each with the appropriate arity). (Most logic textbooks also allow relation symbols, but we will not need them.)

Let us define the *basic interpretation* $\mathcal{I}_0$ (relative to a fixed sequence S of shells) for M to be the interpretation for M whose domain is the set of all function symbols of members of S together with several other function symbols, defined as follows. For each given shell, let **CONST** be its constructor, **b** its (optional) bottom object, **R** its recognizer, $\mathbf{AC_i}$ ($1 \le i \le n$) its accessors, $\mathbf{tr_i}$ ($1 \le i \le n$) its type restrictions, and $\mathbf{dv_i}$ ($1 \le i \le n$) its default values. (Note the close tie with pp. 38-39 of [2].) The function symbol COUNT is intended to denote a function which measures roughly the maximum depth of the constructors in a given term.

Let $\mathbf{r} = \mathcal{I}_0$ R, $\mathbf{const} = \mathcal{I}_0$ **CONST**, and $\mathbf{ac_i} = \mathcal{I}_0$ $\mathbf{AC_i}$, let $\mathbf{t_i}$ ($1 \le i \le n$) be arbitrary elements of M, and let count = $\mathcal{I}_0$ COUNT. Also let $\mathbf{t_i}^* = \mathbf{t_i}$ if $\mathbf{tr_i}$ holds of $\mathbf{t_i}$; otherwise $\mathbf{t_i}^* = \mathbf{dv_i}$. Notice that $\mathbf{(CONST\ t_1^*\ \ldots\ t_n^*)}$ is the form of an arbitrary element of M - AT. Then we define:

$\mathbf{r\ (CONST\ t_1^*\ \ldots\ t_n^*)} = T$

$\mathbf{r\ b} = T$ (if **b** exists)

$\mathbf{r\ (CONST'\ t_1\ \ldots\ t_n)} = F$ for **CONST'** a constructor of a different shell from that of **CONST** and $\mathbf{(CONST'\ t_1\ \ldots\ t_n)} \in M$

$\mathbf{r\ b'} = F$ for **b'** a bottom object of another shell

$\mathbf{r}\ a = F$ for $a \in AT$

$\mathbf{const}(t_1,\ldots,t_n) = \mathbf{(CONST\ t_1^*\ \ldots\ t_n^*)}$

$\mathbf{ac_i\ (CONST\ t_1^*\ \ldots\ t_n^*)} = \mathbf{t_i}^*$

count $\mathbf{(CONST\ t_1^*\ \ldots\ t_n^*)}$ = count $(\mathbf{t_1}^*)$ + ... + count $(\mathbf{t_n}^*)$ + 1

count (**b**) = 0

count (a) = 0 for $a \in AT$

Also $\mathcal{I}_0$ (EQUAL) is the characteristic function of equality, and $\mathcal{I}_0$ behaves the expected way on the other built-in functions of the Prover (IF, AND, OR, NOT, LESSP, PLUS, and so on).

For a variable-free term **t** and an interpretation $\mathcal{I}$, we say that **t** *holds in* $\mathcal{I}$ if $\mathcal{I}(\mathbf{t}) \neq \mathbf{F}$. (So if the leading function symbol of **t** is EQUAL or a boolean connective, then we know that $\mathcal{I}(\mathbf{t})$ is actually **T**.) More generally, a term **t** *holds in* $\mathcal{I}$ if for every variable-free term $\mathbf{t}^*$ resulting from substituting for the variables in **t**, $\mathbf{t}^*$ holds in $\mathcal{I}$. The following lemma will be useful in our proof of soundness. It follows from the fact that the axioms clearly hold in $\mathcal{I}$ and of course, propositional consequence preserves "holding in" $\mathcal{I}$. (We omit the details.)

Lemma 2.1 (soundness of the logic). Fix an interpretation $\mathcal{I}$ for the term model M and let S be any collection of terms which hold in $\mathcal{I}$. Let **t** be a term which is provable in the Boyer-Moore Computational Logic using as axioms terms which hold in $\mathcal{I}$. Then **t** holds in $\mathcal{I}$. □

In the course of carrying out a proof, one generally extends the "environment" with DEFN and SASL-DEFN commands. We'll deal with SASL-DEFN in the next section; for now, let us consider DEFN commands. By the argument on pages 46 to 51 of [2], every DEFN command defines a unique function:

Lemma 2.2 (extension by DEFN events). Let $\mathcal{I}$ be an interpretation for the term model M and let **def** be a definition $\mathbf{f}\ \mathbf{x}_1 \ldots \mathbf{x}_n = \mathbf{body}$, where **f** is not in the domain of $\mathcal{I}$ but every other function symbol of **body** is in the domain of $\mathcal{I}$. Then there is a unique interpretation extending $\mathcal{I}$ by adding **f** to its domain such that **(EQUAL (f** $\mathbf{x}_1 \ldots \mathbf{x}_n$**) body)** holds. Henceforth we refer to this extension as *the extension of* $\mathcal{I}$ *via* **def**. □

2.B **Interpretations corresponding to sequences of events**

Our basic strategy for proving soundness is as follows. One sets up a one-to-one correspondence between a term model and an appropriate subset of the SASL domain, namely one which has the same first-order properties as the SASL domain. By viewing the term model alternatively as a SASL domain, one can make sense of SASL-DEFN events. We then argue the soundness of LIFT events and of the SASL-DEFN type set computation in Sections 2.C and 2.D.

Here are the stages in which we assign interpretations to sequences of events.

(i) Defining an appropriate correspondence between two universes
(ii) Interpreting (on M) the declared functions of the initial database: the initial interpretation
(iii) Extending the initial interpretation according to SASL-DEFN events
(iv) Summary

(i) *Defining two universes and an appropriate correspondence between them*

Let AT be the set of SASL characters, except that one character is omitted. (That character will be mapped to **(MINUS 0)**.) Let M be the corresponding term model, in a context where the existing shells are exactly those introducted in Section 1.C, i.e. those appropriate for SASL.

The intended model for SASL is known as the *SASL domain*, denoted V, and is constructed from basic principles in Kaufmann [8]. As it is a standard Scott-style domain (actually a c.p.o., or continuous partial order), it can be constructed using results of Scott, as in [16].

Fix a finite set C of SASL values, i.e. elements of the SASL domain V. Correspondingly, let V be the first-order structure with universe V (the SASL domain); binary function • interpreting juxtaposition; constants hd, tl, and cons; constants for the recognizers (logical, char, ...); constants for the SASL operators (plus, and, cond [for "...->...;..."], and so on); and constants for the elements of C. Now V is a first-order structure, so by the Löwenheim-Skolem theorem of first-order logic, we may choose a countable submodel V_C of V which satisfies the same first-order sentences as does V.

Next, we define a bijection j from the universe V_C of V_C onto M as follows. As usual, we'll use boldface to denote terms. NOTE: we do <u>not</u> claim that the definition of j is in any sense constructive.

- ☐ j(true) = **T**, j(false) = **F**, j(BOTTOM) = **(BTM)**, j([]) = **(SASL-NIL)**.
- ☐ j(c) = c for c a character in AT; otherwise, j(c) = **(MINUS 0)**
- ☐ j(n) = **n** for all nonnegative integers n.
  NOTE: We use the convention that **n** abbreviates **(ADD1 (ADD1 ... (ADD1 0)...))** (for n-many **ADD1**s).
- ☐ j(-n) = **(MINUS n)** for all positive integers n.
- ☐ Let List be the set of all list elements of V_C. Then let Finlist be the set of all $x \in$ List such that for some natural number n, every

composition f of n functions from the set {hd, tl} has the property that f(x) = BOTTOM. (Intuitively, these are the finite trees in V_C.) Then for all $x \in (V_C\text{-List}) \cup \text{Finlist}$ and $y \in \text{Finlist} \cup \{\text{BOTTOM}\}$, we define j recursively by:
j(cons x y) = **(FINPAIR t_1 t_2)** where t_1 = j(x) and t_2 = j(y).

- ☐ Let Inflist = List - Finlist. Then (j | Inflist) maps Inflist 1-1 onto the terms of type LISTP.
- ☐ Let Function be the set of function elements of V. Then (j | Function) maps Function one-one onto the set of terms of type LITATOM.

The last two clauses make sense because V_C is countable.

(ii) *Interpreting (on M) the declared functions of the initial database: the initial interpretation* $\mathcal{I}_1$

We may now interpret the declared functions in Appendix 2 by using the bijection j. We still leave unspecified a fixed finite set C of SASL values. Let $\mathcal{I}_0$ be the basic interpretation for M defined in Section 2.A for the basic SASL shells defined in Section 1.C. We wish to extend $\mathcal{I}_0$ to an interpretation $\mathcal{I}_1$ for M. For notational convenience we use lower case for the interpretation by $\mathcal{I}_1$ of a symbol given in upper case, e.g. foo = $\mathcal{I}_1$(FOO). We adopt the usual convention that application '•' associates to the left, i.e. x•y•z = (x•y)•z. Also, we use italics for SASL values in the definition of $\mathcal{I}_1$ below, and write x for $j^{-1}(x)$ and x for $j(x)$ (similarly for y, z, ...). Finally, we write 'hd', 'tl', 'cons', 'plus', 'eq', and so on for the familiar function elements of the SASL domain, which we assume belong to C. (Here 'eq' is computable SASL equality.) NOTE: we do <u>not</u> claim that the definition of $\mathcal{I}_1$ is constructive.

sasl-char x = j(char • x) where char is the SASL recognizer; similarly for sasl-decode and sasl-code, where (decode n) is the character with ASCII code n if one exists, and is BOTTOM otherwise, and (code x) is the ASCII code of x if x is a character, and is BOTTOM otherwise.

cons-inf x y = j(cons • x • y) if x or y has type LISTP;
inf otherwise, for a fixed arbitrary infinite list *inf*.

hd-inf x = j(hd • x) if x has type LISTP; otherwise, *inf*

tl-inf x = j(tl $\cdot x$) if x has type LISTP; otherwise, *inf*
ap x y = j($x \cdot y$)
sasl-plus- x y = j(plus $\cdot x \cdot y$)
(similarly for sasl-times-, sasl-difference-, sasl-div-,
sasl-rem-, sasl-exp-; these could have been *defined* but there is no need
to do so)
extensionality-witness x y is chosen to be any z such that
ap x z $\neq$ ap y z, if such z exists; otherwise, **(BTM)**

In addition, it is necessary to give the interpretations of SASL-EQUAL and REAL-LENGTH since these functions are added essentially as axioms rather than as definitions (because of the extra two arguments to DEFN, namely NIL and T). We continue with the same conventions as above:

sasl-equal x y = j(eq $\cdot x \cdot y$)
real-length x = least n such that (tl $\cdot$ tl $\cdot$ ... $\cdot$ tl $\cdot x$) = BOTTOM, where
there are n occurrences of tl, if such n exists; otherwise, **(BTM)**.

(iii) *Extending the initial interpretation according to SASL-DEFN events*

Our next step is to consider a sequence of SASL-DEFN events and extend the initial interpretation I_1 to an interpretation appropriate to that sequence. Of course, in an actual session with the Prover one would probably intermingle DEFN and SASL-DEFN events, rather than putting all of the SASL-DEFN events first (as might be suggested by the approach we are taking). Fortunately, we'll see that because SASL-DEFN events never use function symbols defined in DEFN events (except for those in the domain of I_1), it's perfectly all right to imagine them as all coming before the DEFN events. This approach simplifies the exposition, since we will use the sequence of SASL-DEFN events to interpret RENAME and to determine the set C of SASL values used in the section above.

As we mentioned before, the only function symbols which may occur in SASL-DEFN events are those from previous SASL-DEFN events, the defined symbol itself, and certain given (SASL) functions, which we now make explicit:

Definition. The *SASL primitives* include BTM, SASL-NIL, and AP-OUTER, together with all of the functions defined or declared in Appendix 2 which have the prefix "SASL" and do not end with "-" (so e.g. SASL-PLUS- is excluded).

Now given any term whose function symbols are all SASL function symbols (or RENAME, with appropriate argument), one would like to produce a corresponding SASL expression (program), i.e. one in which RENAME does not occur.

Definition. Given any term t, the *corresponding SASL expression* is obtained by replacing each subterm of t of the form (RENAME 'FN) with FN.

Here are some relevant notions from SASL semantics.

- An *environment* is a map from the set of identifiers to the set of SASL values (i.e. elements of the SASL domain V).

- The function Eval takes two arguments: a SASL expression and an environment. It returns a SASL value, namely the value of the given expression in the given environment.

- The function Dval takes two arguments: a sequence of SASL definitions and an environment. It returns a new environment, namely the environment obtained by modififying the given environment according to the given definitions.

Careful definitions of Eval and Dval, based on definitions from Turner [18], can be found in [11]. However, we will attempt to keep this exposition self-contained.[6]

Now fix a sequence **defs** of SASL-DEFN events. We may also view **defs** as a sequence of SASL definitions, by replacing each definition body with its corresponding SASL expression. Corresponding to this sequence is a SASL environment, i.e. a map from identifiers to values. Formally, this environment is ρ = Dval $[\![\mathbf{defs}]\!]\ \rho_0$, where ρ_0 is the *initial SASL environment*: it assigns the appropriate value to each SASL primitive (including AP-OUTER, which is mapped to $\lambda x.\ \lambda y.$ [if x is a list, then x_y; if x is a function, then $x \cdot y$; otherwise, BOTTOM]). Now we may make the relevant definitions.

- Let Z be the set of identifiers defined in **defs**.

- Let C = $\{\rho\ z:\ z \in Z\}$.

☐ Extend $\mathcal{I}_1$ to an interpretation $\mathcal{I}_{\mathbf{defs}}$ as follows:

☐ Let $\mathcal{I}_{\mathbf{defs}}$(RENAME) = rename be an arbitrary function from M into the set of elements of M of type LITATOM, such that for all $z \in Z$, rename ('z) = j (ρ z).

☐ For each definition

f v_1 ... v_n = body

in **defs**, $\mathcal{I}_{\mathbf{defs}}$(f) is the n-place function F satisfying:

F x_1 ... x_n = j(f • x_1 • ... • x_n)

where $f = \rho$ f and $x_i = j^{-1}(x_i)$ for $1 \le i \le n$. (As usual, 0-place functions are constants.)

(iv) *Summary*

It is clear now how to define an interpretation corresponding to a sequence SEQ of events. First, extend the initial interpretation $\mathcal{I}_1$ as shown in Subsection (iii) above to the interpretation $\mathcal{I}_{\mathbf{defs}}$, where **defs** is the subsequence of SASL-DEFN events extracted from the original sequence SEQ. Next let SEQ_1 consist of the subsequence of DEFN events of SEQ. Extend $\mathcal{I}_{\mathbf{defs}}$ as in Lemma 2.2, successively for each member of SEQ_1 (in order). The final such interpretation is the *interpretation corresponding to* SEQ.

We conclude with a lemma which allows us to move between terms in the Boyer-Moore logic and their values in the term model, on the one hand, and SASL expressions and their values in V_C, on the other hand. First, it is helpful to extend the notion of interpretation to allow values to be assigned to variables.

Definition. An *extended interpretation* is a map $\mathcal{I}$ from a set of function symbols and variables into a set.

In conventional logic terminology, an extended interpretation is really an interpretation together with an assignment (of values to variables). Hence there is a clear notion of meaning of terms in an extended interpretation, so that the notion "holds in" extends naturally in this context (i.e., one implicitly universally quantifies all variables in the term

which are not in the domain of the extended interpretation). Notice that every interpretation is also an extended interpretation. We may now state the promised lemma.

Lemma 2.3. Let **defs** be a sequence of SASL-DEFN events and let $\mathcal{I}_{\mathbf{defs}}$ be the corresponding interpretation defined above. Let t be a term all of whose function symbols are in the domain of $\mathcal{I}_{\mathbf{defs}}$, and let t' be the corresponding SASL expression. Let $\mathcal{I}$ be an extended interpretation which extends $\mathcal{I}_{\mathbf{defs}}$ and assigns values $e_1, ..., e_n$ to the variables $x_1, ..., x_n$ occurring in t. Finally, let $\underline{t}$ be the value of t under $\mathcal{I}$ and let $\underline{t'}$ be the appropriate value for t', namely
$\underline{t'}$ = Eval ⟦t'⟧ ((Dval ⟦**defs**⟧ ρ_0)[e_i/x_i]) where ρ_0 is the initial SASL environment. Then $\underline{t} = j(\underline{t'})$.

Proof. Routine by induction on terms using the definition of $\mathcal{I}_{\mathbf{defs}}$.

□

Henceforth we will often blur the distinction between a term and its corresonding SASL expression, when there is no danger of confusion.

2C LIFT events: directed completeness and soundness

The idea of directed completeness is that certain propositions (the directedly complete ones) have the property that whenever they hold of a sequence of values, then they hold of the limit of that sequence. This idea was explored in Gordon et al [3], for example, to give proofs by fixpoint induction. Our approach, which is similar, is made precise and proved sound in this section. We begin with some preliminary definitions. Throughout this section, we fix a sequence of events.

Definition. A *SASL function symbol* is one which is either defined by a SASL-DEFN event (in the given event sequence) or is a SASL primitive. A *basic SASL term* is one which is either a nonnegative numeral **n**, a negative numeral **(MINUS n)** (including **(MINUS 0)**, which is a character), a term of the form **(SASL-DECODE t)**, **(BTM)**, **(SASL-NIL)**, **T**, or **F**. A *finite SASL term* is one which is built up from basic SASL terms using only the function symbol FINPAIR. A *SASL term* is one which

is built up from finite SASL terms using SASL function symbols.

A *literal* term is a term of the form (EQUAL t u) or (NOT (EQUAL t u)), where t and u are terms.

Finally, we write vars(t) to denote the set of variables which occur in the term t.

Definition. Let VARS be a set of variables. The class of literal terms which are *directedly complete in* VARS is defined as follows.

□ (EQUAL t u) is directedly complete in VARS iff:
VARS ∩ vars(t) ≠ ∅ ⇒ t is a SASL term; and
VARS ∩ vars(u) ≠ ∅ ⇒ u is a SASL term.

□ (NOT (EQUAL t u)) is directedly complete in VARS iff:
VARS ∩ vars(t) ≠ ∅ ⇒ t is a SASL term and u is a finite SASL term; and,
VARS ∩ vars(u) ≠ ∅ ⇒ u is a SASL term and t is a finite SASL term.

More generally, let t be a term in conjunctive normal form, i.e. one which is a conjunction of disjunctions of literal terms. (Of course, every term is provably equal to one in this form.) Then t is *directedly complete in* vars if and only if each disjunct of each conjunct of t is directedly complete, as defined above. Finally, an arbitrary term is *directedly complete in* vars if it is equivalent to one which is directedly complete in vars. Thus for example, if we broaden the notion of "literal" to include SASL terms, then the notion of directed completeness for arbitrary terms is unchanged, since t is always equivalent to (NOT (EQUAL t F)).

Now when given a command of the form

(LIFT lifted_theorem_name theorem_types source_theorem_name
(x_1 ... x_n))

the Prover checks that **source_theorem_name** is the name of a previously proved lemma of the form

(IMPLIES (NLISTP x_1)
  (IMPLIES (NLISTP x_2)
  ...
    (IMPLIES (NLISTP x_n)
      t) ...))

where t is directedly complete in $\{x_1, \ldots x_n\}$. (NLISTP is the composition of NOT with LISTP.) If the Prover is able to verify this, then it adds a new theorem to the database, as described in Section 1.D. Otherwise, the Prover rejects this command.

Lemma 2.4 (soundness of LIFT events). Suppose that t is a term which is directedly complete in the set of variables $\{x_1, \ldots, x_n\}$. Let SEQ be a sequence of events and let $\mathcal{I}$ be the interpretation corresponding to SEQ. Finally, suppose that the term t' given by

(IMPLIES (NLISTP x_1)
  (IMPLIES (NLISTP x_2)
  ...
    (IMPLIES (NLISTP x_n)
      t) ...))

holds in $\mathcal{I}$. Then t holds in $\mathcal{I}$.

Proof. Let **defs** be the subsequence of SASL definitions in SEQ, and let ρ be the environment Dval $[\![\textbf{defs}]\!]\,\rho_0$, where ρ_0 is the initial SASL environment. Let '$\sqsubseteq$' be the natural partial order on the SASL domain, as described for example in Kaufmann [8] (or in any other treatment of denotational semantics, such as Stoy [17]). We may write each $\mathcal{I}(x_i)$ as the supremum of an $\sqsubseteq$-increasing sequence $\langle e_{in}: n \geq 0\rangle$ of SASL values, none of which is an infinite list structure (or *inflist*, in the notation of Subsection 2.B(i)). For $n \geq 0$ let $\mathcal{I}_n$ be the result of extending $\mathcal{I}$ by setting its value on x_i equal to e_{in}. Since t' holds in $\mathcal{I}$, clearly t holds in $\mathcal{I}_n$ for all n. Thus it suffices to prove the following claim:

(1) If t is a directedly complete term which holds in $\mathcal{I}_n$ for all n, then t holds in $\mathcal{I}$.

To prove (1), we observe that without loss of generality we may assume that t is in conjunctive normal form. It is also clear then that it suffices to consider the case that t is a single disjunction of literals. Since this disjunction is finite, the hypothesis of (1) implies that some disjunct of t holds in $\mathcal{I}_n$ for all n belonging to some infinite subsequence of the natural numbers. If we can prove the lemma for literals only, then by restricting to this subsequence we see that this disjunct holds in $\mathcal{I}$ as well; hence so does t. Therefore, it suffices to prove the lemma under the assumption that t is a directedly complete literal.

Case 1.1: t is of the form (EQUAL u u') for SASL terms u and u'. By hypothesis, this term holds in $\mathcal{I}_n$ for all $n \geq 0$. By Lemma 2.3, we have

$$\text{Eval } [\![u]\!] (\rho[e_{in}/x_i]) = \text{Eval } [\![u']\!] (\rho[e_{in}/x_i])$$

for all $n \geq 0$. By the continuity of Eval (which is a standard sort of fact and can be found in [11, §8, Lemma D10]), it follows that

$$\text{Eval } [\![u]\!] (\rho[e_i/x_i]) = \text{Eval } [\![u']\!] (\rho[e_i/x_i])$$

where $e_i = \mathcal{I}(x_i)$. Then applying Lemma 2.3 again we conclude that t holds in $\mathcal{I}$.

Case 1.2: t is of the form (EQUAL u u'), where u is a SASL term and $\{x_1, \ldots, x_n\} \cap \text{vars}(u') = \emptyset$. Let $\underline{u'}$ be the value of u' under $\mathcal{I}$. By Lemma 2.3, we have

$$\text{Eval } [\![u]\!] (\rho[e_{in}/x_i]) = \underline{u'}$$

for all $n \geq 0$. By the continuity of Eval, it follows that

$$\text{Eval } [\![u]\!] (\rho[e_i/x_i]) = \underline{u'}$$

where $e_i = \mathcal{I}(x_i)$. Then applying Lemma 2.3 again we conclude that t holds in $\mathcal{I}$.

Cases 1.3, 1.4: The other two cases for (EQUAL u u') are similar.

Case 2.1: t is of the form (NOT (EQUAL u u')), where u is a SASL term and u' is a finite SASL term. As in the proof of Case 1.1, Lemma 2.3 implies

$$\text{Eval } [\![u]\!] (\rho[e_{in}/x_i]) = \text{Eval } [\![u']\!] (\rho[e_{in}/x_i])$$

for all $n \geq 0$, i.e.

$$(2) \quad \text{Eval } [\![u]\!] (\rho[e_{in}/x_i]) = \text{Eval } [\![u']\!] \rho$$

since u' is variable-free. Since u' is a finite SASL term, it's clear that Eval $[\![u']\!]$ ρ is not a non-trivial limit under $\sqsubseteq$. Hence by the continuity of Eval together with (2), this implies

$$\text{Eval } [\![u]\!] (\rho[e_i/x_i]) = \text{Eval } [\![u']\!] \rho$$

where $e_i = \mathcal{I}(x_i)$. Then applying Lemma 2.3 again we conclude that t holds in $\mathcal{I}$.

The remaining cases are similar, and the proof is complete. □

2.D Type sets

We introduced the logic of the SASL Prover in Sections C and D of Part 1. However, we left out some axioms that are automatically added to the system in conjuction with each DEFN and SASL-DEFN event. In the case of DEFN, these axioms are described in Chapter VI of [2], and they take the following form:

$$(*) \quad \backslash/ \text{ type}_i(f\ x_1 \ldots x_n) \ \backslash/ \ \backslash/ \ (\text{EQUAL } (f\ x_1 \ldots x_n)\ x_j)$$

where one disjunction may be empty, and each "type_i" is a type recognizer such as NUMBERP or the "type" OTHERS, which holds of objects not belonging to any shell (in our case, the characters other than **(MINUS 0)**). Such axioms are also added for SASL-DEFN events, but the algorithm for producing them is a bit different from the one for DEFN events. In this

section we comment on each of these two cases. The results will be used in the proof of soundness in the next section.

We begin with some notions adapted from Chapter VI of [2]. A *type set* is simply a collection of types, i.e. shells, where in our context we have a fixed set of shells (given in Section 1.C). As in [2], we have a type OTHERS (which in our case will be the type of all the characters except **(MINUS 0)**). A *type prescription* is a pair <ts,vars> where ts is a type set and vars is a set of variables. A type prescription <ts,vars> for a function symbol f defined with formal parameters $(x_1,...,x_n) \supseteq$ vars asserts that for all $x_1,...,x_n$, either $f(x_1,...,x_n)$ has a type belonging to ts or else $f(x_1,...,x_n) = x_i$ for some $x_i \in$ vars; in this case we say that <ts,vars> *holds for* f (in the interpretation under consideration). A *function type assumption* is a mapping from a set of function symbols to type prescriptions, while a *term type assumption* is a mapping from a set of terms to type prescriptions. A *type assumption* is a pair <FTA,TTA> where FTA is a function type assumption and TTA is a term type assumption.

There is a straightforward recursive algorithm for giving the type *prescription of a term* t *relative to a type assumption* TA = <FTA,TTA> *and a set* VARS *of variables*, as follows. If t is in the domain of TTA, then return TTA(t). If t is a variable then return <{},{t}> if $v \in$ VARS; otherwise return <UNIVERSE,{}>, where UNIVERSE is the union of all types. The other cases are straightforward, as described in Chapter VI of [2], but here is a brief summary. One branches on IF-expressions, taking the union of the first and second components of the type expressions for each branch where for each branch one appropriately extends FTA according to the assumption of truth or falsity of the test of the IF-expression (unless the type set computation for the test shows either that it's always F or that it's never F). Finally, let f be a function symbol other than IF, with formals $x_1, ..., x_n$. Let TA be a type assumption which assigns type prescription <U,V> to f. Then the type prescription of $(\mathbf{f}\ t_1 \ldots t_n)$ relative to TA is defined to be <X,Y>, where: t_i has type prescription $\langle X_i,Y_i\rangle$ relative to TA and VARS $(1 \le i \le n)$, $X = \cup \{X_i: x_i \in V\} \cup U$, and $Y = \cup \{Y_i: x_i \in V\}$.

Now that we've made all of these definitions, we would like to associate a type prescription with each defined function symbol. Given a term t and a type prescription <X,Y>, one may consider the following term (thought of as a formula with free variable t) which asserts that t

obeys this type prescription:

$$tp_{\langle X,Y\rangle}(t) \Box \text{ (OR } (\backslash/ \{(\mathbf{r_{type}\ t})\text{: type}\in X\}) \; (\backslash/ \{(\mathbf{EQUAL\ t\ y})\text{: } y\in Y\}))$$

where r_{type} is the recognizer for type [7]. (Of course, the symbol $\backslash/$ is used to abbreviate iterated use of the function symbol OR.) Let $\mathcal{I}$ be an extended interpretation into M. We say that a term type assumption TTA holds in $\mathcal{I}$ if for all terms t in the domain of TTA, TTA(t) = ⟨X,Y⟩ implies that $tp_{\langle X,Y\rangle}(t)$ holds in $\mathcal{I}$. A function type assumption FTA *holds in* $\mathcal{I}$ if for every function symbol f in the domain of FTA, with formal parameters $x_1, \ldots, x_n$, if FTA(f) = ⟨X,Y⟩ then $tp_{\langle X,Y\rangle}(f\ x_1 \ldots x_n)$ holds in $\mathcal{I}'$ for all extended interpretations $\mathcal{I}'$ which agree with $\mathcal{I}$ except that they may assign aribitrary values to $x_1, \ldots, x_n$. Finally, a type assumption ⟨FTA,TTA⟩ *holds in* $\mathcal{I}$ if FTA and TTA both hold in $\mathcal{I}$. The following lemma is easily proved by a straightforward induction on terms.

Lemma 2.5 (correctness of type prescription algorithm). Suppose that TA = ⟨FTA,TTA⟩ is a type assumption which holds in a given extended interpretation $\mathcal{I}$. Let t be a term whose function symbols are all in the domain of FTA, let VARS be a set of variables, and suppose that ⟨X,Y⟩ is the type prescription of t relative to TA and VARS. Then $tp_{\langle X,Y\rangle}(t)$ holds in $\mathcal{I}$. □

If $\langle X_i,Y_i\rangle$ (i=1,2) are type prescriptions such that $X_1 \subseteq Y_1$ and $X_2 \subseteq Y_2$, then we say that $\langle X_1,Y_1\rangle$ *is contained in* $\langle X_2,Y_2\rangle$. Unlike Lemma 2.5, the following lemma is purely syntactic. We omit its routine proof as well.

Lemma 2.6 (monotonicity of type computations). Suppose that TA = ⟨FTA,TTA⟩ and TA' = ⟨FTA',TTA'⟩ are type assumptions such that FTA(f) is contained in FTA'(f) for all f in their common domain, and similarly TTA(t) is contained in TTA'(t) for all t in their common domain. Then the type prescription computed for any term t relative to TA and a set VARS of variables is contained in the type prescription computed for t relative to TA' and VARS. □

Now the type prescription for a DEFN event

f V_1 ... V_n = BODY

relative to a function type assumption FTA is computed as follows. Let f be the function symbol being defined. Let $\langle X_0,Y_0\rangle$ be the type prescription $\langle\{\},\{\}\rangle$. Given $\langle X_j,Y_j\rangle$, let $\langle X_{j+1},Y_{j+1}\rangle$ be the type prescription computed for BODY relative to $\langle$FTA',$\{\}\rangle$ and $\{V_1, ..., V_n\}$, where FTA' assigns the type prescription $\langle X_j,Y_j\rangle$ to f and otherwise agrees with FTA. By the monotonicity of type computations (Lemma 2.6), we know that for some J we have $\langle X_J,Y_J\rangle = \langle X_{J+1},Y_{J+1}\rangle$ (and hence $\langle X_J,Y_J\rangle = \langle X_j,Y_j\rangle$ for all $j>J$). We let $\langle X_J,Y_J\rangle$ be the type prescription for the given DEFN event relative to FTA.

The following lemma is used in our soundness proof. The idea is that given a (successful) DEFN event D, there are a measure and a well-founded relation used to justify that the function is defined on all arguments (cf. Chapter III of [2]). An induction on the measure applied to the arguments of the function, along this well-founded relation, can be used to establish this lemma; we omit this argument.

Lemma 2.7 (soundness of DEFN type prescriptions). Let FTA be a function type assumption which holds in a given interpretation $\mathcal{I}$, let D be a DEFN event defining a function symbol f, and let $\mathcal{J}$ be the extension of $\mathcal{I}$ via D (as defined at the end of Section 2.A). Then the type prescription for D relative to FTA holds for f in $\mathcal{J}$. □

The *type-prescription algorithm for SASL-DEFN events* is the same as the one above for DEFN events, except that this time the initial type prescription $\langle X_0,Y_0\rangle$ is $\langle\{$BTMP$\},\{\}\rangle$, where BTMP is the recognizer for the shell containing BOTTOM. Thus we have to be a bit more careful this time in how we apply monotonicity if BTMP$\notin X_1$. Fortunately, one may easily show by induction on the structure of BODY that if BTMP$\notin X_1$ then $\langle X_1,Y_1\rangle$ remains the same if it is computed instead using $\langle\{\},\{\}\rangle$ for $\langle X_0,Y_0\rangle$ (as before). (This argument is particularly simple because the function symbol IF does not occur in SASL-DEFN events.) Thus the monotonicity argument again applies to guarantee that the sequence

$\langle X_j, Y_j \rangle$ stabilizes (as before). Notice that if the definition is not recursive then the result is simply $\langle X_1, Y_1 \rangle$.

More generally, we may define the notion of a type prescription for a sequence of SASL-DEFN events. First, let us fix an initial function type assumption.

Definition. The *initial function type assumption* is defined as follows. For function symbols in the domain of the basic interpretation I_0 (cf. Section 2.A), the type prescriptions assigned to these functions are fairly obvious: for example, a recognizer R has type prescription <{R},{}>, and PLUS has type prescription <NUMBERP,{}>. (We omit the details.) Now we may successively extend *that* function type assumption to obtain the initial function type assumption, as follows. Consider the sequence of events in the initial SASL library (Appendix 2). For each DEFN event we extend the interpretation via that event (as in Lemma 2.2). For each DCL event we extend by assigning type prescriptions as follows:

- CONS-INF: <{LISTP},{}>
- HD-INF: <UNIVERSE,{}>
- TL-INF: <{LISTP,FINLISTP,BTMP},{}>
- AP: <UNIVERSE,{}>
- SASL-PLUS- (and analogous "dashed" arithmetic functions): <{NUMBERP,BTMP},{}> [8]
- RENAME: <{LITATOM},{}>
- EXTENSIONALITY-WITNESS: <UNIVERSE,{}>
- SASL-EQUAL: <{TRUEP,FALSEP,BTMP},{}>

Definition. Let **defs** be a sequence of SASL-DEFN events. Then the *SASL function type assumption for* **defs** is obtained by beginning with the initial function type assumption (defined just above) and successively extending it by applying the type-prescription algorithm for SASL-DEFN events to the members of **defs** (in order).

Lemma 2.8 (soundness of SASL-DEFN type prescriptions). Let **defs** be a sequence of SASL-DEFN events, and let $I_{\mathbf{defs}}$ be the corresponding extension of I_1 as defined in Subsection 2.B(iii). Then the SASL type assumption for **defs** holds in $I_{\mathbf{defs}}$.

Proof. The proof is by induction on the length of the sequence **defs**. If this length is 0 then this is clear by definition of the initial function type

assumption. For the inductive step, suppose that the lemma holds for the sequence **defs** and let us consider the sequence **defs'** obtained by adding a definition

$$f\ x_1 \ldots x_n = \text{body}$$

to the end of **defs**. By the inductive hypothesis, it suffices to show that the type prescription assigned to f by the SASL type assumption for **defs'** holds for f in $\mathcal{I}_{\mathbf{defs'}}$. Let ρ_0 be the initial SASL environment (cf. Subsection 2.A(iii)), and let ρ = Dval $[\![\mathbf{defs}]\!]\ \rho_0$. Since the function symbol f does not occur in **defs** (for otherwise the Prover would reject the definition of f), the denotational semantics of SASL guarantee that Dval $[\![\mathbf{defs'}]\!]\ \rho_0$ = Dval $[\![f\ x_1 \ldots x_n = \text{body}]\!]\ \rho$ (cf. [11,§8,Lemma D5]). Henceforth let us denote this common environment by ρ'.

Let $\langle \langle X_n, Y_n \rangle : n \geq 0 \rangle$ be the sequence of type prescriptions computed for the definition of f. Now we may take the definition of f and consider instead an infinite sequence Δ of nonrecursive definitions

$$f_0\ x_1 \ldots x_n = \mathbf{(BTM)}$$
$$f_{j+1}\ x_1 \ldots x_n = \text{body}_j \quad (j>0)$$

where body_j is the result of replacing each occurrence of f by f_j in body (including those occurrences in **(RENAME 'f)**). We make the following claims, where **defs**(Δ) is the result of appending Δ to the end of **defs**. Note: we use Lemma 2.3 implicitly, so that we may view Δ as a sequence of SASL definitions or of SASL-DEFN events, as we choose.

(1) The SASL type assumption TA(Δ) for **defs**(Δ) assigns a type prescription $\langle X^j, Y^j \rangle$ to f_j which holds in $\mathcal{I}_{\mathbf{defs}(\Delta)}$, all $j \geq 0$; i.e., $tp_{\langle X^j, Y^j \rangle}(f_j\ x_1 \ldots x_n)$ holds in $\mathcal{I}_{\mathbf{defs}(\Delta)}$.

(2) $\rho'\ f$ is the supremum of the values Dval $[\![\mathbf{defs}(\Delta)]\!]\ \rho_0\ f_j$ $(j \geq 0)$.

(3) Suppose that a SASL function g is the supremum of SASL functions g_i. Also suppose that for each i, the type prescription $\phi(f)$ holds in an extension of the initial interpretation $\mathcal{I}_1$ in which f is mapped to

the function taking
$a_1, ..., a_n$ to $j(g_i \cdot a_1 \cdot ... \cdot a_n)$, where $a_k = j^{-1}(a_k)$. Then $\Phi(f)$ holds in any extension $\mathcal{I}$ of $\mathcal{I}_1$ in which f is mapped to the function taking $a_1, ..., a_n$ to $j(g \cdot a_1 \cdot ... \cdot a_n)$, where $a_k = j^{-1}(a_k)$.

Let us see first how the theorem follows from these claims. For each $i \geq 0$ let $g_i = \text{Dval } [\![\Delta]\!]\ \rho\ f_i$, and let $g = \rho' f$. By (2), g is the supremum of the g_i. Also, for each j we have $tp_{\langle X^j, Y^j \rangle}(f_j\ x_1 \ ...\ x_n)$ holds in $\mathcal{I}_{\mathbf{defs}(\Delta)}$, by (1). The hypotheses of (3) follow, and hence the conclusion of (3) holds taking $\mathcal{I} = \mathcal{I}_{\mathbf{defs}'}$. That is what we needed to show. Thus it remains only to prove the claims.

The proof of (1) is by induction on j. The case j=0 is clear since the type prescription assigned to f_0 is $\langle \{BTMP\}, \{\} \rangle$. Now suppose that (1) holds for $i \leq j$. By Lemma 2.5, the type prescription $\langle X, Y \rangle$ computed for $body_j$ relative to $TA(\Delta)$ and $\{x_1, ..., x_j\}$ holds in $\mathcal{I}_{\mathbf{defs}(\Delta)}$, i.e.

(4) $tp_{\langle X, Y \rangle}(body_j)$ holds in $\mathcal{I}_{\mathbf{defs}(\Delta)}$.

Let $body_j'$ be the SASL expression corresponding to $body_j$ (as defined in Subsection 2.B(iii)). Now by definition of Dval (cf. [11]), if we set $\rho'' = \text{Dval } [\![\mathbf{defs}(\Delta)]\!]\ \rho_0$, then $\rho''\ f_{j+1} = \lambda e_1. ... \lambda e_n.\ \text{Eval } [\![body_j']\!]\ (\rho''[e_i/x_i])$. Thus for all $e_1, ..., e_n \in V_C$, the terms $(f_{j+1}\ x_1 \ ...\ x_n)$ and $body_j'$ have the same value in the environment $\rho''[e_i/x_i]$. It follows from Lemma 2.3 that the terms $(f_{j+1}\ x_1 \ ...\ x_n)$ and $body_j$ have the same value under the interpretation $\mathcal{I}_{\mathbf{defs}(\Delta)}$ extended by assigning e_i to x_i $(1 \leq i \leq n)$. This fact together with (4) implies that $tp_{\langle X, Y \rangle}(f_{j+1}\ x_1 \ ...\ x_n)$ holds in $\mathcal{I}_{\mathbf{defs}(\Delta)}$, as desired.

The proof of (2) is an routine exercise in the denotational semantics of SASL, which goes as follows. First, it's not difficult to show by induction on j that for all j we have: $\text{Dval } [\![\mathbf{defs}(\Delta)]\!]\ \rho_0\ f_j \sqsubseteq \rho' f$. A separate induction also easily establishes that for all j, j iterations of $\lambda\rho_1.\ \text{Decl } [\![f\ x_1 \ ...\ x_n = body]\!]\ \rho_1\ \rho$, when applied to the bottom environment and to f, is below $\text{Dval } [\![\mathbf{defs}(\Delta)]\!]\ \rho_0\ f_j$ in the order $\sqsubseteq$; taking the limit

as $j \to \infty$, we have that $\rho' f$ is in the relation $\sqsubseteq$ to the supremum of the values $Dval [\![\mathbf{defs}(\Delta)]\!] \rho_0 f_j$ $(j \geq 0)$. We omit the details.

To prove (3), we first observe that for every type prescription $\langle X,Y \rangle$ assigned to a SASL primitive function symbol f by the initial SASL type assumption, we have $LISTP \in X$ iff $FINLISTP \in X$. It follows easily that the type prescription computed for any subsequent SASL definition must also have this property. Of course, for any formula $tp_{\langle X,Y \rangle}(t)$ with this property, we may replace

(OR (LISTP t) (FINLISTP t))

by

(AND (SASL-LIST t) (NOT (EQUAL t (BTM))) .

Similarly, **(BTMP t)** may be replaced by **(EQUAL t (BTM))** (similarly for TRUEP and FALSEP), **(LITATOM t)** by **(AND (EQUAL (SASL-FUNCTION t) T) (NOT (EQUAL t (BTM)))**, **(NUMBERP t)** by **(EQUAL (SASL-GTE t 0) T)**, and **(NEGATIVEP t)** by **(OR (EQUAL (SASL-LT t 0) T) (EQUAL t (MINUS 0)))**. Thus, it's now clear that for type prescriptions $\langle X,Y \rangle$ of SASL definitions, $tp_{\langle X,Y \rangle}(t)$ may be written as a boolean combination of equations, where each equation has one side consisting simply of **T**, of **F**, of **(BTM)**, or of **(MINUS 0)** and the other side having the property that all of its function symbols are SASL functions. By the proof of Lemma 2.3 (but without even worrying about RENAME), we may view all such terms as being SASL expressions (using the appropriate character for **(MINUS 0)**, of course). Finally, then, (3) follows from a directed completeness argument very similar to the proof of Lemma 2.4, except that the present formula is directedly complete in the function symbol f (rather than in a set of variables). We leave the precise definition of this notion of directed completeness as well as the requisite lemma to the reader, as they are entirely analogous to the corresponding work in Section 2.C. □

2.E **The soundness theorem**

All of the pieces are finally in place for a statement and proof of soundness. The lemma below is the conclusion of all the work above, while the theorem below it gives a result appropriate to the original SASL domain. We conclude with a few remarks about how one might generalize

this latter theorem.

Lemma 2.9 (soundness for the term model). Let SEQ be a sequence of events run successfully by the Prover. Then every equation stored by the Prover holds in the interpretation corresponding to SEQ (as defined in Seciton 2.B).

Proof. The proof is by induction on the length of SEQ. If SEQ is empty then we only need check that every axiom in Appendix 2 holds in the initial interpretation $\mathcal{I}_1$, as do the defining equations of SASL-EQUAL and REAL-LENGTH. This is routine (thought tedious) and left to the reader.

For the inductive step, there are several cases according to the form of the last command of SEQ. If that last command is a PROVE-LEMMA event, then we are done by the inductive hypothesis together with Lemma 2.1 (soundness of Boyer-Moore logic). If the command is a DEFN event, then Lemma 2.2 (extension by DEFN events) and Lemma 2.7 (soundness of DEFN type prescriptions) combine with the inductive hypothesis to yield the conclusion. For a SASL-DEFN event, one applies Lemma 2.8 (soundness of SASL-DEFN type prescriptions) and Lemma 2.3, together with a simple check that the axioms involving RENAME (see Section 1.C) hold. Finally, if the last event is a LIFT event then one simply applies Lemma 2.4 (soundness of LIFT events) together with the inductive hypothesis. □

We would like to state a theorem which applies to the original SASL domain. To that end we make the following definition.

Definition. Let ρ be an environment. For SASL expressions t_1 and t_2 we say that ρ *satisfies* the equation $t_1 = t_2$, written $\rho \models t_1 = t_2$, iff Eval $[\![t_1]\!]\ \rho$ = Eval $[\![t_2]\!]\ \rho$. More generally, we define the notion $\rho \models \phi$ for ϕ a boolean combination of equations, by: $\rho \models$ (AND ϕ_1 ϕ_2) iff $\rho \models \phi_1$ and $\rho \models \phi_2$; $\rho \models$ (OR ϕ_1 ϕ_2) iff $\rho \models \phi_1$ or $\rho \models \phi_2$; and $\rho \models$ (NOT ϕ) iff it is not the case that $\rho \models \phi$.

Theorem. Suppose that ϕ is a boolean combination of equations between SASL terms and that ϕ is proved by the Prover in a sequence SEQ of events whose SASL-DEFN events form a subsequence **defs**. Then $\rho \models \phi$, for any environment ρ of the form (Dval $[\![\textbf{defs}]\!]\ \rho_0$)$[e_i/x_i: 1 \le i \le n]$, where ρ_0 is the initial SASL environment (as defined in Subsection

2.B(iii)) and $x_1, \ldots, x_n$ are the variables occurring in ϕ.

Proof. By Lemma 2.9, we know that ϕ holds in the interpretation corresponding to SEQ, and hence in its restriction $\mathcal{I}_{\mathbf{defs}}$ (defined in Subsection 2.B(iii)), since each equation in ϕ is an equation between SASL terms. Then by Lemma 2.3, the sentence $(\forall x_1) \ldots (\forall x_n)\, \phi$ is true in $\mathcal{V}_C$ (defined in Subsection 2.B(i)), where C contains a constant symbol for each function declared or defined in Appendix 2 together with each symbol defined in **defs**. Since $\mathcal{V}_C$ satisfies the same first-order sentences as $\mathcal{V}$ (by definition), this sentence $(\forall x_1) \ldots (\forall x_n)\, \phi$ is also true in $\mathcal{V}$, and this fact is just a restatement of the desired conclusion. □

The theorem above can be made somewhat more powerful than may appear at first. For example, consider the following lemma from Appendix 3:

```
(IMPLIES (AND (NUMBERP N)
              (NUMBERP K)
              (LESSP 0 K))
         (EQUAL (SUBSC (SASL-FROM N) K)
                (SUB1 (PLUS N K))))
```

Suppose that one defines interpretations for the function symbols NUMBERP, LESSP, SUBSC, SUB1, and PLUS on $\mathcal{V}$ in the "obvious" way, i.e. so that NUMBERP returns false on all non-numbers (even BOTTOM), SUBSC is list subscription, SUB1 returns 0 on every value that is not a positive integer (just as it does in the term model), and PLUS coerces its arguments to natural numbers by replacing other values by 0. Now even though these are not SASL functions (except for SUBSC, in a sense) because they fail to be continuous, still one can add these symbols to the language (which thus far has had only '=' and constants for elements of C) and require that $\mathcal{V}_C$ be an elementary submodel of $\mathcal{V}$ with respect to this larger language. In this manner one can make perfect sense out of functions such as NUMBERP and even REAL-LENGTH, INF-LIST, and so on. However, we see no particularly clean uniform way to do this at this point So, we'll leave that open and end our discussion of soundness here.

PART 3: AN ALTERNATE APPROACH

The approach described in the first two parts can be needlessly awkward when dealing only with finite structures. In this part we present an approach, appropriate for some such situations, which avoids reasoning about BOTTOM. We illustrate this approach with a couple of examples.

3.A **The alternate initial library and outline of soundness**

The alternate approach involves use of an initial SASL library which differs somewhat from the one described heretofore. Moreover, the LIFT event is no longer allowed. These are however the only two changes to the Prover from the version already described.

The alternate initial SASL library is in Appendix 4. The difference between this and the original SASL library is in the way the set of all lists is divided into the "finite" and "infinite" lists. As before, a "finite" list is once again a finite tree, except that this time one does not allow functions (LITATOMs) or BOTTOM in the tree. Such lists constitute the shell with recognizer SLIST ("S" for "strict", but that doesn't matter) and constructor SCONS, and accessors SHD and STL. Thus, the type restrictions are that the first argument of SCONS not be a LITATOM, a LISTP, or BOTTOM and that the second argument be an SLIST. Notice that the default values are SASL-NIL, so that for example (SHD (SASL-NIL)) equals (SASL-NIL). However, this is not a problem, since the function SASL-HD is defined so that (SASL-HD (SASL-NIL)) equals (BTM); for X an SLIST, we have (SASL-HD X) = (SHD X) only when X ≠ (SASL-NIL).

A most thorough approach here would parallel the contents of Part 2. However, it is more convenient (to us and to the reader!) merely to indicate the changes needed in Part 2 in order to establish soundness for the alternate approach given in this Part.

Of course, Section 2.A (construction of a term model) is general and applies to this Part as well. Next, consider Section 2.B: interpretations corresponding to sequences of events. Subsection (i) defined a countable subuniverse of the SASL domain as well as a term model and an isomorphism between them. This time the term model is for a set of shells in which the SLIST shell replaces the FINLISTP shell. The isomorphism is as before, except that one breaks the SASL lists into "finite" and "infinite" ones as indicated above, and builds the isomorphism accordingly. Subsection (ii) defined the initial interpretation. For the alternate approach one replaces CONS-INF, HD-INF, and TL-INF by LCONS,

LHD, and LTL respectively. (The "L" is for "lazy" in honor of infinite lists, but that doesn't matter.) This time, we find it convenient to set (LHD X) and (LTL X) equal to (BTM) when X is not a LISTP, but otherwise Subsection (ii) and the rest of Section 2.B remain essentially unchanged for the alternate approach. Section 2.C is irrelevant for the alternate approach, since LIFT events are no longer allowed.

Section 2.D -- type prescriptions -- remains unchanged through Lemma 2.7. The initial function type assumption also remains unchanged, except that the type prescriptions for CONS-INF, HD-INF, and TL-INF are replaced by:

- LCONS: <{LISTP},{}>
- LHD: <UNIVERSE,{}>
- LTL: <{LISTP,SLIST,BTMP},{}>

Then the proof of Lemma 2.8 remains unchanged. (The use of a version of directed completeness near the end of the proof has nothing to do with notions of "finiteness" for lists.) Finally, the soundness results of Section 2.E go through unchanged (except that one should now ignore the LIFT command).

3.B **Proof of correctness of a SASL pattern-matching program**[9]

Consider the problem of defining a function which tests a list P of strings against a list D of strings, to see if the "pattern" P matches the list D in the following sense. Two empty lists of strings always match, but if just one list is empty then they do not match. The string "*" in P matches any finite positive number of strings in D, so that for example

["ab","*","cd"]

matches

["ab","123","4567","Matt","cd"]

but does not match ["ab","cd"]. Finally, the string "?" matches any single string, so that for example

["ab","?","cd"]

matches

["ab","uvw","cd"]

but does not match ["ab","123","456","cd"]. The following recursive Pure Lisp program is presented on page 325 of Winston [20] (with the modification indicated at the top of page 327) as a solution to this problem. (Actually, in Lisp one considers lists of symbols rather than lists of strings, but that difference is trivial.)

```
(DEFINE (MATCH P D)
        (COND ((AND (NULL P) (NULL D)) T)
              ((OR (NULL P) (NULL D)) NIL)
              ((OR (EQUAL (CAR P) '?)
                   (EQUAL (CAR P) (CAR D)))
               (MATCH (CDR P) (CDR D)))
              ((EQUAL (CAR P) '*)
               (COND ((MATCH (CDR P) (CDR D)))
                     ((MATCH P (CDR D)))))))
```

However, in our attempt to mechanically prove correctness of a SASL analogue of this program, we found a small error in the above program! Consider, for example, the question of whether the pattern

```
('* 'do)
```

matches

```
('* 'howdy 'do)
```

According to the informal specification given at the start of this section, the given pattern should match the given list, as the symbol '* is allowed to correspond to the list ('* 'howdy). However, the condition

```
(OR  (EQUAL (CAR P) '?)
     (EQUAL (CAR P) (CAR D)))
```

is satisfied, and this test comes before the test for whether (CAR P)

equals '* in the program above. The question thus reduces to whether the pattern ('do) matches the pattern ('howdy 'do), and this of course comes out false.

Obviously it's easy to correct this error by switching the order of the tests which check for equality of the first member of the pattern to '* or '? . The result translates to SASL as follows:

```
match P D =
  P=[] & D=[] -> TRUE;
  P=[] | D=[] -> FALSE;
  hd P = "*" -> match (tl P) (tl D) | match P (tl D);
  hd P = "?" | hd P = hd D -> match (tl P) (tl D);
  FALSE
```

The sequence of events comprising the proof of correctness of this program appears in Appendix 5. To state correctness we use some auxiliary notions. How does one assert that pattern P matches list D? We illustrate our approach by way of the following example. Consider the pattern

P = ["hi", "*", "boo", "?", "fido"] ,

which matches the list of strings

D = ["hi", "a","b", "boo", "c", "fido"] .

Now consider an "intermediate" list

L = ["hi", ["a","b"], "boo", "c", "fido"] .

The list L has the following properties: (1) P and L are lists which agree at every coordinate, except that when "*" occurs at a given position of P then an arbitrary list of strings occurs at the same position of L, and when "?" occurs at a given position of P then an arbitrary string occurs at the same position of L; and (2) D is obtained from L by flattening out L -- for example, the element ["a","b"] of L is replaced by two corresponding elements of D. In the parlance of Appendix 5, we say that (EXPANSION P L) equals T and (FLATTEN L) equals D.

The statement of correctness is thus broken into two directions. The theorem SUFFICIENCY-FOR-MATCH of Appendix 5 states that if P and D

are "finite" lists of strings for which a "finite" list L exists meeting the requirements above, then (MATCH P D) equals TRUE. Conversely, the theorem NECESSITY-OF-MATCH states that if P and D are "finite" lists of strings for which (MATCH P D) is TRUE (or in fact, does not equal FALSE), then such L exists. In fact, the existence of such L is given as a function W ("witness") of P and D.

3.C **Proof of correctness of a SASL unification program**

The paper Manna-Waldinger [12] presents a careful proof of correctness of a unification algorithm, with an eye toward possible machine certification of the proof. Paulson [13] outlines a successful machine-aided verification of a unification program using an enhanced version of Edinburgh LCF [3], which Paulson calls Cambridge LCF. We decided to try such a verification with the Prover for two reasons. First, we wanted to present verification of executable code in an actual language. Paulson's program is close to meeting this criterion, though his functions are actually given by axioms. The other reason is that there is a well-defined semantics for the SASL language and (as argued above) soundness for the Prover, which in our view makes verification much more meaningful.

The reader is referred to [12] for suitable background, including a development of a unification algorithm.

Our proof used 623 events in addition to those in the alternate initial SASL library, cf. Kaufmann [6].[10] To save space, we include here only a statement of the final theorem. The definitions of the SASL unification function and its supporting cast are given in Appendix 6. But let us give here a brief description of the functions mentioned in this theorem.

A *term* is either a variable, a constant, or (recursively) a list of terms, and (TERM X) is T exactly if X is a term. (For our purposes, a *variable* is a string beginning with the character 'v', and a *constant* is any other string.) A *substitution* is a "finite" list of variable-term pairs, and (SUBSTP X) is true exactly if X is a substitution. The function SUBST and its finite analog SUBST- take two arguments, and return the result of substituting into the second argument (presumably a term) using the first argument (presumably a substitution). The function UNIFY is the desired unification function, so that (UNIFY L M) produces a substitution which unifies the given terms L and M, if such a substitution exists, and otherwise returns "FAIL". The function COMPOSE (with finite analog

COMPOSE-) takes two substitutions and produces a new substitution which acts like their composition, in the sense that the result of applying SUBST to this result and a given term is the same as the result of successively applying SUBST to each of the given substitutions (and the given term). IDEMP recognizes *idempotent* substitutions, i.e. ones whose domain is disjoint from the set of variables occurring in the range. (It follows that the result of composing an idempotent substitution with itself is itself.) The function SSUBSET recognizes whether its first argument is a subset of its second argument. The SUB-VARS of a substitution is a list of all variables which belong to its domain or occur in its range, while the VARS of a term is a list of all variables occurring in that term. Finally, (FAIL) is just the string "FAIL".

The main theorem is a conjunction of the following two implications. The idea of the first implication is that if there exists a unifier of given terms L and M, then (UNIFY L M) is a unifier with nice properties, while the second implication states if (UNIFY L M) is not a unifier of L and M (i.e., by the first implication we know that there is no unifier of L and M), then (UNIFY L M) equals "FAIL".

Before stating these two parts of the main theorem, we make a few comments about the first of them. Notice that the last few parts of the conclusion assert the equivalence of various functions with their finite analogs, under certain assumptions. The hypothesis (EQUAL (SUBST- SUB L) (SUBST- SUB M)) states that there is *some* unifier of terms L and M, namely SUB. The fourth conclusion says that, under the given assumptions (especially the one just mentioned), (UNIFY L M) is a unifier of L and M. The second conclusion says that in fact it is a *most general* unifier (cf. [12]). We let the other conclusions speak for themselves. Here, then, are the two implications.

```
(IMPLIES (AND (TERM L)
              (TERM M)
              (SUBSTP SUB)
              (EQUAL (SUBST- SUB L) (SUBST- SUB M))
              (TERM Z))
         (AND (SUBSTP (UNIFY L M))
              (EQUAL
                  (SUBST- (COMPOSE- (UNIFY L M) SUB) Z)
                  (SUBST- SUB Z))
              (IDEMP (UNIFY L M))
              (EQUAL
                  (SUBST- (UNIFY L M) L)
                  (SUBST- (UNIFY L M) M))
              (TERM (SUBST- (UNIFY L M) L))
              (TERM (SUBST- (UNIFY L M) M))
              (SUBSTP (COMPOSE- (UNIFY L M) SUB))
              (SSUBSET
                  (SUB-VARS (UNIFY L M))
                  (APPEND- (VARS L) (VARS M)))
              (EQUAL
                  (SUBST- (UNIFY L M) Z)
                  (SUBST (UNIFY L M) Z))
              (EQUAL
                  (SUBST- SUB Z)
                  (SUBST SUB Z))
              (EQUAL
                  (COMPOSE- (UNIFY L M) SUB)
                  (COMPOSE (UNIFY L M) SUB))
              (EQUAL
                  (SUBST- (COMPOSE- (UNIFY L M) SUB) Z)
                  (SUBST (COMPOSE (UNIFY L M) SUB) Z))))


(IMPLIES (AND (TERM L)
              (TERM M))
         (OR  (AND (SUBSTP (UNIFY L M))
                   (EQUAL
                       (SUBST- (UNIFY L M) L)
                       (SUBST- (UNIFY L M) M)))
              (EQUAL (UNIFY L M) (FAIL))))
```

APPENDIX 1: AN INTRODUCTORY EXAMPLE USING THE PROVER

What follows is a sequence of events which has been run by the Prover (on a Symbolics 3640). Everything following a semicolon on a line is a comment. We give only the input to the Prover, omitting the output. This appendix is adapted from Kaufmann [9].

```
;;; We begin by recalling the library of SASL facts:

(NOTE-LIB ">thm>sasl.lib" ">thm>sasl.lisp")

;;; Now we begin feeding the Prover events.

(SASL-DEFN FACT1 (N)
           (SASL-IF (SASL-EQUAL N 0)
                    1
                    (SASL-TIMES N (FACT1 (SASL-DIFFERENCE N 1)))))

(SASL-DEFN FACT2 (ACC N)
           (SASL-IF (SASL-EQUAL N 0)
                    ACC
                    (FACT2 (SASL-TIMES ACC N) (SASL-DIFFERENCE N 1))))

;;; Having made these definitions, we would like to prove the theorem
;;; FACT-IS-FACT below.  However, this is quite impossible for the
;;; Prover, as it would in fact be for any human, unless
;;; it is derived as a special case of the more general
;;; FACT-IS-FACT-LEMMA below.  It would be nice if the Prover could
;;; just prove this latter lemma without any hints, but there are
;;; three reasons why this isn't so easy.  For one, in the current
;;; environment it really doesn't "know" much number theory, though
;;; there are existing files of number-theoretic facts that would
;;; solve that problem.  Since we're not getting those files here,
;;; the theorem-prover is told below to prove them for us.  Here, then,
;;; is the number theory that turns out to be needed.

(PROVE-LEMMA TIMES-ID (REWRITE)
             (IMPLIES (NUMBERP X)
                      (AND (EQUAL (TIMES X 1) X)
                           (EQUAL (TIMES 1 X) X))))


(PROVE-LEMMA ASSOC-OF-TIMES (REWRITE)
             (IMPLIES (AND (NUMBERP X)
                           (NUMBERP Y)
                           (NUMBERP Z))
                      (EQUAL (TIMES X (TIMES Y Z))
                             (TIMES (TIMES X Y) Z))))
```

```
;;; The next, more serious problem is that the induction used to prove
;;; FACT-IS-FACT-LEMMA isn't quite a simple induction on N.  Here is an
;;; outline of the inductive step of the proof:

; ACC * FACT1(N+1) = ACC * ((N+1) * FACT1(N))  (by def. of FACT1)
;                  = (ACC * (N+1)) * FACT1(N)) (by associativity of *)
;                  = FACT2 (ACC*(N+1)) N       (by the inductive hypothesis)
;                  = FACT2 ACC (N+1)           (by def. of FACT2) .

;;; Notice that when the inductive hypothesis is applied, it is applied
;;; where ACC is replace by (ACC * (N+1)).  The Prover can't
;;; figure that out on its own, so we give it a definition FACT-IS-FACT-IND
;;; below, and then give a hint for the proof of FACT-IS-FACT-LEMMA to
;;; do the proof by induction, using the induction principle generated
;;; when the FACT-IS-FACT-IND definition was accepted.


(DEFN FACT-IS-FACT-IND (ACC N)
  (IF (ZEROP N)
      T
      (FACT-IS-FACT-IND (TIMES ACC N) (SUB1 N))))

;;; But even this hint isn't enough, as the Prover can't yet prove
;;; FACT-IS-FACT-LEMMA at this stage.  One problem is shown by the
;;; following output, which appears in the attempted proof:

;       (IMPLIES (AND (NUMBERP X)
;                     (NUMBERP ACC)
;                     (NOT (EQUAL X 0))
;                     (NOT (NUMBERP (FACT1 X)))
;                     (NOT (NEGATIVEP (FACT1 X))))
;                (EQUAL (TIMES ACC (BTM))
;                       (TIMES (TIMES ACC (ADD1 X))
;                              (FACT1 X))))

;;; Why is the theorem-prover considering (TIMES ACC (BTM)) here?
;;; ( (BTM) denotes the "bottom" value of the SASL domain.)  It would
;;; have seemed that we should only be dealing with numbers.  Well, the
;;; hypothesis (NOT (NUMBERP (FACT1 X))) seems to contradict the
;;; hypothesis (NUMBERP X), but the theorem-prover isn't aware of this
;;; fact.  What's more, the theorem-prover can't prove that this is a
;;; contradiction, at least not without some help.  So we ask it to
;;; prove that  FACT1  always returns a non-negative number.  However,
;;; it gets stuck there as well, so first we give it an appropriate
;;; induction.

(DEFN FACT1-IS-NUMBERP-IND (N)
  (IF (ZEROP N)
      T
      (FACT1-IS-NUMBERP-IND (SUB1 N))))

(PROVE-LEMMA FACT1-IS-NUMBERP (REWRITE)
             (IMPLIES (NUMBERP N)
                      (NUMBERP (FACT1 N)))
             ((INDUCT (FACT1-IS-NUMBERP-IND N))))
```

```
;;; It turns out to be useful to disable the definition of  TIMES  at
;;; this point.

(DISABLE TIMES)

;;; Finally we are ready to prove the main lemma and then easily derive
;;; the desired theorem.

(PROVE-LEMMA FACT-IS-FACT-LEMMA (REWRITE)
             (IMPLIES (AND (NUMBERP N)
                           (NUMBERP ACC))
                      (EQUAL (TIMES ACC (FACT1 N))
                             (FACT2 ACC N)))
             ((INDUCT (FACT-IS-FACT-IND ACC N))))


(PROVE-LEMMA FACT-IS-FACT (REWRITE)
             (IMPLIES (AND (NUMBERP N)
                           (NUMBERP ACC))
                      (EQUAL (FACT1 N)
                             (FACT2 1 N))))
```

APPENDIX 2: THE INITIAL SASL LIBRARY

```
(ADD-SHELL BTM NIL BTMP NIL)

(DCL SASL-CHAR (X))

(DCL SASL-DECODE (X))

(DCL SASL-CODE (X))

(ADD-SHELL FINPAIR SASL-NIL FINLISTP
   ((FINHD
      (NONE-OF LISTP)
      BTM)
    (FINTL
      (ONE-OF BTMP FINLISTP)
      BTM)))

(DCL CONS-INF (X Y))

(ADD-AXIOM CONS-INF-LISTP (REWRITE)
           (LISTP (CONS-INF X Y)))

(DEFN SASL-CONS (X Y)
     (IF (OR (LISTP X)
             (LISTP Y))
         (CONS-INF X Y)
         (FINPAIR X Y)))

(DCL HD-INF (X))

(DEFN SASL-HD (X) (IF (LISTP X) (HD-INF X) (FINHD X)))

(DCL TL-INF (X))

(DEFN SASL-TL (X) (IF (LISTP X) (TL-INF X) (FINTL X)))

(DCL AP (FN X))

(DEFN SASL-TRUE () T)

(DEFN SASL-FALSE () F)

(DEFN SASL-LIST (X)
  (IF (EQUAL X (BTM))
      (BTM)
      (IF (FINLISTP X)
          T
          (LISTP X))))
```

```
(DEFN SASL-LOGICAL (X)
 (IF (EQUAL X (BTM))
     (BTM)
     (IF X
         (IF (EQUAL X T) T F)
         T)))

; SASL-CHAR defined near the beginning

(DEFN SASL-FUNCTION (X)
  (IF (EQUAL X (BTM))
      (BTM)
      (LITATOM X)))

(DEFN SASL-INTEGER (X)
  (IF (EQUAL X (BTM))
      (BTM)
      (OR (NUMBERP X)
          (AND (NOT (EQUAL X (MINUS 0)))
               (NEGATIVEP X)))))

(DEFN SASL-IF (X Y Z)
  (IF X
      (IF (EQUAL X T) Y (BTM))
      Z))

(DEFN SASL-AND (X Y)
  (IF X
      (IF (EQUAL X T)
          (IF Y
              (IF (EQUAL Y T)
                  T
                  (BTM))
              F)
          (BTM))
      F))

(DEFN SASL-OR (X Y)
  (IF X
      (IF (EQUAL X T) T (BTM))
      (IF Y
          (IF (EQUAL Y T) T (BTM))
          F)))

(DEFN SASL-NOT (X)
  (IF X
      (IF (EQUAL X T) F (BTM))
      T))

(ADD-AXIOM HD-INF-TL-INF-ELIMINATION
           (ELIM)
           (IMPLIES
             (LISTP X)
             (EQUAL (CONS-INF (HD-INF X) (TL-INF X)) X)))
```

```
(ADD-AXIOM HD-INF-SASL-AND-TL-INF-OF-NONLISTS
           (REWRITE)
           (IMPLIES
             (NOT (LISTP X))
             (AND (EQUAL (HD-INF X) (BTM))
                  (EQUAL (TL-INF X) (BTM)))))

(ADD-AXIOM TL-INF-REWRITE (REWRITE)
           (OR (LISTP (TL-INF X))
               (FINLISTP (TL-INF X))
               (EQUAL (TL-INF X) (BTM))))

(ADD-AXIOM HD-INF-OF-SASL-CONS
           (REWRITE)
           (IMPLIES (OR (LISTP X)
                        (LISTP Y))
                    (EQUAL (HD-INF (CONS-INF X Y)) X)))

(ADD-AXIOM TL-INF-OF-SASL-CONS
           (REWRITE)
           (IMPLIES (OR (LISTP X)
                        (LISTP Y))
                    (EQUAL (TL-INF (CONS-INF X Y))
                           (IF (SASL-LIST Y) Y (BTM)))))

(DCL SASL-PLUS- (X Y))

(DEFN SASL-PLUS (X Y)
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (PLUS X Y)
      (IF (OR (AND (NOT (NEGATIVEP X))
                   (NOT (NUMBERP X)))
              (AND (NOT (NEGATIVEP Y))
                   (NOT (NUMBERP Y))))
          (BTM)
          (SASL-PLUS- X Y))))

(DCL SASL-TIMES- (X Y))

(DEFN SASL-TIMES (X Y)
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (TIMES X Y)
      (IF (OR (AND (NOT (NEGATIVEP X))
                   (NOT (NUMBERP X)))
              (AND (NOT (NEGATIVEP Y))
                   (NOT (NUMBERP Y))))
          (BTM)
          (SASL-TIMES- X Y))))

(DCL SASL-DIFFERENCE- (X Y))
```

```
(DEFN SASL-DIFFERENCE (X Y)
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (DIFFERENCE X Y)
      (IF (OR (AND (NOT (NEGATIVEP X))
                   (NOT (NUMBERP X)))
              (AND (NOT (NEGATIVEP Y))
                   (NOT (NUMBERP Y))))
          (BTM)
          (SASL-DIFFERENCE- X Y))))

(DCL SASL-DIV- (X Y))

(DEFN SASL-DIV (X Y)
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (IF (EQUAL Y 0)
          (BTM)
          (QUOTIENT X Y))
      (IF (OR (AND (NOT (NEGATIVEP X))
                   (NOT (NUMBERP X)))
              (AND (NOT (NEGATIVEP Y))
                   (NOT (NUMBERP Y))))
          (BTM)
          (SASL-DIV- X Y))))

(DCL SASL-REM- (X Y))

(DEFN SASL-REM (X Y)
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (IF (EQUAL Y 0)
          (BTM)
          (REMAINDER X Y))
      (IF (OR (AND (NOT (NEGATIVEP X))
                   (NOT (NUMBERP X)))
              (AND (NOT (NEGATIVEP Y))
                   (NOT (NUMBERP Y))))
          (BTM)
          (SASL-REM- X Y))))

(DEFN EXP (X Y)
  (IF (ZEROP Y)
      1
      (TIMES X (EXP X (SUB1 Y)))))

(DCL SASL-EXP- (X Y))

(DEFN SASL-EXP (X Y)
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (IF (AND (EQUAL X 0) (EQUAL Y 0))
          (BTM)
          (EXP X Y))
      (IF (OR (AND (NOT (NEGATIVEP X))
                   (NOT (NUMBERP X)))
              (AND (NOT (NEGATIVEP Y))
                   (NOT (NUMBERP Y))))
          (BTM)
          (SASL-EXP- X Y))))
```

```
(DEFN SASL-GT (X Y)
  (IF (OR (EQUAL X (MINUS 0))
          (EQUAL Y (MINUS 0)))
      (BTM)
      (IF (NUMBERP Y)
          (IF (NUMBERP X)
              (GREATERP X Y)
              (IF (NEGATIVEP X)
                  F
                  (BTM)))
          (IF (NEGATIVEP Y)
              (IF (NUMBERP X)
                  T
                  (IF (NEGATIVEP X)
                      (LESSP (NEGATIVE-GUTS X)
                             (NEGATIVE-GUTS Y))
                      (BTM)))
              (BTM)))))

(DEFN SASL-LT (X Y)
  (IF (OR (EQUAL X (MINUS 0))
          (EQUAL Y (MINUS 0)))
      (BTM)
      (IF (NUMBERP Y)
          (IF (NUMBERP X)
              (LESSP X Y)
              (IF (NEGATIVEP X)
                  T
                  (BTM)))
          (IF (NEGATIVEP Y)
              (IF (NUMBERP X)
                  F
                  (IF (NEGATIVEP X)
                      (GREATERP (NEGATIVE-GUTS X)
                                (NEGATIVE-GUTS Y))
                      (BTM)))
              (BTM)))))

(DEFN SASL-GTE (X Y)
  (IF (OR (EQUAL X (MINUS 0))
          (EQUAL Y (MINUS 0)))
      (BTM)
      (IF (NUMBERP Y)
          (IF (NUMBERP X)
              (GEQ X Y)
              (IF (NEGATIVEP X)
                  F
                  (BTM)))
          (IF (NEGATIVEP Y)
              (IF (NUMBERP X)
                  T
                  (IF (NEGATIVEP X)
                      (LEQ (NEGATIVE-GUTS X)
                           (NEGATIVE-GUTS Y))
                      (BTM)))
              (BTM)))))
```

```
(DEFN SASL-LTE (X Y)
  (IF (OR (EQUAL X (MINUS 0))
          (EQUAL Y (MINUS 0)))
      (BTM)
      (IF (NUMBERP Y)
          (IF (NUMBERP X)
              (LEQ X Y)
              (IF (NEGATIVEP X)
                  T
                  (BTM)))
          (IF (NEGATIVEP Y)
              (IF (NUMBERP X)
                  F
                  (IF (NEGATIVEP X)
                      (GEQ (NEGATIVE-GUTS X)
                           (NEGATIVE-GUTS Y))
                      (BTM)))
              (BTM)))))


(DCL RENAME (X))

(ADD-AXIOM SASL-LOGICALREP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-LOGICAL) X)
                  (SASL-LOGICAL X)))

(ADD-AXIOM SASL-CHARREP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-CHAR) X)
                  (SASL-CHAR X)))

(ADD-AXIOM SASL-INTEGERREP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-INTEGER) X)
                  (SASL-INTEGER X)))

(ADD-AXIOM SASL-LIST-REP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-LIST) X)
                  (SASL-LIST X)))

(ADD-AXIOM SASL-FUNCTIONREP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-FUNCTION) X)
                  (SASL-FUNCTION X)))

(ADD-AXIOM SASL-CONS-REP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-CONS) X) Y)
                  (SASL-CONS X Y)))
```

```
(ADD-AXIOM SASL-HD-REP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-HD) X)
                  (SASL-HD X)))

(ADD-AXIOM SASL-TL-REP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-TL) X)
                  (SASL-TL X)))

(ADD-AXIOM SASL-IFREP
           (REWRITE)
           (EQUAL (AP (AP (AP (RENAME 'SASL-IF) X) Y) Z)
                  (SASL-IF X Y Z)))

(ADD-AXIOM SASL-ANDREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-AND) X) Y)
                  (SASL-AND X Y)))

(ADD-AXIOM SASL-ORREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-OR) X) Y)
                  (SASL-OR X Y)))

(ADD-AXIOM SASL-NOTREP
           (REWRITE)
           (EQUAL (AP (RENAME 'SASL-NOT) X)
                  (SASL-NOT X)))

(ADD-AXIOM SASL-PLUSREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-PLUS) X) Y)
                  (SASL-PLUS X Y)))

(ADD-AXIOM SASL-TIMESREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-TIMES) X) Y)
                  (SASL-TIMES X Y)))

(ADD-AXIOM SASL-DIVREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-DIV) X) Y)
                  (SASL-DIV X Y)))

(ADD-AXIOM SASL-REMREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-REM) X) Y)
                  (SASL-REM X Y)))

(ADD-AXIOM SASL-LTREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-LT) X) Y)
                  (SASL-LT X Y)))
```

```
(ADD-AXIOM SASL-GTREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-GT) X) Y)
                  (SASL-GT X Y)))

(ADD-AXIOM SASL-GTE-REP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-GTE) X) Y)
                  (SASL-GTE X Y)))

(ADD-AXIOM SASL-LTE-REP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-LTE) X) Y)
                  (SASL-LTE X Y)))

(ADD-AXIOM SASL-EXPREP
           (REWRITE)
           (EQUAL (AP (AP (RENAME 'SASL-EXP) X) Y)
                  (SASL-EXP X Y)))

(ADD-AXIOM SASL-CONS-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-CONS) X))
                (NOT (LISTP (AP (RENAME 'SASL-CONS) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-CONS) X)))))

(ADD-AXIOM SASL-IF-FN-1
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-IF) X))
                (NOT (LISTP (AP (RENAME 'SASL-IF) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-IF) X)))))

(ADD-AXIOM SASL-IF-FN-2
           (REWRITE)
           (AND (LITATOM (AP (AP (RENAME 'SASL-IF) X) Y))
                (NOT (LISTP (AP (AP (RENAME 'SASL-IF) X) Y)))
                (NOT (FINLISTP (AP (AP (RENAME 'SASL-IF) X) Y)))))

(ADD-AXIOM SASL-AND-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-AND) X))
                (NOT (LISTP (AP (RENAME 'SASL-AND) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-AND) X)))))

(ADD-AXIOM SASL-OR-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-OR) X))
                (NOT (LISTP (AP (RENAME 'SASL-OR) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-OR) X)))))

(ADD-AXIOM SASL-PLUS-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-PLUS) X))
                (NOT (LISTP (AP (RENAME 'SASL-PLUS) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-PLUS) X)))))
```

```
(ADD-AXIOM SASL-TIMES-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-TIMES) X))
                (NOT (LISTP (AP (RENAME 'SASL-TIMES) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-TIMES) X)))))

(ADD-AXIOM SASL-DIFFERENCE-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-DIFFERENCE) X))
                (NOT (LISTP (AP (RENAME 'SASL-DIFFERENCE) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-DIFFERENCE) X)))))

(ADD-AXIOM SASL-DIV-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-DIV) X))
                (NOT (LISTP (AP (RENAME 'SASL-DIV) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-DIV) X)))))

(ADD-AXIOM SASL-REM-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-REM) X))
                (NOT (LISTP (AP (RENAME 'SASL-REM) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-REM) X)))))

(ADD-AXIOM SASL-EXP-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-EXP) X))
                (NOT (LISTP (AP (RENAME 'SASL-EXP) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-EXP) X)))))

(ADD-AXIOM SASL-LT-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-LT) X))
                (NOT (LISTP (AP (RENAME 'SASL-LT) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-LT) X)))))

(ADD-AXIOM SASL-LTE-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-LTE) X))
                (NOT (LISTP (AP (RENAME 'SASL-LTE) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-LTE) X)))))

(ADD-AXIOM SASL-GTE-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-GTE) X))
                (NOT (LISTP (AP (RENAME 'SASL-GTE) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-GTE) X)))))

(ADD-AXIOM SASL-GT-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-GT) X))
                (NOT (LISTP (AP (RENAME 'SASL-GT) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-GT) X)))))
```

```
(DEFN SUBSC (X N)
  (IF (NUMBERP N)
      (IF (EQUAL N 0)
          (BTM)
          (IF (EQUAL N 1)
              (SASL-HD X)
              (IF (EQUAL (SASL-LIST (SASL-TL X)) T)
                  (SUBSC (SASL-TL X) (SUB1 N))
                  (BTM))))
      (BTM)))

(DEFN AP-OUTER (X Y)
  (IF (EQUAL (SASL-LIST X) T)
      (SUBSC X Y)
      (IF (LITATOM X)
          (AP X Y)
          (BTM))))

(DCL EXTENSIONALITY-WITNESS (X Y))

(ADD-AXIOM SASL-CONS-OF-NON-FINOBJS
           NIL
           (EQUAL (LISTP (SASL-CONS X Y))
                  (OR (LISTP X) (LISTP Y))))

(ADD-AXIOM EXTENSIONALITY
           (REWRITE)
           (IMPLIES (AND
                      (LITATOM X)
                      (LITATOM Y)
                      (NOT (EQUAL X Y)))
                    (NOT (EQUAL (AP X (EXTENSIONALITY-WITNESS X Y))
                                (AP Y (EXTENSIONALITY-WITNESS X Y))))))

(ADD-AXIOM FUNCTION-RENAME (REWRITE)
           (LITATOM (RENAME X)))

(DEFN SASL-EQUAL (X Y)
  (IF (OR (EQUAL X (BTM))
          (EQUAL Y (BTM)))
      (BTM)
      (IF (AND (SASL-LIST X)
               (NOT (EQUAL X (SASL-NIL)))
               (SASL-LIST Y)
               (NOT (EQUAL Y (SASL-NIL))))
          (SASL-AND (SASL-EQUAL (SASL-HD X) (SASL-HD Y))
                    (SASL-EQUAL (SASL-TL X) (SASL-TL Y)))
          (IF (AND (LITATOM X)
                   (LITATOM Y))
              (BTM)
              (EQUAL X Y))))
  NIL T)

(ADD-AXIOM SASL-EQUAL-TYPE (REWRITE)
    (OR (EQUAL (SASL-EQUAL X Y) T)
        (EQUAL (SASL-EQUAL X Y) F)
        (EQUAL (SASL-EQUAL X Y) (BTM))))
```

```
(ADD-AXIOM SASL-EQUAL-FN
           (REWRITE)
           (AND (LITATOM (AP (RENAME 'SASL-EQUAL) X))
                (NOT (LISTP (AP (RENAME 'SASL-EQUAL) X)))
                (NOT (FINLISTP (AP (RENAME 'SASL-EQUAL) X)))))

(DEFN REAL-LENGTH (X)
  (IF (EQUAL (SASL-LIST X) T)
      (IF (EQUAL X (SASL-NIL))
          0
          (IF (NUMBERP (REAL-LENGTH (SASL-TL X)))
              (ADD1 (REAL-LENGTH (SASL-TL X)))
              (BTM)))
      0) NIL T)

(ADD-AXIOM REAL-LENGTH-TYPE (REWRITE)
   (OR (NUMBERP (REAL-LENGTH X))
       (BTMP (REAL-LENGTH X))))

(DEFN INF-LIST (X)
  (AND (LISTP X) (EQUAL (REAL-LENGTH X) (BTM))))


(PROVE-LEMMA FINHD-GEN (GENERALIZE)
    (NOT (LISTP (FINHD X))))

(PROVE-LEMMA FINTL-GEN (GENERALIZE)
    (OR (FINLISTP (FINTL X))
        (EQUAL (FINTL X) (BTM))))

(ADD-AXIOM SASL-CHAR-BTM (REWRITE)
           (EQUAL (SASL-CHAR (BTM)) (BTM)))
```

APPENDIX 3: SOME EXAMPLES

```
;; See Section 1.D for events regarding a SASL append function.

(PROVE-LEMMA SASL-LIST-SASL-LOGICAL
             NIL
             (OR (EQUAL (SASL-LIST X) T)
                 (EQUAL (SASL-LIST X) F)
                 (EQUAL (SASL-LIST X) (BTM))))

(PROVE-LEMMA SASL-LIST-BTM
             NIL
             (EQUAL (EQUAL (SASL-LIST X) (BTM)) (EQUAL X (BTM))))

(PROVE-LEMMA SASL-LIST-BTM2 NIL
              (EQUAL (SASL-LIST (BTM)) (BTM)))

(PROVE-LEMMA SASL-NIL-IS-A-LIST
              NIL
              (EQUAL (SASL-LIST (SASL-NIL)) T))

(PROVE-LEMMA SASL-NOT-A-LIST
              NIL
              (IMPLIES (OR (EQUAL X T)
                           (EQUAL X F)
                           (NUMBERP X)
                           (NEGATIVEP X)
                           ;(CHARP X)
                           (LITATOM X))
                       (EQUAL (SASL-LIST X) F)))

(PROVE-LEMMA COUNT-LISTP (REWRITE)
              (IMPLIES (LISTP X) (NOT (EQUAL (COUNT X) 0)))
              ((INDUCT (LENGTH X))))

(SASL-DEFN SASL-STRING (L)
           (SASL-OR (SASL-EQUAL L (SASL-NIL))
                    (SASL-AND (SASL-CHAR (SASL-HD L))
                              (SASL-STRING (SASL-TL L)))))

(SASL-DEFN SASL-ALL (P L)
           (SASL-IF (SASL-EQUAL L (SASL-NIL))
                    (SASL-TRUE)
                    (SASL-AND (AP-OUTER P (SASL-HD L))
                              (SASL-ALL P (SASL-TL L)))))

(SASL-DEFN SASL-TREE (L)
           (SASL-OR (SASL-STRING L)
                    (SASL-ALL (RENAME 'SASL-TREE) L)))
```

```
(SASL-DEFN SASL-MAP (FN L)
           (SASL-IF (SASL-EQUAL L (SASL-NIL))
                    (SASL-NIL)
                    (SASL-IF (SASL-NOT (SASL-LIST L))
                             (BTM)
                             (SASL-CONS (AP-OUTER FN (SASL-HD L))
                                        (SASL-MAP FN (SASL-TL L))))))

(SASL-DEFN SASL-CONCAT (L)
           (SASL-IF (SASL-EQUAL L (SASL-NIL))
                    (SASL-NIL)
                    (SASL-APPEND (SASL-HD L)
                                 (SASL-CONCAT (SASL-TL L)))))

(SASL-DEFN SASL-GLUE (L)
          (SASL-IF (SASL-STRING L)
                   L
                   (SASL-CONCAT (SASL-MAP (RENAME 'SASL-GLUE) L))))

(DEFN STRING (L)
  (IF (EQUAL L (BTM))
      (BTM)
      (IF (NOT (FINLISTP L))
          (BTM)
          (IF (EQUAL L (SASL-NIL))
              T
              (SASL-AND (SASL-CHAR (FINHD L))
                        (STRING (FINTL L)))))))

(PROVE-LEMMA STRING-IS-STRING (REWRITE)
             (IMPLIES (NOT (LISTP L))
                      (EQUAL (SASL-STRING L)
                             (STRING L))))

(DEFN AUX-TREE (FLAG L)
  (IF (EQUAL L (SASL-NIL))
      T
      (IF (NOT (FINLISTP L))
          (BTM)
          (IF FLAG
              (SASL-OR (STRING L)
                       (SASL-AND (AUX-TREE T (FINHD L))
                                 (AUX-TREE F (FINTL L))))
              (SASL-AND (AUX-TREE T (FINHD L))
                        (AUX-TREE F (FINTL L)))))))

(PROVE-LEMMA TREE-IS-TREE (REWRITE)
             (IMPLIES (NOT (LISTP L))
                      (AND
                        (EQUAL (SASL-TREE L)
                               (AUX-TREE T L))
                        (EQUAL (SASL-ALL (RENAME 'SASL-TREE) L)
                               (AUX-TREE F L)))))
```

```
(DEFN CONCAT (L)
  (IF (NOT (FINLISTP L))
      (BTM)
      (IF (EQUAL L (SASL-NIL))
          (SASL-NIL)
          (APPEND (FINHD L)
                  (CONCAT (FINTL L))))))

(PROVE-LEMMA CONCAT-IS-CONCAT (REWRITE)
             (IMPLIES (NOT (LISTP L))
                      (EQUAL (SASL-CONCAT L)
                             (CONCAT L))))

(DEFN AUX-GLUE (FLAG L)
  (IF (NOT (FINLISTP L))
      (BTM)
      (IF (EQUAL L (SASL-NIL))
          (SASL-NIL)
          (IF FLAG
              (SASL-IF (SASL-STRING L)
                       L
                       (CONCAT (FINPAIR (AUX-GLUE T (FINHD L))
                                        (AUX-GLUE F (FINTL L)))))
              (FINPAIR (AUX-GLUE T (FINHD L))
                       (AUX-GLUE F (FINTL L)))))))

(PROVE-LEMMA AUX-GLUE-IS-GLUE-AND-MAP-GLUE NIL
             (IMPLIES (NOT (LISTP L))
                      (EQUAL (AUX-GLUE FLG L)
                             (IF FLG
                                 (SASL-GLUE L)
                                 (SASL-MAP (RENAME 'SASL-GLUE) L)))))

(PROVE-LEMMA AUX-GLUE-IS-GLUE-AND-MAP-GLUE-REWRITE (REWRITE)
             (IMPLIES (NOT (LISTP L))
                      (EQUAL (SASL-GLUE L) (AUX-GLUE T L)))
             ((USE (AUX-GLUE-IS-GLUE-AND-MAP-GLUE (FLG T)))
              (DISABLE AUX-GLUE)))


(PROVE-LEMMA THEOREM-AUX-GLUE NIL
             (IMPLIES (AND (NOT (LISTP L))
                           (EQUAL (AUX-TREE FLG L) T))
                      (IF FLG
                          (EQUAL (STRING (AUX-GLUE FLG L)) T)
                          (EQUAL (STRING (CONCAT (AUX-GLUE FLG L))) T))))

(PROVE-LEMMA THEOREM-SASL-GLUE (REWRITE)
             (IMPLIES (NLISTP L)
                      (IMPLIES (EQUAL (SASL-TREE L) T)
                               (EQUAL (SASL-STRING (SASL-GLUE L)) T)))
             ((USE (THEOREM-AUX-GLUE (FLG T)))))

(LIFT THEOREM-SASL-GLUE-LIFTED NIL THEOREM-SASL-GLUE (L))
```

```
(SASL-DEFN SASL-LENGTH (L)
    (SASL-IF (SASL-LIST L)
             (SASL-IF (SASL-EQUAL L (SASL-NIL))
                      0
                      (SASL-PLUS 1 (SASL-LENGTH (SASL-TL L))))
             (BTM)))

(DEFN LEN (L)
  (IF (FINLISTP L)
      (IF (EQUAL L (SASL-NIL))
          0
          (SASL-PLUS 1 (LEN (FINTL L))))
      (BTM)))

(PROVE-LEMMA LEN-LEN (REWRITE)
             (IMPLIES (NLISTP L)
                      (EQUAL (SASL-LENGTH L) (LEN L))))

(PROVE-LEMMA LEN-TYPE (REWRITE)
             (OR (EQUAL (LEN X) (BTM))
                 (NUMBERP (LEN X))))

(PROVE-LEMMA LEN-APP (REWRITE)
             (IMPLIES (AND (NLISTP L)
                           (SASL-LIST L)
                           (EQUAL (LEN L) (BTM)))
                      (EQUAL (APPEND L K) L)))


(PROVE-LEMMA LEN-APP2 (REWRITE)
             (IMPLIES (NLISTP L)
                      (IMPLIES (AND
                                 (SASL-LIST L)
                                 (EQUAL (SASL-LENGTH L) (BTM)))
                               (EQUAL (SASL-APPEND L K) L))))

(LIFT LEN-APP2-LIFTED (REWRITE) LEN-APP2 (L))

(PROVE-LEMMA SASL-LEN-TYPE (REWRITE)
             (IMPLIES (NLISTP X)
                      (OR (EQUAL (SASL-LENGTH X) (BTM))
                          (EQUAL (SASL-GTE (SASL-LENGTH X) 0) T))))

(LIFT SASL-LEN-TYPE-LIFTED (REWRITE) SASL-LEN-TYPE (X))

(PROVE-LEMMA SASL-LEN-TYPE-TYPE-SET (REWRITE)
            (OR (EQUAL (SASL-LENGTH X) (BTM))
                (NUMBERP (SASL-LENGTH X)))
            ((USE (SASL-LEN-TYPE-LIFTED))))

(DEFN HACK (N L)
  (IF (ZEROP N)
      T
      (HACK (SUB1 N) (SASL-TL L))))
```

```
(PROVE-LEMMA SASL-LENGTH-0 (REWRITE)
             (EQUAL (EQUAL (SASL-LENGTH L) 0)
                    (EQUAL L (SASL-NIL)))
             ((USE (SASL-LENGTH))))

(PROVE-LEMMA SASL-LENGTH-IS-SOMETIMES-REAL-LENGTH NIL
             (IMPLIES (AND
                        (EQUAL N (SASL-LENGTH L))
                        (NUMBERP N))
                      (EQUAL (SASL-LENGTH L)
                             (REAL-LENGTH L)))
             ((INDUCT (HACK N L))))

(PROVE-LEMMA INFLIST-LENGTH-IS-BTM (REWRITE)
             (IMPLIES (INF-LIST L)
                      (EQUAL (SASL-LENGTH L) (BTM)))
             ((USE (SASL-LENGTH-IS-SOMETIMES-REAL-LENGTH
                     (N (SASL-LENGTH L))))))

(PROVE-LEMMA LEN-APP2-LIFTED-INF (REWRITE)
             (IMPLIES (INF-LIST L)
                      (EQUAL (SASL-APPEND L K) L)))
```

```
(SASL-DEFN SASL-REVERSE (L)
    (SASL-IF (SASL-LIST L)
             (SASL-IF (SASL-EQUAL L (SASL-NIL))
                      (SASL-NIL)
                      (SASL-APPEND (SASL-REVERSE (SASL-TL L))
                                   (SASL-CONS (SASL-HD L) (SASL-NIL))))
             (BTM)))

(DEFN REVERSE (L)
    (IF (FINLISTP L)
        (IF (EQUAL L (SASL-NIL))
            (SASL-NIL)
            (APPEND (REVERSE (FINTL L))
                    (FINPAIR (FINHD L) (SASL-NIL))))
        (BTM)))

(PROVE-LEMMA NUMBERP-LEN-APP (REWRITE)
             (IMPLIES (AND (NUMBERP (LEN X))
                           (NUMBERP (LEN Y)))
                      (NUMBERP (LEN (APPEND X Y)))))

(PROVE-LEMMA REVERSE-GEN (GENERALIZE)
             (IMPLIES (NUMBERP (LEN X))
                      (NUMBERP (LEN (REVERSE X)))))

(PROVE-LEMMA LEN-REV (REWRITE)
     (IMPLIES (NOT (EQUAL (LEN L) (BTM)))
              (EQUAL (REVERSE (REVERSE L))
                     L)))
```

```
(PROVE-LEMMA SASL-REVERSE-IS-REVERSE (REWRITE)
             (IMPLIES (NLISTP X)
                      (EQUAL (SASL-REVERSE X)
                             (REVERSE X))))

(PROVE-LEMMA SASL-REV-REV (REWRITE)
             (IMPLIES (NLISTP X)
                      (IMPLIES (NOT (EQUAL (SASL-LENGTH X)
                                           (BTM)))
                               (EQUAL (SASL-REVERSE (SASL-REVERSE X))
                                      X))))

(LIFT SASL-REV-REV-LIFTED (REWRITE) SASL-REV-REV (X))
```

```
(SASL-DEFN SASL-FROM (N)
           (SASL-CONS N (SASL-FROM (SASL-PLUS 1 N))))

(DEFN TYPE-HELP (K N)
  (IF (NUMBERP K)
      (IF (EQUAL K 0)
          0
          (TYPE-HELP (SUB1 K) (ADD1 N)))
      0))

(PROVE-LEMMA TYPE-OF-SASL-FROM (REWRITE)
    (IMPLIES (AND (NUMBERP N)
                  (NUMBERP (REAL-LENGTH (SASL-FROM N)))
                  (NUMBERP K))
             (AND (EQUAL (GREATERP (REAL-LENGTH (SASL-FROM N)) K) T)
                  (NUMBERP (REAL-LENGTH (SASL-FROM (ADD1 N))))))
    ((INDUCT (TYPE-HELP K N))))

(PROVE-LEMMA NOT-NUMBERP-SASL-FROM (REWRITE)
             (IMPLIES (NUMBERP N)
                      (EQUAL (NUMBERP (REAL-LENGTH (SASL-FROM N))) F))
             ((USE (TYPE-OF-SASL-FROM (K (REAL-LENGTH (SASL-FROM N)))))
              (DISABLE TYPE-OF-SASL-FROM)))

(PROVE-LEMMA CORRECT-TYPE-OF-SASL-FROM (REWRITE)
             (OR (FINLISTP (SASL-FROM N))
                 (LISTP (SASL-FROM N)))
             ((USE (SASL-FROM))))

(PROVE-LEMMA FINPAIRS-HAVE-FINITE-REAL-LENGTH (REWRITE)
             (IMPLIES (FINLISTP L)
                      (EQUAL (NUMBERP (REAL-LENGTH L)) T))
             ((INDUCT (APPEND L Z))))

(PROVE-LEMMA SASL-FROM-YIELDS-LISTP (REWRITE)
             (IMPLIES (NUMBERP N)
                      (EQUAL (LISTP (SASL-FROM N)) T))
             ((USE (NOT-NUMBERP-SASL-FROM)
                   (FINPAIRS-HAVE-FINITE-REAL-LENGTH (L (SASL-FROM N))))))
```

```
(PROVE-LEMMA SASL-FROM-YIELDS-INFLIST (REWRITE)
             (IMPLIES (NUMBERP N)
                      (EQUAL (INF-LIST (SASL-FROM N)) T))
             ((DISABLE NOT-NUMBERP-SASL-FROM)
              (USE (NOT-NUMBERP-SASL-FROM))))

(PROVE-LEMMA FROM-SUBSC (REWRITE)
             (IMPLIES (AND (NUMBERP N)
                           (NUMBERP K)
                           (LESSP 0 K))
                      (EQUAL (SUBSC (SASL-FROM N) K)
                             (SUB1 (PLUS N K))))
             ((INDUCT (TYPE-HELP K N))))
```

APPENDIX 4: THE ALTERNATE INITIAL SASL LIBRARY

```
(ADD-SHELL BTM NIL BTMP NIL)

(DCL SASL-CHAR (X))

(DCL SASL-DECODE (X))

(DCL SASL-CODE (X))

;  Here is a shell for finite trees whose leaves are all atoms,
;  i.e. are not bottom, infinite trees, or functions.

(ADD-SHELL SCONS SASL-NIL SLIST
   ((SHD
      (NONE-OF BTMP LISTP LITATOM)
      ; could have been (ONE-OF SCONS TRUEP FALSEP NUMBERP NEGATIVEP CHARP)
      SASL-NIL)  ; notice that (SHD (SASL-NIL)) = (SASL-NIL), which is OK
                 ; since SHD will only be called on SLISTs that are not
                 ; (SASL-NIL)
    (STL
      (ONE-OF SLIST)
      SASL-NIL)))

;  In this approach I'll merely DECLARE the SASL-CONS function, and add its
;  definition as a rewrite axiom.  This will make it easier to disable
;  SASL-CONS.  But I'll do this by means of the auxiliary function
;  SASL-CONS0 in order to take advantage of the type-prescription machinery.

(DCL SASL-CONS (X Y))

(DCL LCONS (X Y))  ; for the leftover cases not covered by SCONS

(DEFN SASL-CONS0 (X Y)
  (IF (OR (LISTP X)
          (LITATOM X)
          (EQUAL X (BTM))
          (NOT (SLIST Y)))
      (LCONS X Y)
      (SCONS X Y)))

(ADD-AXIOM SASL-CONS-DEF (REWRITE)
   (EQUAL (SASL-CONS X Y)
          (SASL-CONS0 X Y)))

(ADD-AXIOM CONS-INF-LISTP (REWRITE)
           (LISTP (LCONS X Y)))

;  The following abbreviation should add substantially to readability and
;  may also be useful in proving theorems.  As with SASL-CONS, I'll actually
;  declare it and add its definition as a rewrite axiom.
```

```
(DEFN FINDEF0 (X)
  (AND (NOT (EQUAL X (BTM)))
       (NOT (LISTP X))
       (NOT (LITATOM X))))

(DCL FINDEF (X))

(ADD-AXIOM FINDEF-DEF (REWRITE)
           (EQUAL (FINDEF X)
                  (FINDEF0 X)))

(ADD-AXIOM DECODE-TYPE-1 (REWRITE)
           (NOT (OR (LISTP (SASL-DECODE X))
                    (LITATOM (SASL-DECODE X)))))

(ADD-AXIOM DECODE-TYPE-2 (REWRITE)
           (IMPLIES (AND (GREATERP X 0)
                         (LESSP X 129))
                    (AND (NOT (BTMP (SASL-DECODE X)))
                         (NOT (EQUAL (SASL-DECODE X) (BTM))))))

(ADD-AXIOM CHAR-DECODE-1 (REWRITE)
           (SASL-CHAR (SASL-DECODE X)))

(ADD-AXIOM CHAR-DECODE-2 (REWRITE)
           (IMPLIES (AND (GREATERP X 0)
                         (LESSP X 129))
                    (EQUAL (SASL-CHAR (SASL-DECODE X)) T)))

(ADD-AXIOM SASL-CHAR-TYPE-1 (REWRITE)
           (IMPLIES (AND (NOT (EQUAL (SASL-CHAR X) T))
                         (NOT (EQUAL X (BTM))))
                    (EQUAL (SASL-CHAR X) F)))

(ADD-AXIOM SASL-CHAR-TYPE-2 (REWRITE)
           (IMPLIES (OR (SLIST X)
                        (LISTP X)
                        (NUMBERP X)
                        (EQUAL X T)
                        (EQUAL X F))
                    (EQUAL (SASL-CHAR X) F)))

(ADD-AXIOM SASL-CHAR-TYPE-3 NIL
           (IMPLIES (AND (NEGATIVEP X)
                         (NOT (EQUAL X (MINUS 0))))
                    (EQUAL (SASL-CHAR X ) F)))

(DCL LHD (X))

(DEFN SASL-HD0 (X)
  (IF (SLIST X)
      (IF (EQUAL X (SASL-NIL))
          (BTM)
          (SHD X))
      (LHD X)))

(DCL SASL-HD (X))
```

```
(ADD-AXIOM SASL-HD-DEF (REWRITE)
           (EQUAL (SASL-HD X)
                  (SASL-HD0 X)))

(DCL LTL (X))

(ADD-AXIOM LTL-TYPE (REWRITE)
           (OR (LISTP (LTL X))
               (SLIST (LTL X))
               (EQUAL (LTL X) (BTM))))

(DEFN SASL-TL0 (X)
  (IF (SLIST X)
      (IF (EQUAL X (SASL-NIL))
          (BTM)
          (STL X))
      (LTL X)))

(DCL SASL-TL (X))

(ADD-AXIOM SASL-TL-DEF (REWRITE)
           (EQUAL (SASL-TL X)
                  (SASL-TL0 X)))

(PROVE-LEMMA SASL-TL-TYPE (REWRITE)
             (OR (LISTP (SASL-TL X))
                 (SLIST (SASL-TL X))
                 (EQUAL (SASL-TL X) (BTM))))

(DCL AP (FN X))

(DEFN SASL-LIST0 (X)
  (IF (EQUAL X (BTM))
      (BTM)
      (IF (SLIST X)
          T
          (LISTP X))))

(DCL SASL-LIST (X))

(ADD-AXIOM SASL-LIST-DEF (REWRITE)
           (EQUAL (SASL-LIST X)
                  (SASL-LIST0 X)))

(PROVE-LEMMA SASL-LIST-TYPE (REWRITE)
             (OR (EQUAL (SASL-LIST X) T)
                 (EQUAL (SASL-LIST X) F)
                 (EQUAL (SASL-LIST X) (BTM))))

; Most of the rest of the events are quite analogous to the
; ones in the earlier version (Appendix 2), so we omit them
; here.  Below is a criterion for equality, FINDEF-EQUAL-EQUAL,
; which is very useful.  (First, we establish the induction used
; for its proof.)  We conclude with two other useful lemmas.
```

```
(DEFN FINDEF-EQUAL-EQUAL-IND (X Y)
  (IF (OR (NOT (SLIST X))
          (EQUAL X (SASL-NIL))
          (NOT (SLIST Y))
          (EQUAL Y (SASL-NIL)))
      T
      (AND (FINDEF-EQUAL-EQUAL-IND (SHD X) (SHD Y))
           (FINDEF-EQUAL-EQUAL-IND (STL X) (STL Y)))))

(PROVE-LEMMA FINDEF-EQUAL-EQUAL (REWRITE)
             (IMPLIES (AND (FINDEF X)
                           (FINDEF Y))
                      (EQUAL (SASL-EQUAL X Y)
                             (EQUAL X Y)))
             ((INDUCT (FINDEF-EQUAL-EQUAL-IND X Y))))

(PROVE-LEMMA FINDEF-TL (REWRITE)
             (IMPLIES (AND (SLIST L)
                           (NOT (EQUAL L (SASL-NIL))))
                      (AND (SLIST (STL L))
                           (FINDEF (STL L)))))

(PROVE-LEMMA FINDEF-HD (REWRITE)
             (IMPLIES (AND (SLIST L)
                           (NOT (EQUAL L (SASL-NIL))))
                      (FINDEF (SHD L))))


;  It's important to note that with this sequence of events,
;  the lifting principle is no longer sound.
```

APPENDIX 5:
A MECHANICALLY-CHECKED PROOF OF CORRECTNESS OF A SASL PATTERN-MATCHING PROGRAM

This sequence of events is organized into parts as follows.

Part I: Definitions of the relevant functions, including lemmas on the equivalences between the SASL functions and their corresponding "finite analogues". For example, early on we define a SASL function MATCH and then a corresponding function MATCH-, and then prove that these agree on finite definite lists of strings (see MATCH-IS-MATCH some time later). The "finite analogues" are generally closely related to their "parents", but they use the theorem-prover's boolean functions IF, AND, and so on rather than the more complex SASL versions, and they also use the "strict" list-processing functions SCONS, SHD, STL.

Part II: Proof of one direction of correctness of pattern-matching function MATCH:

```
(SLIST P)  (SLIST D)  (SLIST L)
(STRINGLISTO P)  (STRINGLISTO D)
(EXPANSION P L)  ( (FLATTEN L) = D)
   ==>  (MATCH P D) = T
```

Part III: Proof of other direction of correctness of MATCH:

```
(SLIST P) /\ (SLIST D) /\
(STRINGLISTO P) /\ (STRINGLISTO D) /\ (MATCH P D)
  ==>  for some L,
         [ (EXPANSION P L) = T  /\  (FLATTEN L) = T ].
```

(In fact, this L is defined as a function (W P D) of P and D.)

These events are run in the context of the alternate initial SASL library which is summarized in Appendix 4. They are taken from Kaufmann [5].

```
; PART I:  Definitions of the relevant functions -- general set-up.


(SASL-DEFN Q ()  ;; the string "?"
           (SASL-CONS (SASL-DECODE 63) (SASL-NIL)))

(SASL-DEFN STAR ()   ;; the string "*"
           (SASL-CONS (SASL-DECODE 42) (SASL-NIL)))

(SASL-DEFN MATCH (P D)
  (SASL-IF
    (SASL-AND (SASL-EQUAL P (SASL-NIL))
              (SASL-EQUAL D (SASL-NIL)))
    T    ;; empty lists match
    (SASL-IF
      (SASL-OR (SASL-EQUAL P (SASL-NIL))
               (SASL-EQUAL D (SASL-NIL)))
      F    ;; because one string is NIL but the other is not
      (SASL-IF
        (SASL-EQUAL (SASL-HD P) (STAR))
        (SASL-OR (MATCH (SASL-TL P) (SASL-TL D))
                 (MATCH P (SASL-TL D)))
        (SASL-IF
          (SASL-OR (SASL-EQUAL (SASL-HD P) (Q))
                   (SASL-EQUAL (SASL-HD P) (SASL-HD D)))
          (MATCH (SASL-TL P) (SASL-TL D))
          F)))))

(DEFN MATCH- (P D)
  (IF (OR (NOT (SLIST P))
          (NOT (SLIST D)))
      F
      (IF (AND (EQUAL P (SASL-NIL))
               (EQUAL D (SASL-NIL)))
          T
          (IF (OR (EQUAL P (SASL-NIL))
                  (EQUAL D (SASL-NIL)))
              F
              (IF (EQUAL (SHD P) (STAR))
                  (OR (MATCH- (STL P) (STL D))
                      (MATCH- P (STL D)))
                  (IF (OR (EQUAL (SHD P) (Q))
                          (EQUAL (SHD P) (SHD D)))
                      (MATCH- (STL P) (STL D))
                      F))))))

(SASL-DEFN STRING (L)
           (SASL-IF (SASL-NOT (SASL-LIST L))
                    F
                    (SASL-IF (SASL-EQUAL L (SASL-NIL))
                             T
                             (SASL-AND (SASL-CHAR (SASL-HD L))
                                       (STRING (SASL-TL L))))))
```

```
(DEFN CHARP (X)
  (EQUAL (SASL-CHAR X) T))

(DEFN STRING- (L)
  (IF (NOT (SLIST L))
      F
      (IF (EQUAL L (SASL-NIL))
          T
          (AND (CHARP (SHD L))
               (STRING- (STL L))))))

(PROVE-LEMMA Q-IS-STRING (REWRITE)
             (STRING- (SCONS (SASL-DECODE 63) (SASL-NIL))))

(PROVE-LEMMA STAR-IS-STRING (REWRITE)
             (STRING- (SCONS (SASL-DECODE 42) (SASL-NIL))))

(PROVE-LEMMA STRING-IS-STRING (REWRITE)
             (IMPLIES (FINDEF L)
                      (EQUAL (STRING L)
                             (STRING- L))))

(SASL-DEFN STRINGLISTO (L)
  (SASL-IF (SASL-NOT (SASL-LIST L))
           F
           (SASL-IF (SASL-EQUAL L (SASL-NIL))
                    T
                    (SASL-AND (STRING (SASL-HD L))
                              (STRINGLISTO (SASL-TL L))))))

(DEFN STRINGLISTO- (L)
  (IF (NOT (SLIST L))
      F
      (IF (EQUAL L (SASL-NIL))
          T
          (AND (STRING- (SHD L))
               (STRINGLISTO- (STL L))))))

(PROVE-LEMMA STRINGLISTO-IS-STRINGLISTO (REWRITE)
             (IMPLIES (FINDEF L)
                      (EQUAL (STRINGLISTO L)
                             (STRINGLISTO- L))))

(PROVE-LEMMA STRINGLISTO-IS-STRINGLISTO-AGAIN (REWRITE)
             (IMPLIES (SLIST L)
                      (EQUAL (STRINGLISTO L)
                             (STRINGLISTO- L))))

(SASL-DEFN STRINGLIST (L)
           (SASL-AND (STRINGLISTO L)
                     (SASL-NOT (SASL-EQUAL L (SASL-NIL)))))
```

```
(DEFN STRINGLIST- (L)
  (AND (STRINGLISTO- L)
       (NOT (EQUAL L (SASL-NIL)))))

(PROVE-LEMMA STRINGLIST-IS-STRINGLIST (REWRITE)
             (IMPLIES (FINDEF L)
                      (EQUAL (STRINGLIST L)
                             (STRINGLIST- L))))

(PROVE-LEMMA MATCH-IS-MATCH (REWRITE)
             (IMPLIES (AND (FINDEF P)
                           (FINDEF D)
                           (EQUAL (STRINGLISTO P) T)
                           (EQUAL (STRINGLISTO D) T))
                      (EQUAL (MATCH P D)
                             (MATCH- P D)))
             ((EXPAND (MATCH (SCONS (SCONS (SASL-DECODE 42) (SASL-NIL))
                                    (SASL-NIL))
                             (STL D)))
              (INDUCT (MATCH- P D))))

;; The following function returns true when the stringlist P matches up
;; with the list L, in the sense that the two lists have the same
;; length, each non-STAR coordinate of P corresponds to a stringlist
;; at the same position in L, every Q in P corresponds to an
;; arbitrary string in L, and the lists P and L agree at all other
;; positions.
;;    Notice the use of STRINGLIST rather than STRINGLISTO, as
;; commented in the following definition.  That's because STAR
;; matches a positive (not zero!) number of strings.

(SASL-DEFN EXPANSION (P L)
  (SASL-IF
    (SASL-AND (SASL-EQUAL P (SASL-NIL))
              (SASL-EQUAL L (SASL-NIL)))
    T
    (SASL-IF
      (SASL-OR (SASL-EQUAL P (SASL-NIL))
               (SASL-EQUAL L (SASL-NIL)))
      F
      (SASL-IF
        (SASL-EQUAL (SASL-HD P) (STAR))
        (SASL-AND (STRINGLIST (SASL-HD L)) ;not STRINGLISTO, cf. above
                  (EXPANSION (SASL-TL P) (SASL-TL L)))
        (SASL-IF
          (SASL-EQUAL (SASL-HD P) (Q))
          (SASL-AND (STRING (SASL-HD L))
                    (EXPANSION (SASL-TL P) (SASL-TL L)))
          (SASL-IF (SASL-EQUAL (SASL-HD P) (SASL-HD L))
                   (EXPANSION (SASL-TL P) (SASL-TL L))
                   F))))))
```

```
(DEFN EXPANSION- (P L)
  (IF (OR (NOT (SLIST P))
          (NOT (SLIST L)))
      F
      (IF (AND (EQUAL P (SASL-NIL))
               (EQUAL L (SASL-NIL)))
          T
          (IF (OR (EQUAL P (SASL-NIL))
                  (EQUAL L (SASL-NIL)))
              F
              (IF (EQUAL (SHD P) (STAR))
                  (AND (STRINGLIST- (SHD L))
                       (EXPANSION- (STL P) (STL L)))
                  (IF (EQUAL (SHD P) (Q))
                      (AND (STRING- (SHD L))
                           (EXPANSION- (STL P) (STL L)))
                      (IF (EQUAL (SHD P) (SHD L))
                          (EXPANSION- (STL P) (STL L))
                          F)))))))

(PROVE-LEMMA EXPANSION-IS-EXPANSION (REWRITE)
   (IMPLIES (AND (SLIST P)
                 (SLIST L))
            (EQUAL (EXPANSION P L)
                   (EXPANSION- P L)))
   ((DISABLE STRING STRINGLIST)))  ; such hints tend to save time

(SASL-DEFN APPEND (L M)
           (SASL-IF (SASL-LIST L)
                    (SASL-IF (SASL-EQUAL L (SASL-NIL))
                             (SASL-IF (SASL-LIST M)
                                      M
                                      (BTM))
                             (SASL-CONS (SASL-HD L)
                                        (APPEND (SASL-TL L) M)))
                    (BTM)))

(DEFN APPEND- (L M)
  (IF (NOT (SLIST L))
      M  ; this ruins equivalence of APPEND functions over non-lists,
         ; but avoids the need to introduce BTM needlessly
      (IF (EQUAL L (SASL-NIL))
          M
          (SCONS (SHD L) (APPEND- (STL L) M)))))

(PROVE-LEMMA APPEND-IS-APPEND (REWRITE)
             (IMPLIES (AND (SLIST L)
                           (SLIST M))
                      (EQUAL (APPEND L M)
                             (APPEND- L M))))
```

```
(SASL-DEFN FLATTEN (L)
  (SASL-IF (SASL-NOT (SASL-LIST L))
           (BTM)
           (SASL-IF (SASL-EQUAL L (SASL-NIL))
                    (SASL-NIL)
                    (SASL-IF (STRINGLIST (SASL-HD L))
                             (APPEND (SASL-HD L)
                                     (FLATTEN (SASL-TL L)))
                             (SASL-CONS (SASL-HD L)
                                        (FLATTEN (SASL-TL L)))))))

(DEFN FLATTEN- (L)
  (IF (NOT (SLIST L))
      (SASL-NIL)
      (IF (EQUAL L (SASL-NIL))
          (SASL-NIL)
          (IF (STRINGLIST- (SHD L))
              (APPEND- (SHD L) (FLATTEN- (STL L)))
              (SCONS (SHD L) (FLATTEN- (STL L)))))))

(PROVE-LEMMA FLATTEN-SLIST (REWRITE)
             (IMPLIES (SLIST L)
                      (SLIST (FLATTEN- L))))

(PROVE-LEMMA FLATTEN-IS-FLATTEN (REWRITE)
  (IMPLIES (SLIST L)
           (EQUAL (FLATTEN L)
                  (FLATTEN- L)))
  ((EXPAND (STRINGLISTO- (SHD L)))))  ; needed in order to apply
                                      ; APPEND-IS-APPEND
```

```
; PART II:  One direction of correctness.

;;; Our approach will be to prove the desired result (as described in
;;; the introduction) in terms of the "finite analogues" first.  We'll
;;; then rely on the various equivalences proved in Part I in order to
;;; obtain the desired results.

;;; The following function P1 is an abbreviation for the aforementioned
;;; "finite analogue" of the desired result.

(DEFN P1 (P L D)
  (IMPLIES (AND (STRINGLISTO- P)
                (STRINGLISTO- D)
                (EXPANSION- P L)
                (EQUAL (FLATTEN- L) D))
           (MATCH- P D)))

(DEFN CONS1 (A L)
      (SCONS (SCONS A (SHD L)) (STL L)))

(DISABLE MATCH)

(DEFN TAIL1 (L)
  (SCONS (STL (SHD L))
         (STL L)))

;;; We will prove the desired result by induction as indicated by the
;;; following definition.

(DEFN P1-IND (P L D)
  (IF (OR (NOT (SLIST P))
          (NOT (SLIST L))
          (NOT (SLIST D)))
      T
      (IF (EQUAL P (SASL-NIL))
          T
          (IF (EQUAL L (SASL-NIL))
              T
              (IF (EQUAL D (SASL-NIL))
                  T
                  (IF (EQUAL (SHD P) (STAR))
                      (IF (EQUAL (STL (SHD L)) (SASL-NIL))
                          (P1-IND (STL P) (STL L) (STL D))
                          (P1-IND P (TAIL1 L) (STL D)))
                      (P1-IND (STL P) (STL L) (STL D))))))))

;;; The approach now is to prove each of the induction cases, one
;;; at a time.

(PROVE-LEMMA P1-1 (REWRITE)
             (IMPLIES (OR (NOT (SLIST P))
                          (NOT (SLIST L))
                          (NOT (SLIST D)))
                      (P1 P L D)))
```

```
(PROVE-LEMMA P1-2 (REWRITE)
             (IMPLIES (EQUAL P (SASL-NIL))
                      (P1 P L D)))

(PROVE-LEMMA P1-3 (REWRITE)
             (IMPLIES (EQUAL L (SASL-NIL))
                      (P1 P L D)))

(PROVE-LEMMA P1-4 (REWRITE)
             (IMPLIES (AND (EQUAL D (SASL-NIL))
                           (NOT (EQUAL P (SASL-NIL)))
                           (NOT (EQUAL L (SASL-NIL))))
                      (P1 P L D)))

(PROVE-LEMMA NIL-FLATTEN (REWRITE)
             (IMPLIES (AND (SLIST V)
                           (EQUAL (FLATTEN- V) (SASL-NIL)))
                      (EQUAL (EQUAL V (SASL-NIL)) T)))

(PROVE-LEMMA P1-5 (REWRITE)
             (IMPLIES (AND (NOT (EQUAL P (SASL-NIL)))
                           (NOT (EQUAL L (SASL-NIL)))
                           (EQUAL (SHD P) (STAR))
                           (EQUAL (STL (SHD L)) (SASL-NIL))
                           (P1 (STL P) (STL L) (STL D)))
                      (P1 P L D)))

(PROVE-LEMMA APPEND-NOT-NIL (REWRITE)
             (IMPLIES (AND (SLIST L)
                           (NOT (EQUAL L (SASL-NIL))))
                      (NOT (EQUAL (APPEND- L M) (SASL-NIL)))))

(PROVE-LEMMA P1-6 (REWRITE)
             (IMPLIES (AND (NOT (EQUAL P (SASL-NIL)))
                           (NOT (EQUAL L (SASL-NIL)))
                           (EQUAL (SHD P) (STAR))
                           (NOT (EQUAL (STL (SHD L)) (SASL-NIL)))
                           (P1 P (TAIL1 L) (STL D)))
                      (P1 P L D)))

(PROVE-LEMMA STRING-STRINGLIST (REWRITE)
             (IMPLIES (AND (STRINGLIST0- L)
                           (STRING- L))
                      (EQUAL (EQUAL L (SASL-NIL)) T)))

(PROVE-LEMMA P1-7 (REWRITE)
             (IMPLIES (AND (NOT (EQUAL P (SASL-NIL)))
                           (NOT (EQUAL L (SASL-NIL)))
                           (NOT (EQUAL (SHD P) (STAR)))
                           (P1 (STL P) (STL L) (STL D)))
                      (P1 P L D)))

(DISABLE NIL-FLATTEN)

(DISABLE STRING-STRINGLIST)

(DISABLE APPEND-NOT-NIL)
```

```
(PROVE-LEMMA SUFFICIENCY-FOR-MATCH-LEMMA NIL
             (P1 P L D)
             ((INDUCT (P1-IND P L D))
              (DISABLE STRINGLISTO- EXPANSION- FLATTEN- MATCH- P1)))

;; Notice that in the following theorem, the third through fifth
;; hypotheses are weaker than what would be sufficient for our
;; purposes; for example, "(STRINGLISTO P)" means the same thing as
;; "(NOT (EQUAL (STRINGLISTO P) F))", which is weaker than asserting
;; "(EQUAL (STRINGLISTO P) T)".  Of course, weakening the hypotheses
;; can only strengthen the theorem -- i.e. there's no harm in doing so.

; ************************************************************************
; HALF OF THE THEOREM
; ************************************************************************

(PROVE-LEMMA SUFFICIENCY-FOR-MATCH (REWRITE)
             (IMPLIES (AND (SLIST P)
                           (SLIST D)
                           (SLIST L)
                           (STRINGLISTO P)
                           (STRINGLISTO D)
                           (EXPANSION P L)
                           (EQUAL (FLATTEN L) D))
                      (EQUAL (MATCH P D) T))
             ((USE (SUFFICIENCY-FOR-MATCH-LEMMA))))
```

```
; Part III:  The other direction.


(DISABLE P1-1)

(DISABLE P1-2)

(DISABLE P1-3)

(DISABLE P1-4)

(DISABLE P1-5)

(DISABLE P1-6)

(DISABLE P1-7)

(DISABLE SUFFICIENCY-FOR-MATCH)

;;; The "other direction" says that if pattern  P  matches stringlist
;;; D, then an appropriate list  L  exists, i.e. there is a list  L
;;; such that the hypotheses of the theorem above hold.  To find this
;;; list L, we need to define the appropriate witnessing function.  Then
;;; we will prove:

;  (IMPLIES (AND (STRINGLISTO- P)
;                (STRINGLISTO- D)
;                (MATCH- P D))
;           (AND (EXPANSION- P L)
;                (EQUAL (FLATTEN- L) D)))

;;; where  L = (W P D).

(DEFN W (P D)
  (IF (OR (NOT (SLIST P))     ;; This case
          (NOT (SLIST D)))    ;; should
      (SASL-NIL)              ;; never arise
      (IF (AND (EQUAL P (SASL-NIL))
               (EQUAL D (SASL-NIL)))
          (SASL-NIL)
          (IF (OR (EQUAL P (SASL-NIL))    ;; This case
                  (EQUAL D (SASL-NIL)))   ;; should
              (SASL-NIL)                  ;; never arise
              (IF (EQUAL (SHD P) (STAR))
                  (IF (MATCH- (STL P) (STL D))
                      (SCONS (SCONS (SHD D) (SASL-NIL))
                             (W (STL P) (STL D)))
                      (CONS1 (SHD D) (W P (STL D))))
                  (SCONS (SHD D) (W (STL P) (STL D)))))))))

;; The following lemma went through on the first try, with no hints!
```

```
(PROVE-LEMMA W-EXPANSION (REWRITE)
             (IMPLIES (AND (STRINGLISTO- P)
                           (STRINGLISTO- D)
                           (MATCH- P D))
                      (EXPANSION- P (W P D))))

(DISABLE W-EXPANSION)

;;; Let's let P2 name the second half of our desired goal.

(DEFN P2 (P D)
  (IMPLIES (AND (STRINGLISTO- P)
                (STRINGLISTO- D)
                (MATCH- P D))
           (EQUAL (FLATTEN- (W P D)) D)))

;;; We'll prove  P2  by induction along the definition of  W.

(PROVE-LEMMA P2-1 (REWRITE)
             (IMPLIES (OR (NOT (SLIST P))
                          (NOT (SLIST D)))
                      (P2 P D)))

(PROVE-LEMMA P2-2 (REWRITE)
             (IMPLIES (AND (EQUAL P (SASL-NIL))
                           (EQUAL D (SASL-NIL)))
                      (P2 P D)))

(PROVE-LEMMA P2-3 (REWRITE)
             (IMPLIES (OR (EQUAL P (SASL-NIL))
                          (EQUAL D (SASL-NIL)))
                      (P2 P D)))

(PROVE-LEMMA P2-4 (REWRITE)
             (IMPLIES (AND (EQUAL (SHD P) (STAR))
                           (MATCH- (STL P) (STL D))
                           (P2 (STL P) (STL D)))
                      (P2 P D)))

(PROVE-LEMMA P2-5-LEMMA-1 (REWRITE)
             (IMPLIES (AND (STRINGLISTO- (STL P))
                           (STRINGLISTO- D)
                           (EQUAL (SHD P) (STAR))
                           (MATCH- P D))
                      (STRINGLISTO- (SHD (W P D)))))

(PROVE-LEMMA P2-5-LEMMA-2 (REWRITE)
             (IMPLIES (AND (MATCH- P D)
                           (NOT (EQUAL P (SASL-NIL)))
                           (EQUAL (SHD P) (STAR)))
                      (NOT (EQUAL (SHD (W P D))
                                  (SASL-NIL)))))
```

```
(PROVE-LEMMA P2-5 (REWRITE)
             (IMPLIES (AND (EQUAL (SHD P) (STAR))
                           (NOT (MATCH- (STL P) (STL D)))
                           (P2 P (STL D)))
                      (P2 P D)))

(ENABLE STRING-STRINGLIST)

(PROVE-LEMMA P2-6 (REWRITE)
             (IMPLIES (AND (NOT (EQUAL P (SASL-NIL)))
                           (NOT (EQUAL D (SASL-NIL)))
                           (NOT (EQUAL (SHD P) (STAR)))
                           (P2 (STL P) (STL D)))
                      (P2 P D)))

(DISABLE P2-5-LEMMA-1)

(DISABLE P2-5-LEMMA-2)

(PROVE-LEMMA P2-ALWAYS-HOLDS NIL
             (P2 P D)
             ((INDUCT (W P D))
              (DISABLE P2 FLATTEN- MATCH- STRINGLISTO- W)))

(ENABLE W-EXPANSION)

;; See the comment proceeding the previous direction, which applies
;; here as well.

; ***************************************************************************
;  OTHER HALF OF THE THEOREM
; ***************************************************************************

(PROVE-LEMMA NECESSITY-OF-MATCH (REWRITE)
             (IMPLIES (AND (SLIST P)
                           (SLIST D)
                           (STRINGLISTO P)
                           (STRINGLISTO D)
                           (MATCH P D))
                      (AND (EQUAL (EXPANSION P (W P D)) T)
                           (EQUAL (FLATTEN (W P D)) D)))
             ((USE (P2-ALWAYS-HOLDS))
              (DISABLE  MATCH- EXPANSION- FLATTEN- W)))
```

APPENDIX 6: SASL DEFINITIONS FOR SASL UNIFICATION FUNCTION

```
STRING L =
  (SASL-LIST L) &
  (L = [] | ((CHAR (HD L)) & (STRING (TL L))))

CHAR-V  = 'V'

VAR L = (STRING L) & (~ L = []) & (HD L = CHAR-V)

CONST L = (STRING L) & (~ VAR L)

FAIL  = "FAIL"

SECOND L = HD (TL L)

MK-PAIR X Y = [X,Y]

APPEND L M = LIST L -> (L=[] -> (LIST M -> M; BOTTOM);
                        HD L : APPEND (TL L) M);
             BOTTOM

OCCURS V L = (VAR L) -> V=L;
             (CONST L) -> FALSE;
             OCCURS V (HD L) | OCCURS V (TL L)

IDENT  = []

MK-SUB V L = MK-PAIR V L : []

SASL-LOOKUP TABLE DEFAULT OBJECT =
  TABLE = [] -> DEFAULT;
  OBJECT = (HD (HD TABLE)) -> SECOND (HD TABLE);
  SASL-LOOKUP (TL TABLE) DEFAULT OBJECT

VAR-SUBST SUB V = SASL-LOOKUP SUB V V

SUBST SUB L = VAR L -> VAR-SUBST SUB L;
              CONST L -> L;
              SUBST SUB (HD L) : SUBST SUB (TL L)

MAP FN L = L=[] -> [];
           LIST L -> FN (HD L) : MAP FN (TL L);
           BOTTOM

SUBST2 SUB L = MK-PAIR (HD L) (SUBST SUB (SECOND L))

VAR-UNIFY V L = V=L -> IDENT;
                OCCURS V L -> FAIL;
                MK-SUB V L
```

```
COMPOSE SUB1 SUB2 = SUB1=FAIL | SUB2=FAIL -> FAIL;
                    APPEND (MAP (SUBST2 SUB2) SUB1) SUB2

UNIFY L M = VAR L -> VAR-UNIFY L M;
            VAR M -> VAR-UNIFY M L;
            (CONST L) | (CONST M) -> (L=M -> IDENT; FAIL);
            COMPOSE SUBHD SUBTL
            WHERE
            SUBHD = UNIFY (HD L) (HD M)
            SUBTL = UNIFY (SUBST SUBHD (TL L))
                          (SUBST SUBHD (TL M))
```

FOOTNOTES

1. "NORMA" is an acronym for Normal Order Reduction MAchine, which is so-named because it is a combinator graph reduction machine which uses the normal-order reduction strategy. This strategy supports what is known as "lazy evaluation", which in turn allows for infinite data structures.

2. Actually, SASL also includes real numbers along with some constructs for handling I/O, which we ignore in this paper. For simplicity, we also ignore some of the syntactic sugar, such as WHERE clauses, pattern-matching, and ZF-expressions (cf. Richards [14], [15]).

3. Certainly a parser from SASL to Lisp would be a nice package to put on top of the Prover, among others.

4. Using more of the constructs available in SASL, we can write these definitions more clearly and elegantly as follows:

```
primes = sieve [2 ..]
sieve (a:L) = a : sieve (filter a L)
filter a (b:L) = (b REM a) = 0 -> filter a L;
                 b : filter a L
```

Still more elegantly:

```
primes = sieve [2 ..]
sieve (a:L) = a : sieve [b <- L; b REM a ~= 0]
```

5. More formally, a type restriction for the ith argument is a term with Xi as the only variable and TRUE, FALSE, previously introduced shell recognizers, and the current shell recognizers as the only function symbols.

6. Though the semantics of SASL are given in [11], any standard reference on denotational semantics (e.g. Stoy [16]) should be adequate to provide a good idea of what is going on. In the jargon of that field, the SASL domain is the (least) solution to the domain equation

$$V = R + C + B + V^* + [V \rightarrow V]$$

where

$$V^* = \underline{nil} + V^+$$
$$V^+ = V \times V^*.$$

7. We may take **(OTHER t)** to be an abbreviation for (NOT (\/ [(**r t**): **r** a recognizer]), though this differs a bit from the system's implementation.

8. Some of these type prescriptions are implemented using the ADD-AXIOM command (in Appendix 2). Others remain unimmplemented, which results in the default type prescription <UNIVERSE,{}>, which is of course the weakest possible and hence does not harm soundness.

9. The material in this section is based on Kaufmann [5].

10. Two minor modifications of the PROVE-LEMMA command were added to the Prover to facilitate the proof. Neither of these affects the logic. The PROVE-LEMMA-TAUT command instructs the Prover to forget all of the rewrite rules that it has stored except for some very basic facts about arithmetic and such, though an optional argument can be given to change this. The PROVE-LEMMA-ALLOW command is similar, except that the user specifies rewrite rules which are not to be thus disabled. Both of these commands are useful for directing a proof. For example, a given complicated fact may follow tautologically from some simpler facts, and to obtain that final fact one would like to turn off the rewrite rules (which can slow things down immensely).

REFERENCES

[1] Boyer, R. S. and Kaufmann, M. On the feasibility of mechanically verifying SASL programs. Technical Report ARC 84-16 (1984), Burroughs Austin Research Center.

[2] R. S. Boyer, R. S. and Moore, J S. *A Computational Logic*. Academic Press, 1979.

[3] Gordon, M. J. C., Milner, A. J., and Wadsworth, C. P. *Edinburgh LCF*. Springer-Verlag, Lec. Notes Comp. Sci. 78, 1979.

[4] Hughes, J. Why functional programming matters. Programming Methodology Group Memo PMG-40, Chalmers Tekniska Hogskola, 41296 Goteborg, Sweden.

[5] Kaufmann, M. A mechanically-checked proof of correctness of a SASL pattern-matching program. Technical Report ARC 85-11 (1985, revised 1/86), Burroughs Corp.

[6] Kaufmann, M. A mechanically-checked proof of correctness of a SASL unification program. Technical Report ARC 85-12 (1985, revised 1/86), Burroughs Corp.

[7] Kaufmann, M. A mechanically-checked proof of correctness of a SASL quicksort program, Technical Report ARC 85-10 (1985), Burroughs Corp.

[8] Kaufmann, M. A direct construction of a SASL domain. Technical Report ARC 84-14 (1984), Burroughs Corp.

[9] Kaufmann, M. A basic introduction to the Boyer-Moore system and its modification for SASL. Technical Report ARC 85-08 (1985), Burroughs Corp.

[10] Kaufmann, M. Theorem-proving for a higher-order functional language (abstract). *J. Symbolic Logic* (to appear).

[11] Kaufmann, M. and Surber, D. Syntax, semantics, and a formal logic for SASL. Technical Report ARC 85-3 (1985), Burroughs Corp.

[12] Manna, Z. and R. Waldinger, R. Deductive synthesis of the unification algorithm. Sci. Comput. Programming 1 (1981), 5-48.

[13] Paulson, L. Verifying the unification algorithm in LCF. Sci. Comput. Programming 5 (1985), 143-169.

[14] Richards, H. *Programming in SASL*. Burroughs Corp., Austin Research Center, 1984.

[15] Richards, H. An overview of ARC SASL. *SIGPLAN Notices 19* (October 1984), 40-45.

[16] Scott, D. Domains for denotational semantics. ICALP '82, Aarhus, Denmark.

[17] Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Mass., 1977.

[18] Turner, D. *Aspects of the Implementation of Programming Languages*. Ph. D. thesis, University of Oxford, February 1981.

[19] D. Turner, Functional programming and proofs of program correctness, in: Neel (ed.), *Tools and Notions for Program Construction*. Cambridge University Press, Cambridge, 1982.

[20] Winston, P. *Artificial Intelligence*. Addison-Wesley, 1977.