Structured Theory Development for a Mechanized Logic[*]

Matt Kaufmann[†] and J Strother Moore[‡]

March 1, 1999

Abstract

Experience has shown that large or multi-user interactive proof efforts can benefit significantly from structuring mechanisms, much like those available in many modern programming languages. Such a mechanism can allow some lemmas and definitions to be exported, and others not. In this paper we address two such structuring mechanisms for the ACL2 theorem prover: *encapsulation* and *books*. After presenting an introduction to ACL2, this paper justifies the implementation of ACL2's structuring mechanisms and, more generally, formulates and proves high-level correctness properties of ACL2. The issues in the present paper are relevant not only for ACL2 but also for other theorem-proving environments.

1 Overview

Modern programming languages provide convenient features even though in principle, all computations can be performed using Turing machines. By analogy, mechanized logics often include convenient features even though conventional mathematical logics, for example first-order logic, provide a firm foundation for formal reasoning. In particular, experience has shown that large or multi-user interactive proof efforts can benefit significantly from structuring mechanisms, much like those available in many modern programming languages.

[*]This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406, and the Office of Naval Research, Contract N00014-94-C-0193. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of EDS, Inc., the University of Texas, Computational Logic, Inc., the Defense Advanced Research Projects Agency, the Office of Naval Research, or the U.S. Government.

[†]EDS CIO Services, 98 San Jacinto Blvd., Suite 500, Austin, TX 78701, kaufmann@cio2000.eds.com

[‡]Dept. of Computer Sciences, University of Texas at Austin, Austin, TX 78712, moore@cs.utexas.edu

In this paper we provide structuring mechanisms, implemented in the ACL2[1] theorem prover [8, 11, 4, 12]. Similar mechanisms might be appropriate for other theorem proving systems.

Like its predecessor Nqthm [1, 5, 3], the ACL2 system supports development of (first-order) theories. Users introduce axioms by way of *extension principles*, which are usually definitions, and establish provability of alleged axioms from those axioms. Unlike Nqthm, the ACL2 theorem prover provides powerful structuring mechanisms that allow the user to designate certain definitions and lemmas as "**`local`**", not to be exported from specified scopes. These mechanisms, in concert with others offered by ACL2, challenge us to formalize what these structuring mechanisms really mean. In fact, unsoundness crept into early versions of ACL2 because of our failure to understand subtle aspects of such structuring mechanisms. In this paper, we give a way to understand what an ACL2 session means in terms of first-order logic.

The rest of this paper proceeds as follows. First, we give a reasonably brief overview of ACL2 and, in particular, of the structuring mechanisms that are the subject of this paper. That section provides the ACL2 prerequisites for the rest of the paper; no prior knowledge of ACL2 is assumed. Next, Section 3 provides an informal introduction without proof of the main result of the paper, that alleged theorems are indeed theorems. Following a presentation of logical preliminaries in Section 4, we present a somewhat new treatment of recursion and induction in Section 5. This treatment is used for explaining why ACL2's structuring mechanisms are sound. Section 6 introduces the crucial notion of *history* and proves some important properties of histories. Section 7 then introduces the notion of *chronology*, and demonstrates that the theorems of a chronology are consequences of its axioms. Finally, we tie things together in Section 8 by showing that ACL2 sessions correspond to chronologies. In particular, the session's theorems are first-order consequences of its axioms, and as a consequence (modulo an obvious restriction) the session's first-order theory is consistent.

There are also two appendices, which address technical points deferred from the main exposition. Appendix A proves the correctness of functional instantiation against the backdrop of histories. The correctness statement encompasses an optimization in the ACL2 implementation of its **`encapsulate`** structuring mechanism. Appendix B proves a relevant technical result on Skolemization.

A preliminary work in the direction of this paper is [9]. That work, and the present paper, stand in contrast to the lower-level presentation provided in [10], in which we lay out the syntax of ACL2 and present a particular logic suited to reasoning without quantification. We also consider in [10] important issues including *packages* and *macros*. In this paper we work at a higher level of abstraction, assuming familiarity only with classical first-order logic and ignoring syntactic and extra-logical issues. The issues in the present paper are relevant

[1] "A Computational Logic for Applicative Common Lisp"

not only for ACL2 but also for other theorem-proving environments.

Prerequisites. We assume no familiarity with ACL2 or Common Lisp, providing the necessary background in Section 2. We do assume some basic familiarity with first-order logic.

Conventions. Definitions and introduction of notation, as well as proofs of lemmas and theorems, conclude with the symbol $\Box$. When the proof is obvious, this symbol appears immediately after the statement of the lemma or theorem. All sequences in this paper are finite sequences. We use `typewriter font` to display actual syntax and we use font such as f both to indicate "meta-variables" that stand for syntax and for mathematical entities other than syntactic elements. Example: "f is a function symbol such as `CONS` appearing in the set S of sentences."

Acknowledgments. Bob Boyer played a major role in the development of ACL2 for its first few years. We gratefully acknowledge his numerous contributions to its design and implementation. Virtually all of the implementation work related to this paper was done in the friendly environment of Computational Logic, Inc. EDS graciously provided Kaufmann some time away from its Year 2000 project to work on this paper.

2 Introduction to ACL2

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." We use the name both for a mathematical logic based on applicative Common Lisp [19] and for a mechanized theorem proving system for that logic. ACL2 is closely related to the Boyer-Moore logic and system and its interactive enhancement [1, 5, 3]. The key reason we abandoned the Nqthm logic and adopted applicative Common Lisp is that the latter can result in extremely efficient runtime code.

2.1 The ACL2 Logic

The kernel of the ACL2 logic consists of a syntax, some rules of inference, and some axioms. The kernel logic is given precisely in [10]. The logic supported by the mechanized ACL2 system is an extension of the kernel logic.

The kernel syntax describes terms composed of variables, constants, and function symbols applied to fixed numbers of argument terms. Thus, `(* X (FACT N))` is a term that might be written as $x \times n!$ in more traditional syntactic systems. After introducing Lisp-like terms, the kernel logic introduces the notion of "formulas" composed of equalities between terms and the usual propositional connectives. There are no quantifiers.

The rules of inference are those for propositional calculus with equality, instantiation, an induction principle and extension principles allowing for the definition of new total recursive functions, new constant symbols, new "symbol packages," and the declaration of the "current package" (used in the resolution of naming conflicts). Our extension principles specify conditions under which the proposed extensions are admissible. For example, recursive definitions must be proved to terminate. The admissibility requirements insure the consistency of the resulting extensions.

The ACL2 axioms describe the properties of certain Common Lisp primitives. For example,

Axioms.
`X` $=$ `Y` $\rightarrow$ `(EQUAL X Y)` $=$ `T`

`X` $\neq$ `Y` $\rightarrow$ `(EQUAL X Y)` $=$ `NIL`

`X` $=$ `NIL` $\rightarrow$ `(IF X Y Z)` $=$ `Z`

`X` $\neq$ `NIL` $\rightarrow$ `(IF X Y Z)` $=$ `Y`

Using the function symbols `EQUAL` and `IF` it is possible to "embed" propositional calculus and equality into the term language of the logic. When we write a term p where a formula is expected it is an abbreviation for the formula $p \neq$ `NIL`. Thus, by

Axiom (CAR-CONS).
`(EQUAL (CAR (CONS X Y)) X)`

we mean

Axiom.
`(EQUAL (CAR (CONS X Y)) X)` $\neq$ `NIL`

which is provably equivalent to

Axiom.
`(CAR (CONS X Y))` $=$ `X`.

We similarly feel free to write $\neg\tau$ to denote the term or formula obtained by applying the function symbol `NOT` to the term τ, where formally:

Axiom.
`(NOT X)` $=$ `(IF X NIL T)`.

The kernel logic includes axioms that characterize the primitive functions for constructing and manipulating certain Common Lisp numbers, characters, strings, symbols, and ordered pairs. In the present work, we consider only theories that extend a certain first-order theory GZ. This theory is assumed to contain the axioms of the kernel logic mentioned above, together with the analogous `CDR-CONS` for `CDR` in place of `CAR`, guaranteeing that the ordered-pair operation `CONS` is one-to-one. It also contains a few other axioms described in Section 4, notably a well-founded induction principle. Notice that the axioms above imply: `T` $\neq$ `NIL`.

The logic supported by the ACL2 system is somewhat richer than the kernel logic. The full logic is obtained from the kernel by (a) a syntactic extension and some syntactic restrictions (b) the inclusion of a new extension principle called "encapsulation" and a derived rule of inference called "functional instantiation," and (c) the inclusion of a new extension principle called "`defchoose`." The syntactic extension is provided via the incorporation of Common Lisp's notion of macros, whereby new syntactic forms are implemented by functions that translate those forms into terms in the kernel syntax. The syntactic restrictions have to do with syntactic limitations on the use of certain primitives so as to allow efficient execution. For example, functions returning multiple values must return the correct number, even though in the kernel logic a bundle of values is just a list and insufficiently long bundles are effectively padded with `nil`s. Encapsulation and functional instantiation are the subject of the present paper. The `defchoose` feature, discussed below, essentially allows the introduction of universal and existential quantifiers.

2.2 The ACL2 System

The ACL2 system presents itself to the user as a read-eval-print command loop.

The most basic commands are those that define new functions, constants and packages or designate a package as the current one. These commands correspond to the extension principles in the kernel logic. Each time such a command is executed the current first-order theory is extended. Such commands are called *events*. A sample event command is shown below. This command defines a function named `APP`. Immediately following the command we show the system's text output in response. The response is mainly concerned with the admissibility requirements for recursive definitions.

```
(defun APP (X Y)     ; concatenate lists x and y
  (IF (CONSP X)
      (CONS (CAR X) (APP (CDR X) Y))
      Y))
```

```
The admission of APP is trivial, using the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT X).  We observe that the type of APP is
described by the theorem (OR (CONSP (APP X Y)) (EQUAL (APP X Y) Y)).
We used primitive type reasoning.
```

Another event command directs the ACL2 system to prove a theorem, and, when successful, to build the theorem into the system's data base. Below is a command to prove that `APP` is associative. The proof, not shown, is successful and the theorem is built-in as a left-to-right rewrite rule.

```
(defthm ASSOCIATIVITY-OF-APP
  (EQUAL (APP (APP A B) C)
         (APP A (APP B C)))
  :rule-classes :rewrite)
```

Loosely speaking, the record of all successful event commands is called the "chronology" of a session. Other commands allow the user to display parts of the chronology and to "undo" commands so as to roll-back the chronology to some prior setting. For example, the cryptic command

```
:ubt! 1
```

undoes back through the first command and thus restores the system to its initial state.

The ACL2 system also supports the *constrained* introduction of new function symbols. That is, it is possible to introduce a new symbol that is constrained to satisfy certain axioms, without providing a definition that uniquely determines the function's behavior. To insure that the constraints are satisfiable, the user must provide a "witness," i.e., an existing function that can be proved to have the required properties. The mechanism by which all this is accomplished in ACL2 is called *encapsulation.* In a certain sense, this paper is about the logical underpinnings of encapsulation – underpinnings that are sufficiently subtle to have caused us to introduce soundness bugs into versions of ACL2 prior to construction of the careful arguments in this paper.

Here is an example encapsulation. Suppose the chronology at the time this command is executed is called α.

```
(encapsulate
 ((EXECUTE (S) T)                    ; Signatures for execute
  (STATEP (S) T))                    ;  and statep
 (local (defun EXECUTE (S) S))       ; Witness for execute
 (local (defun STATEP (S) (NULL S))) ; Witness for statep
 (defthm STATEP-EXECUTE              ; Constraint: execute
   (IMPLIES (STATEP S)               ;  preserves statep
            (STATEP (EXECUTE S)))))
```

The signatures declare that the `encapsulate` command will introduce two function symbols, `EXECUTE` and `STATEP`, each of which takes one argument and returns one result. The next two forms are "local `defun`s" that define `EXECUTE` to be the identity function and `STATEP` to be `NULL`, i.e., to return `T` or `NIL` according to whether its argument is `NIL`. These two events are "local" in the sense that the axioms added by them are relevant only during the admissibility analysis for the `encapsulate`; these axioms provide alleged witnesses for the constraints to be placed on the new symbols. The last event in the `encapsulate` above is a `defthm` event that asserts that `EXECUTE` preserves `STATEP`. Locally, this event is admissible provided it can be proved. The proof is trivial given the witnesses. This theorem will become the constraint on the new symbols.

This encapsulation command is admissible because the constrained symbols are new, the local and non-local events of the body of the `encapsulate` are admissible (so the witnesses satisfy the constraint), and the non-local events use no locally defined symbols other than the constrained ones. During the admission analysis an extension of α, containing the local and non-local events, is produced. Once the admissibility requirements have been met, this extension is of no further interest.

The effect of the above `encapsulate` is to produce an extension of α in which `EXECUTE` and `STATEP` have the syntactic signatures given and are axiomatized to satisfy the formula named `STATEP-EXECUTE`. Very roughly speaking, the constraints of an `encapsulate` are all of the non-local events. In this example, the only constraint is `STATEP-EXECUTE` and the two new functions have no other constraints on them.

Because local events are not "exported" by the encapsulate, we sometimes use encapsulate simply to structure proofs. For example, one might declare no function signatures and then prove a series of `local` lemmas leading to a desired theorem. The effect of such an encapsulation is to add the desired theorem as a consistent axiom without cluttering the ACL2 data base with the lemmas necessary to prove it.

We now return to the example `encapsulate` above. Once this event has been admitted, the user might issue the following additional commands. The first of these commands defines a function, `CYCLE`, which iteratively applies `EXECUTE` a certain number of times. The next two commands prove theorems about `CYCLE`. Here, `ZP` is true of all but the positive integers.

```
(defun CYCLE (S N)
  (IF (ZP N)
      S
    (CYCLE (EXECUTE S) (- N 1))))

(defthm STATEP-CYCLE
  (IMPLIES (STATEP S)
           (STATEP (CYCLE S N))))

(defthm CYCLE-COMPOSITION
  (IMPLIES (AND (INTEGERP I)
                (<= 0 I)
                (INTEGERP J)
                (<= 0 J))
           (EQUAL (CYCLE S (+ I J))
                  (CYCLE (CYCLE S I) J))))
```

The first theorem says that `CYCLE` preserves states. The second shows how to decompose a long run into two shorter ones. These theorems are proved inductively (and fully automatically).

Recall that `STATEP` and `EXECUTE` are merely constrained, not defined. They may be thought of as "generic." The defined function `CYCLE` and the theorems proved are generic in the same sense. ACL2 provides a means by which such generic function symbols can be instantiated.

Suppose we define some concrete notion of a "machine state" and some concrete "execution step" that we prove preserves the concrete notion of state. Then any theorem about the generic functions gives rise to an analogous theorem about the concrete ones. In particular, if we define a concrete notion of a cyclic execution engine we could appeal to `STATEP-CYCLE` and `CYCLE-COMPOSITION` to obtain two important properties of the concrete cyclic engine. Such an appeal is made by stating the desired concrete theorem and noting that it is a "functional instantiation" of the appropriate generic one, given the functional substitution that maps generic symbols to concrete ones. Such a functional instance is admissible provided the concrete symbols satisfy the constraints on the generic ones. It is not necessary to reconstruct the arguments necessary to prove the generic theorems in the first place.

Thus the development of the generic functions and theorems constitute a useful body of knowledge. This knowledge can be codified in the form of an ACL2 "book." A book is just a file of admissible events, starting with a designation of the current symbol package. The user could thus create a suitable book by putting the text for the `encapsulate`, `defun CYCLE` and the two `defthm`s above into a file and adding `(in-package "ACL2")` at the top. Let us name that file `"generic-cycle.lisp"`.

The command `(certify-book "generic-cycle")` will "certify" the book by checking the admissibility of every event in it. Once certified, the command `(include-book "generic-cycle")` will extend the existing chronology by events in the book, provided certain syntactic requirements are met that insure that names are not redefined.

Books are like encapsulations in that they may contain `local` events. When a book is certified its local events are processed. But when a book is included in a session, its local events are skipped. Thus, we often mark some of the lemmas in a book as `local`. They are proved and used during the certification of the book, but do not clutter the ACL2 data base when the book is included in a user session.

The provision of such structuring mechanisms, especially books and encapsulations, is one of the reasons ACL2 has been successfully applied to "industrial-strength" verification problems such as the verification of floating-point division microcode on the AMD K5 [14] and floating-point square root microcode on the AMD K5 and floating-point square root hardware on the K7 [15, 16], the verification of microcode on the Motorola CAP digital signal processor [4, 6], and the verification of certain aspects of the Rockwell-Collins JEM1 (a silicon Java Virtual Machine) [7]. Structured proofs produce smaller and simpler contexts, making it easier for the user to control the theorem prover while exploiting its automatic features. In addition, books allow users to combine and build on the work of others or their own past work. For example, if an existing book contains a thousand theorems, ten of which are needed in a new project, the user can create a new book which provides just the ten useful theorems: the new book locally includes the old one and then non-locally states the desired ten theorems. Including this new book imports the desired ten without importing the others.

This paper demonstrates a notion of correctness that encompasses both `encapsulate` and `include-book` events. However, the attention in this paper is turned more toward the treatment of `encapsulate` rather than `include-book`, because only the former events can directly introduce constrained functions. Of course, a book may include `encapsulate` events among its events.

From the brief description given so far it may appear that `encapsulate` is too simple to warrant a paper like this. But suppose an encapsulation introduces the new, constrained function symbol `FN`. Suppose the events in the encapsulation include a `local` definition of `FN` (as it must) and a non-`local` theorem about `FN`. That theorem will clearly be part of the constraint on `FN`. But what if the `encapsulate` contains some other non-`local` theorems, not mentioning `FN`? Are they part of the constraints on `FN` or can they be "moved out" of the `encapsulate` altogether? What if they mention function symbols that use `FN`? Can non-`local` function definitions that do not call `FN` be moved out? What if the termination argument for such a function symbol involves `FN`? Finally, what should be done with theorems about the `local` witness for `FN` that are deduced implicitly by ACL2, e.g., theorems that characterize its type and link it to suggested induction schemes? Should such theorems be part of the constraint? These and related issues are considered carefully in this paper. As noted, we uncovered soundness bugs in ACL2 via this careful consideration. One such bug was involved with the use of a constrained function in the termination argument of a recursive function defined non-locally within the same encapsulation. The failure to include the definition of that other function among the constraints of the encapsulate allowed the theorem prover to do unsound inductions after the encapsulation. The well-foundedness argument for those "inductions" depended on properties of the witness used for the constrained function, but those properties were not made explicit among the constraints of the encapsulation.

The following additional events are especially relevant to this paper because of their interactions with `encapsulate`.

The ACL2 user may introduce a new axiom with an event of the form `(defaxiom` *name term ...*`)`, where *term* is a term to added as a new axiom with the name *name*. We generally discourage the use of `defaxiom` because it allows the introduction of inconsistency by over constraining already-axiomatized function symbols. If a new (i.e., previously unaxiomatized) function symbol is to be constrained it is recommended that the introduction of the symbol and of all the constraints on it be done at once, via `encapsulate`.

The ACL2 user may introduce n mutually recursive function definitions with an event of the form `(mutual-recursion` $defun_1$ ... $defun_n$`)`. Mutual recursion is otherwise prohibited by the syntactic restrictions on the `defun` event: every function symbol used in a definition, other than the one being defined, must have been already introduced. Here is a simple example of mutual recursion.

```
(mutual-recursion
 (defun EVEN-NATP (N)
  (IF (ZP N)
      T
      (ODD-NATP (- N 1))))

 (defun ODD-NATP (N)
  (IF (ZP N)
      NIL
      (EVEN-NATP (- N 1)))))
```

The first recognizes the even natural numbers and the second recognizes the odd natural numbers. The measure theorems for a mutually recursive clique establish termination of all the functions in the clique. ACL2 supports explicit mutually recursive definitions even though, as above, singly recursive alternatives are often possible. Indeed, in a certain sense, all mutually recursive definitions are eliminable in favor of a singly recursive definition.

Finally, the user can introduce a conservative "choice" function satisfying a given proposition. For example,

```
(defchoose AN-EVEN-ELEMENT (E) (X)
  (AND (MEMBER E X)
       (EVEN-NATP E)))
```

introduces the function `AN-EVEN-ELEMENT` and axiomatizes it so that when applied to `X` it returns a member of the list `X` that is an even natural number, provided there is such an object. The `defchoose` event is very similar to a stylized use of `encapsulate` in which the new function is constrained to satisfy the axiom described, except that `defchoose` frees the user from exhibiting the witness that would be required by `encapsulate`.

3 Introduction to Main Result

We have seen in the preceding section that a session with ACL2 may be viewed as the extension of a built-in *ground-zero* theory by a sequence of *events*. These events may be classified into two groups. The *axiomatic events* are those that introduce new facts: definitions (especially `defun` events, possibly in the context of `mutual-recursion`), constrained axioms (via `encapsulate` events), and "pure" axioms (`defaxiom` events). The other events are theorems that are, at least allegedly, proved from those facts.

The claim below is probably the first property one would require of a theorem prover: When the prover asserts provability of a formula, then that formula really is provable from the appropriate axioms.

Informal Claim: Provability of ACL2-checked formulas. Every alleged theorem of an ACL2 session is in fact a theorem first-order derivable from the extension of the built-in logic (with induction) by the axiomatic events of that session.

This claim is perhaps stronger than it appears. The ACL2 user typically defines some function symbols of interest but must also introduce extra definitions and lemmas before leading ACL2 to a proof of a conjecture involving those function symbols of interest. The claim above implies that the theorem really is first-order derivable from the original definitions and axioms, i.e., that the extra definitions are at most heuristically useful, not logically necessary. To see this, recall that books may have `local` events, so one can presumably certify the book obtained by marking the auxiliary functions and lemmas as `local`. A new session could start with an `include-book` command, which would import into the session only the non-local events. The application of the claim to the new session allows us to conclude that the theorem is first-order provable without the auxiliary definitions.

The remainder of the paper will justify the informal claim above in two steps. Up through Section 7 we will introduce notions that allow us to state a theorem that formalizes the claim above and to prove that theorem. Then, in Section 8 we will tie that theorem to the informal claim above. We will not consider "extra-logical" correctness issues, such as the correctness of the ACL2 theorem prover's term-manipulation procedures (e.g., rewriter and decision procedures), ACL2's handling of syntax (e.g., packages and macros), and the capability provided for slipping into a "program-only" mode.

4 Logical Preliminaries

In this section we review some notions of first-order logic, a subject with which we assume some familiarity on the part of the reader. We also introduce some logical conventions and some ACL2-specific definitions important for what follows.

The logic considered in this paper is first-order logic with equality, as described for example in [18], with the restriction that there are no relation symbols other than equality. The set of function symbols occurring in a set S of formulas is called the *language of* S. As usual, a *theory* is the set T of first-order consequences of a given set S of *axioms*, that is, the set of formulas first-order derivable from S whose extra-logical symbols all occur in S. These formulas (the elements of T) are sometimes called *theorems* of S, or of T. Recall that a theory T_1 is a *conservative extension of* a theory T_0 if T_1 is a superset of T_0 and for every theorem φ of T_1 in the language of T_0, φ is a theorem of T_0.

The Completeness Theorem for first-order logic is of use on occasion in this paper: if a first-order theory T is consistent then it has a model. We also use implicitly its converse, the Soundness Theorem for first-order logic.

We only consider (consistent) theories that extend the first-order *ground-zero* theory *GZ* first discussed in Section 2. It is easy to see that *GZ* itself is consistent, even when extended as described below.

We write *PR* to denote the language of *GZ*.[2] In this paper we typically write "`IF` x `then` y `else` z" instead of the more formal "`IF`(x, y, z)".

We also assume that *GZ* contains the binary function symbol $\prec$. Intuitively, $\prec$ is a well-founded relation on the ACL2 universe. For each set of function symbols there is a corresponding set of *induction axioms*, which are the universal closures of all formulas of the following form, called the *induction axioms for* φ (with respect to the variable y).

$$\begin{array}{c}(\forall y)(((\forall x \prec y)\varphi[y := x]) \rightarrow \varphi) \\ \rightarrow \\ (\forall y)\varphi\end{array}$$

Definition. The set of *induction axioms in* a given language L is defined to be the set of all induction axioms for φ with respect to y (as defined above), where φ ranges over the set of formulas of L and y ranges over the set of free variables of φ. $\Box$

Definition. A first-order theory T is *inductively complete* if it contains *GZ*together with every induction axiom in the language of T. $\Box$

Assumptions. We use make the following additional assumptions about *GZ*. (1) The ordering $<$ on nonnegative integers is inherited from $\prec$. (2) The predicate `ZP` returns `NIL` on the positive integers and `T` on all other arguments. (3) The Peano axioms are contained in *GZ*. This last assumption permits the carrying out of standard arguments guaranteeing definability of primitive recursive functions on the natural numbers, and allows us to apply certain known theorems and techniques in Appendix B.

All of the above requirements for *GZ* are honored in the ACL2 implementation.

Convention. We consider only consistent, inductively complete first-order theories (which therefore extend *GZ*). Henceforth, the term "theory" will be understood to mean: "consistent and inductively complete theory". $\Box$

Definition. The *inductive completion of* a set S of first-order formulas *with respect to* a language L is the theory whose axioms are S together with the set of induction axioms in the language L. When L is omitted, it is understood to be the language of $S \cup GZ$. $\Box$

Definition. Let T be a theory. A *definitional axiom over* T is a (finite) conjunction D of equations each of the form

$$f(x_1, \ldots, x_n) \quad = \quad term$$

[2]The notations "*PR*" and "*GZ*" are intended to suggest "Primitive Recursive" and "Ground-Zero", respectively.

where: *term* is a term in the union of the language of T with the set of left-hand side function symbols of D; x_1, …, x_n are distinct variables that include all variables occurring in *term*; and each left-hand-side function symbol is distinct from the others and from all function symbols in the language of T. Both D and the conjunct above are said to *define* f, with list of *formal parameters* $(x_1, \ldots, x_n)$. We may also say in this case that $f(x_1, \ldots, x_n)$ is *defined by* D.

When the theory T is understood or is not important for the discussion, we may talk simply about a *definitional axiom.* $\Box$

Notation. We sometimes write $f(\vec{x})$ as an abbreviation for a term of the form $f(x_1, \ldots, x_n)$. We also feel free to extend this abbreviation in obvious ways, for example writing $(\vec{x}, y)$ to abbreviate $(x_1, \ldots, x_n, y)$.

Convention. We will use the standard notion of *first-order definable function with respect to* a theory T. This notion applies to a formula φ together with a list $(\vec{x}, y)$ enumerating the free variables of φ, with the following property: it is a theorem of T that for all $\vec{x}$ there is a unique y such that φ holds. In this case, the *arity* of the implicitly defined function is n. When the theory T is implicit or not important for the discussion, we may omit it and just talk about a first-order definable function. $\Box$

Finally, we introduce a basic logical notion underlying the `defchoose` event.

Definition. For any first-order formula φ with free variables contained in the sequence $v, x_1, .., x_k$ of distinct variables, and any function symbol f of arity k, the *Skolem axiom introducing* f for φ with respect to v is defined to be the following formula.

$$\varphi \quad \rightarrow \quad \textbf{let } v = f(x_1, \ldots, x_k) \textbf{ in } \varphi$$

We also allow more than one bound variable: if φ has free variables among the distinct variables v_1, …, v_n, x_1, …, x_k $(n > 1)$ and f is a function symbol of arity k, the *Skolem axiom introducing* f for φ *with respect to* $\langle v_1, \ldots, v_n \rangle$ is defined to be the following formula.

$$\varphi \rightarrow \textbf{let } (\texttt{LIST } v_1 \ldots v_n) = f(x_1, \ldots, x_k) \textbf{ in } \varphi$$

In any context where we use this definition, it will be provable that f returns a list of length n. $\Box$

5 Recursion and Induction

In this section we introduce two notions of *admissibility* for recursive definitions. One notion, *measure admissibility*, is similar to the *definitional principle* for Nqthm explained in [1, 5], and is used by the ACL2 implementation. However, for our treatment of histories and chronologies below we need a different notion, which we call *interpreter admissibility.* The following properties are demonstrated below.

- If a definitional axiom is measure admissible, then it is interpreter admissible.
- If a definitional axiom is interpreter admissible, then it yields a conservative extension.
- If a definitional axiom is interpreter admissible, then the associated induction rule of inference is a derived (sound) rule.

5.1 Measure Admissibility

Section III.I of [1] argues that if a recursive definition satisfies a certain definitional principle, then this definitional axiom defines a unique function. Here we present a self-contained explication of essentially the same principle, which like the Nqthm version is based a notion of *measure* that can be viewed informally as guaranteeing that the function (or set of functions) being defined terminates on all inputs. The idea is to imagine that each recursive call is *ruled* by certain conditions that guarantee that a given *measure* of the arguments is smaller for that recursive call than for the top-level call.

We first illustrate with an example. Consider the following definition of a pair of functions defining the notions of odd and even natural numbers. Here we assume that `ZIP` ("zero-integer-property") is defined so that it is true of 0 and of all non-integers. Recall that `T` conventionally represents "true" and `NIL` conventionally represents "false". Following a usual convention of ACL2 and Nqthm, we write these functions so that they treat non-integers as 0.

$$\begin{aligned}
\texttt{ODDP}(x) &= \texttt{IF ZIP}(x)\ \texttt{then NIL else} \\
&\quad\ \texttt{IF}\ x < 0\ \texttt{then EVENP}(x+1)\ \texttt{else EVENP}(x-1) \\
\texttt{EVENP}(x) &= \texttt{IF ZIP}(x)\ \texttt{then T else} \\
&\quad\ \texttt{IF}\ x < 0\ \texttt{then ODDP}(x+1)\ \texttt{else ODDP}(x-1)
\end{aligned}$$

It's clear that execution of functions satisfying these definitions will terminate, because the absolute value of x decreases in each recursive call. The absolute value function is what we call a *measure* for the definition above; clearly it is first-order definable if '$<$' and '$-$' are. In general, each function defined in a given definitional axiom may have its own measure function.

The example above may be viewed as generating the following proof obligations for the definition of `ODDP`, when we use $|x|$ as the measure function for both `ODDP` and `EVENP`. The same proof obligations are generated for the definition of `EVENP`. Note that there is one proof obligation for each recursive call, and in each case the "tests" (negated when appropriate) from the if-then-else tree are antecedents for the implication.

$$\begin{aligned}
(\neg\texttt{ZIP}(x) \wedge x < 0) &\rightarrow |x+1| < |x| \\
(\neg\texttt{ZIP}(x) \wedge \neg(x < 0)) &\rightarrow |x-1| < |x|
\end{aligned}$$

We now make precise the notion of *measure.*

Definition. Let D be a definitional axiom over a theory T. A *measure for* D (*over* T) associates with each conjunct

$$f(\vec{x}) \quad = \quad term$$

of D a first-order definable function $m_f(\vec{x})$ with respect to T. $\Box$

Roughly speaking, the measure should decrease for the arguments of each recursive call, subject to the conditions *ruling* that call; see the example above involving `EVENP` and `ODDP`. We now define recursively a notion of *rules.* A more generous notion is found in [10], which for example would say that both `P` and ¬`Q` *govern* the call of `f` in the following term:

```
IF P then g(IF Q then 0 else f(x)) else 1
```

But here, only `P` *rules* the calls of `f`. That is: we only consider the top-level if-then-else structure of the term. This decision is in accord with the ACL2 implementation, which makes this choice for heuristic reasons involving induction.

Definition. We say that a term t *rules* an occurrence of a term s in a term b iff b is of the form `IF` *test* `then` *tbr* `else` *fbr* and one of the following conditions holds:
(a) the occurrence is in *tbr* and either t is *test* or t rules the occurrence of s in *tbr*; or
(b) the occurrence is in *fbr* and either t is ¬*test* or t rules the occurrence of s in *fbr*. $\Box$

Definition. Let D be a definitional axiom with measure m over a theory T. The *measure theorem for* D *via* m, denoted $m(D)$, is the conjunction of the implications obtained as follows from each occurrence of a term $f(u_1, \ldots, u_n)$ in a right-hand side of D, where f is defined by D with formal parameters $x_1, \ldots, x_n$, and the occurrence is ruled by terms $t_1, \ldots, t_k$.

$$t_1 \wedge \ldots \wedge t_k \rightarrow m_f(u_1, \ldots, u_n) \prec m_f(x_1, \ldots, x_n)$$

$\Box$

Definition. Let T be a theory. A definitional axiom D over T is *measure admissible over* T if for some measure m for D over T, the corresponding measure theorem $m(D)$ is a theorem of T. In this case we also say that D is *measure admissible via* m *over* T. $\Box$

5.2 Introduction to Interpreter Admissibility

The handling of recursive definitions in our theory can present an opportunity for confusion or error if it is not done carefully. If we only require the Nqthm notion of admissibility presented just above, then when we include a book with a recursively defined function whose measure is marked `local`, it is not clear that the definitional axiom is still measure admissible, because the measure may have been defined using some of those `local` functions.

Below we develop an alternate notion of admissibility, *interpreter admissibility*. We will show that this notion follows from measure admissibility, which is important for verifying (in Section 8) that the implementation is well-behaved. The main idea of interpreter admissibility is to consider, for a given definitional axiom, a *canonical interpreter* that is definable using only function symbols that are used in the definitional axiom or are built-in (members of PR). This interpreter is first-order definable in the current history because its recursion is particularly simple: an extra "stack depth" argument decreases on each recursive call. We represent "divergence" – insufficient stack depth for termination – by a return value of `NIL`; otherwise, the return value is a pair that is intended to equal the result of `CONS`ing the "real" value with `NIL`.[3] Interpreter admissibility is defined to mean that the canonical interpreter is *total*: for any given argument list there is a sufficient stack depth for non-divergence (termination).

Conventions. For the rest of this section, fix a definitional axiom D over a theory T (which, as always, is assumed to be inductively complete), and let F be the set of function symbols defined by D. We also assume that a one-one map associates each $f \in F$ with a function symbol f' not in the language of $T \cup \{D\}$, whose arity exceeds the arity of f by 1. $\Box$

5.3 Canonical Interpreters

We turn now to the notion of *canonical interpreter* that was promised above. A key idea is embodied in the next definition. We show how to map a term u to a term u_d, which is intended to represent $\texttt{CONS}(u, \texttt{NIL})$ if the "stack depth" d is sufficiently large for computing the value of u using definitional axiom D, otherwise `NIL`. For technical reasons, we want to categorize occurrences of terms according to whether or not they are at the top-level of the if-then-else structure of a right-hand-side of D. The subscript b below is a Boolean flag used in order to make this distinction; it is set to 1 initially (for the right-hand side of a conjunct of D), but becomes 0 when we leave the top-level if-then-else structure.

Definition. For terms t and d, and for value b of 0 or 1, we define the term $t_{d,b}$ by recursion on t as follows.

If t is a variable or a constant: $t_{d,b} = \texttt{CONS}(t, \texttt{NIL})$

$(\texttt{IF}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2)_{d,1} =$
  if $(t_0)_{d,0} = \texttt{NIL}$
  then `NIL`

[3]A similar idea is used in the Nqthm "value-and-cost" function `V&C$` (see [5]), except that here we do not care about the "cost".

```
        else if $\mathtt{CAR}((t_0)_{d,0}) \neq \mathtt{NIL}$
        then $(t_1)_{d,1}$
        else $(t_2)_{d,1}$
```

If $f \notin F$, where $b = 0$ if f is `IF`:

```
$(f(t_1, \ldots, t_n))_{d,b} =$
    if $(t_1)_{d,0} = \mathtt{NIL}$ or $\ldots$ or $(t_n)_{d,0} = \mathtt{NIL}$
    then NIL
    else $\mathtt{CONS}(f(\mathtt{CAR}((t_1)_{d,0}), \ldots, \mathtt{CAR}((t_n)_{d,0})), \mathtt{NIL})$
```

If $f \in F$:

```
$(f(t_1, \ldots, t_n))_{d,b} =$
    if $(t_1)_{d,0} = \mathtt{NIL}$ or $\ldots$ or $(t_n)_{d,0} = \mathtt{NIL}$
    then NIL
    else$f'(d, \mathtt{CAR}((t_1)_{d,0}), \ldots, \mathtt{CAR}((t_n)_{d,0}))$
```

□

Recall from Section 4 that the function symbol `ZP` is defined in the ground-zero theory *GZ* to return `NIL` for positive integer arguments and `T` otherwise.

Definition (Canonical interpreter). The canonical interpreter for D is the definitional axiom obtained by replacing each conjunct $f(\vec{x}) = u$ of D by:

$$f'(d, \vec{x}) = \mathtt{IF}\ \mathtt{ZP}(d)\ \mathtt{then}\ \mathtt{NIL}\ \mathtt{else}\ u_{d-1,1}$$

We say that the canonical interpreter for D is *total in* a theory T if for each $f(\vec{x})$ defined by D, the following is a theorem of T:

$$(\forall \vec{x})(\exists d)(f'(d, \vec{x}) \neq \mathtt{NIL})\ \square$$

Convention (Definability of f'). Standard arguments show that for a given definition, there are first-order definable functions $\{f'' : f \in F\}$ for which the equations of its canonical interpreter are in fact theorems of T. Henceforth, when we mention f' we will be referring to this first-order definition over T (what we have just called f''), rather than to a new function symbol. □

Definition (Interpreter admissibility). A definitional axiom D is said to be *interpreter admissible over* a theory T if the canonical interpreter for D is total in T. □

The following obvious proposition may be used implicitly.

Proposition (Preservation of Interpreter Admissibility). If D is interpreter admissible over T, and T' extends T, then D is interpreter admissible over T'. □

5.4 Some Lemmas

In this subsection we develop machinery to allow us to prove the main results on interpreter admissibility.

Lemma (Interpreter Monotonicity). Let D be a definitional axiom over a theory T, and let u be a term in the language of $T \cup \{D\}$. Let d and d' be variables not occurring in u. Then the following is a theorem of T:

$$d \leq d' \wedge u_{d,b} \neq \texttt{NIL} \quad \rightarrow \quad u_{d,b} = u_{d',b} \wedge u_{d',b} = \texttt{CONS}(\texttt{CAR}(u_{d,b}), \texttt{NIL}) \tag{1}$$

Proof. We start with three reductions. First of all, it suffices to prove (1) without the first conjunct in the consequent of the implication, because the second conjunct gives us a value for $u_{d',b}$ that is independent of the choice of $d' \geq d$. Second, it suffices to prove the theorem only for terms u that are right-hand sides of equations of D. For once that is done, then the theorem follows by an easy induction using the definition of $t_{d,b}$. Third and finally, we assume, by renaming variables in D if necessary, that variables d and d' do not occur in D.

Let $A_{d,d'}$ be the universal closure of the (finite) conjunction of all formulas of the following form, as u ranges over subterms of D and b is 0 or 1:

$$d \leq d' \wedge u_{d,b} \neq \texttt{NIL} \quad \rightarrow \quad u_{d',b} = \texttt{CONS}(\texttt{CAR}(u_{d,b}), \texttt{NIL})$$

Let $\vec{y}$ be a sequence of all the variables occurring in D (and hence not d or d'). It suffices to prove the following claim by strong induction on d within T:

$$T \vdash (\forall \vec{y})(\forall d') A_{d,d'}$$

Hence, it suffices to show that the following holds for each subterm u of D and each value of b in the set $\{0, 1\}$.

$$\begin{aligned} T \quad \vdash \quad & ((\forall e < d)(\forall \vec{y})(\forall e') A_{e,e'}) \rightarrow \\ & (d \leq d' \wedge u_{d,b} \neq \texttt{NIL} \rightarrow u_{d',b} = \texttt{CONS}(\texttt{CAR}(u_{d,b}), \texttt{NIL})) \end{aligned} \tag{2}$$

We show that (2) holds by induction on subterms u of D. So, working inside T, assume

$$(\forall e < d)(\forall \vec{y})(\forall e') A_{e,e'} \tag{3}$$

as well as $d \leq d'$ and $u_{d,b} \neq \texttt{NIL}$, to prove the following.

$$u_{d',b} = \texttt{CONS}(\texttt{CAR}(u_{d,b}), \texttt{NIL}) \tag{4}$$

The rest of the proof depends on the choice of u. If u is a variable or a constant then $u_{d,b}$ and $u_{d',b}$ are both (syntactically) the term $\texttt{CONS}(u, \texttt{NIL})$, so (4) is clear. Otherwise u is of the form $f(t_1, \ldots, t_n)$. The case where f is $\texttt{IF}$ and b is 1 is similar to an argument below, and is left to the reader. Otherwise, since

$u_{d,b} \neq \texttt{NIL}$, $(t_i)_{d,0} \neq \texttt{NIL}$ when $1 \leq i \leq n$. Since (by the inductive hypothesis on terms) property (2) holds for all subterms of u, then for $1 \leq i \leq n$ we have

$$(t_i)_{d',0} \quad = \quad \texttt{CONS}(\texttt{CAR}((t_i)_{d,0}), \texttt{NIL}). \tag{5}$$

There are now two cases. If $f \notin F$, then by definition $u_{d',b}$ is

$$\texttt{CONS}(f(\texttt{CAR}((t_1)_{d',0}), \ldots, \texttt{CAR}((t_n)_{d',0})), \texttt{NIL}),$$

which by (5) and axiom **`CAR-CONS`** (see Subsection 2.1) is equal to

$$\texttt{CONS}(f(\texttt{CAR}((t_1)_{d,0}), \ldots, \texttt{CAR}((t_n)_{d,0})), \texttt{NIL}),$$

which equals $u_{d,b}$ by definition. Otherwise $u_{d',b}$ is

$$f'(d', \texttt{CAR}((t_1)_{d',0}), \ldots, \texttt{CAR}((t_n)_{d',0})),$$

so by (5) and axiom **`CAR-CONS`**, it suffices, for arbitrary $\vec{x}$ such that

$$f'(d, \vec{x}) \quad \neq \quad \texttt{NIL} \tag{6}$$

to prove the following.

$$f'(d', \vec{x}) = \texttt{CONS}(\texttt{CAR}(f'(d, \vec{x})), \texttt{NIL}). \tag{7}$$

Let u be the body of the definition of f, i.e., so that "$f(\vec{x}) = u$" is a conjunct of D. By definition of the canonical interpreter for D, we know that $d > 0$ (and hence $d' > 0$ as well) and

$$f'(d, \vec{x}) \quad = \quad u_{d-1,1} \tag{8}$$
$$f'(d', \vec{x}) \quad = \quad u_{d'-1,1} \tag{9}$$

By (3) we have $A_{d-1,d'-1}$ and hence since $u_{d-1,1} = f'(d, \vec{x}) \neq \texttt{NIL}$ (by (8) and (6)):

$$\begin{aligned} u_{d'-1,1} &= \texttt{CONS}(\texttt{CAR}(u_{d-1,1}), \texttt{NIL}) \\ &= \texttt{CONS}(\texttt{CAR}(f'(d, \vec{x})), \texttt{NIL}). \end{aligned}$$

This, together with (9), implies (7). $\square$

Definition. Fix a variable symbol d that does not occur in D. We define the theory T_D be the inductive completion of the extension of T by the universal closures of the following equations, one for each $f(\vec{x})$ defined by D.

$$f(\vec{x}) = \begin{cases} \texttt{NIL} & \textbf{if} \quad f'(d, \vec{x}) = \texttt{NIL} \text{ for all } d \\ \texttt{CAR}(f'(d, \vec{x})) & \textbf{otherwise}, \quad \text{where } d \text{ is least} \\ & \qquad \text{such that } f'(d, \vec{x}) \neq \texttt{NIL} \end{cases}$$

$\square$

Lemma (Conservativity of T_D). T_D is a conservative extension of T.

Proof. This is clear from the Convention on Definability of f', i.e., f' is definable over T for each function symbol f defined by D. $\square$

Lemma (Interpreter Eliminability). Let u be a term in the language of $T \cup \{D\}$. Then:

$$T_D \vdash u_{d,b} \neq \mathtt{NIL} \rightarrow u_{d,b} = \mathtt{CONS}(u, \mathtt{NIL}).$$

Proof. This is easily established by induction on the term u, using the definition of $u_{d,b}$. The Interpreter Monotonicity Lemma is used for the case $f(\vec{t})$ where $f \in F$. $\square$

Lemma (Divergence Infectiousness). Let u and d be terms. Then the following is a theorem of T_D: $u_{d,0} = \mathtt{NIL}$ if and only if for some subterm $f(\vec{t})$ of u where $f \in F$, $f'(d, \vec{t}) = \mathtt{NIL}$.

Proof. For given d and $f(\vec{t})$ this is easily proved by induction on the term u, using the definition of $u_{d,b}$ and the Interpreter Eliminability Lemma. $\square$

Lemma (Interpreter Correctness) Suppose that D is interpreter admissible over the theory T, i.e., the canonical interpreter for D is total in T. Then $T_D \vdash D$.

Proof. Fix a conjunct $f(\vec{x}) = u$ of D. By totality of the canonical interpreter for D, then working in T_D we may choose d such that $f'(d, \vec{x}) \neq \mathtt{NIL}$. Pick the least such d. Then we have:

$$\begin{array}{rcll} f(\vec{x}) & = & \mathtt{CAR}(f'(d, \vec{x})) & \text{(by definition of } T_D\text{)} \\ & = & \mathtt{CAR}(u_{d-1,1}) & \text{(by definition of the canonical interpreter)} \\ & = & u & \text{(by the Interpreter Eliminability Lemma)} \end{array}$$

We have shown that T_D proves that $f(\vec{x}) = u$, as desired. $\square$

Lemma (Interpreter Provability). Let D be an interpreter admissible definitional axiom over a theory T. Then T_D is a subtheory of the inductive completion T' of the extension of T by D.

Proof. It suffices to prove, even without assuming interpreter admissibility of D, that the conclusion of the Interpreter Eliminability Lemma holds for T' in place of T_D:

$$T' \vdash u_{d,b} \neq \mathtt{NIL} \rightarrow u_{d,b} = \mathtt{CONS}(u, \mathtt{NIL}).$$

The proof is an easy induction on u, as in the proof of the Interpreter Eliminability Lemma (also mostly omitted). $\square$

5.5 Key Properties of Interpreter Admissibility

Our first goal below is to show that measure admissibility of D implies that its canonical interpreter is total. The following lemma is key. Its proof shows why we introduced a flag b in the notion $u_{d,b}$.

Lemma. For every term u, the following is a theorem of T. Suppose that for every subterm of u of the form $f(\vec{t})$ where $f \in F$, if G is the conjunction of the terms ruling this subterm in u then:

$$G \rightarrow (\exists d)(f'(d, \vec{t}) \neq \texttt{NIL}).$$

Then:

$$(\exists d)(u_{d,1} \neq \texttt{NIL}).$$

Proof. By the lemma on Conservativity of T_D, it suffices to prove theoremhood in T_D rather than T. The proof is by induction on u. If u is not a call of `IF` then no subterm of u has any terms ruling it and hence the result is clear from the Divergence Infectiousness Lemma. Otherwise, suppose u has the following form: `IF` $u0$ **`then`** $u1$ **`else`** $u2$. First observe that every subterm of $u0$ of a call of a function in F has no terms ruling it in u, so by the same argument as in the non-`IF` case above, we may choose $d0$ such that $u0_{d0,0} \neq \texttt{NIL}$. It follows from the Interpreter Monotonicity Lemma that $u0_{d,0} \neq \texttt{NIL}$ for all $d \geq d0$. Henceforth restrict to $d \geq d0$. Now assume that $u0 \neq \texttt{NIL}$; the other case is similar. By the Interpreter Eliminability Lemma (note that this is where it is handy to work in T_D rather than in T), $u0_{d,0} = \texttt{CONS}(u0, \texttt{NIL})$. It follows that $u_{d,1} = u1_{d,1}$. The inductive hypothesis may now be applied to $u1$ in place of u to obtain $d1$ for which $u1_{d1,1} \neq \texttt{NIL}$, and the maximum of $d0$ and $d1$ then serves as the desired value of d, by the Interpreter Monotonicity Lemma. □

Theorem (Interpreter Admissibility). Suppose that D is a measure admissible definitional axiom over the theory T. Then D is interpreter admissible over T.

Proof. By definition of measure admissibility, there exists a measure m for D over T such that the corresponding measure theorem $m(D)$ is a theorem of T. Suppose for a contradiction that the canonical interpreter for D is not total in T. By well-foundedness of $\prec$ (more precisely, inductive completeness of T) we may choose a conjunct $f(\vec{x}) = u$ of D such that

$$\neg(\exists d)(f'(d, \vec{x}) \neq \texttt{NIL}) \tag{10}$$

and yet for all $g(\vec{y})$ defined by D (renaming $\vec{y}$ if necessary so as to be disjoint from $\vec{x}$),

$$m_g(\vec{y}) \prec m_f(\vec{x}) \rightarrow (\exists d)(g'(d, \vec{y}) \neq \texttt{NIL})$$

It now follows immediately from the measure theorem $m(D)$ and the preceding lemma that for some d, $u_{d,1} \neq \texttt{NIL}$. But the canonical interpreter yields $f'(d+1, \vec{x}) = u_{d,1}$, so $f'(d+1, \vec{x}) \neq \texttt{NIL}$, contradicting (10). □

Our development of the theory of histories in the next section requires that definitional events provide conservative extensions. The following lemma is thus crucial.

Lemma (Conservativity of Definitions). Suppose that the definitional axiom D is interpreter admissible over the theory T. Then the inductive completion of the extension of T by D is a conservative extension of T.

Proof. The indicated extension of T is a subtheory of T_D, by the Interpreter Correctness Lemma. So, we are done by the Conservativity of T_D Lemma. □

Finally, we consider the use of induction schemes by the ACL2 implementation. One of the reasons that the measure-based notion of admissibility has worked successfully in Nqthm is that it has been used to justify corresponding uses of induction. A similar justification applies for ACL2. At the root of the justification is the measure admissibility of the given definition. In order to regain that justification in the present setting, we need to show that interpreter admissibility implies the existence of an appropriate measure. Below, the resulting measure is defined in the inductive completion of the extension of T by D, rather than in T as is traditional. However, this does not present a problem for justifying inductions. What *is* important, however, is that the measure is defined using only function symbols from D and from PR, so that its definition still exists in appropriate subtheories containing D.

Definition (Canonical Measure). For D an interpreter admissible definitional axiom over an inductively complete theory T, the *canonical measure* m *for* D is defined as follows in T:

$$m_f(\vec{x}) \quad = \quad \text{least } d \text{ such that } f'(d, \vec{x}) \neq \texttt{NIL}$$

The following technical definition and proposition will be used only in Appendix A. Briefly put, it identifies definitions whose recursive calls only involve functions belonging to a given set S, and furthermore, whose top-level `IF` tests only involve functions in S.

Definition (Tight Definability). Let D be a definitional axiom, and let S be a set of function symbols containing GZ. We say that D is *tight with respect to* S if every function symbol of a term u belongs to S provided that u is a subterm of a right-hand side r of D and u belongs to either of the following sets. (1) u is a proper subterm of a call of a member of D. (2) u is a subterm of (or equal to) a term t such that t or its negation rules some subterm of r that is a call of a function introduced by D. □

Proposition (Tight Definability of Canonical Measure). Let D be an interpreter admissible definitional axiom over a theory T (inductively complete), and let S be a set of function symbols containing GZ that is disjoint from the set of function symbols introduced by D. Suppose that D is tight with respect to S. Then the canonical measure for D is first-order definable in T by a definition using only function symbols in S. □

Theorem (Canonical Measure Theorem). For D an interpreter admissible definitional axiom over a theory T with canonical measure m, the measure theorem holds for D via m in the inductive completion of the extension of T by D.

Proof. Let m be the canonical measure for D. It suffices to prove that m is a measure for D over the theory T_D, by the Interpreter Provability Lemma. Thus, fix a conjunct $f(\vec{x}) = u$ of D and suppose that $g \in F$ and $g(\vec{t})$ is a subterm of

u ruled by the set G of terms. We need to prove the following (noting that $\prec$ extends $<$, by Assumption (1) in Section 4).

$$T_D \vdash (\bigwedge G) \rightarrow m_g(\vec{t}) < m_f(\vec{x})$$

By definition of m_f, it then suffices to prove that for $d \geq 0$:

$$T_D \vdash (\bigwedge G) \wedge f'(d+1, \vec{x}) \neq \texttt{NIL} \rightarrow g'(d, \vec{t}) \neq \texttt{NIL}$$

By definition of f' (i.e., of the canonical interpreter), this reduces to:

$$T_D \vdash (\bigwedge G) \wedge u_{d,1} \neq \texttt{NIL} \rightarrow g'(d, \vec{t}) \neq \texttt{NIL} \tag{11}$$

We establish (11) for all terms u, nonnegative integers d, and subterm occurrences $g'(d, \vec{t})$ of u ruled by a set G of terms, by induction on u. If u is not a call of `IF` then $u_{d,1}$ is the same term as $u_{d,0}$, so (11) is immediate from the Divergence Infectiousness Lemma (since we are working in T_D). Otherwise write u as `IF` u_0 **`then`** u_1 **`else`** u_2. If the occurrence $g'(d, \vec{t})$ is in u_0 then the argument is analogous to the case just handled. Otherwise the occurrence is in u_1 or u_2. Assume that it is in u_1; the other case is completely analogous. Then G is the result of adding u_0 to the set G' of terms ruling the occurrence in u_1. By the inductive hypothesis,

$$T_D \vdash (\bigwedge G') \wedge (u_1)_{d,1} \neq \texttt{NIL} \rightarrow g'(d, \vec{t}) \neq \texttt{NIL} \tag{12}$$

If we can show

$$T_D \vdash u_0 \wedge u_{d,1} \neq \texttt{NIL} \rightarrow u_{d,1} = (u_1)_{d,1} \tag{13}$$

then (12) and (13) together imply (11) and we are done. By definition of $u_{d,1}$, (13) follows from:

$$T_D \vdash (u_0)_{d,0} \neq \texttt{NIL} \rightarrow \texttt{CAR}((u_0)_{d,0}) = u_0.$$

But this is immediate from the Interpreter Eliminability Lemma, since we are working in T_D. □

6 Histories

We now introduce the notion of an ACL2 *history*. This notion formalizes the axiomatic content of an ACL2 session. Histories do not record the proved theorems and **`local`** function definitions of a session – aspects of a session which we must formalize in order to reach our ultimate goal of logically characterizing the result of a user's interaction with ACL2. We extend our attention to proved theorems and other aspects of a session when we consider *chronologies* in the next section.

We begin by defining the kinds of objects that we will allow in histories. Each object corresponds to one of the primitive kinds of axiomatic acts of an ACL2 session. At this point, we merely describe the syntax of these objects; we'll consider semantic issues shortly.

Definition. A *labeled formula* is a pair consisting of a *label* and a formula, where the label has one of the following four forms.

- `<defuns,` D`,` F`>`, where F is the finite set of function symbols defined by the definitional axiom D.
- `<defchoose,` φ`,` $\vec{y}$`,` f`>`, where f is a function symbol, φ is a formula, and $\vec{y}$ is a nonempty (and finite) sequence of variables.
- `<defaxiom>`
- `<constraint,` s`>`, where s is a sequence of labeled formulas.

When the meaning is clear, we will feel free to confuse a labeled formula with its *formula*, the second component of the $\langle label, formula\rangle$ pair. We will also feel free to call the universal closure of its formula an *axiom of* or *axiom introduced by* any sequence in which it appears. We may refer to a formula φ as being *labeled by* defuns, defchoose, defaxiom, or constraint, when there is a labeled formula in the present context that has the corresponding label and has formula φ. □

Notation. We write h, A to denote the result of extending a sequence h by a new element A; we write h, A, h' to denote the result of extending a sequence h by the new element A and then by the sequence h'; and so on. It will be clear from the context which objects being concatenated are elements and which are sequences. □

Definition. The (set of) *function symbols introduced by* a labeled formula or a sequence of labeled formulas is defined by recursion as follows. A formula labeled by `<defuns,` D`,` F`>` introduces the set F of function symbols; similarly `<defchoose,...,` f`>` introduces the singleton set $\{f\}$. A formula labeled by `<defaxiom>` introduces the empty set of function symbols. A formula labeled by `<constraint,` s`>` introduces the set of function symbols that s introduces. A sequence of labeled formulas introduces the union of the sets of function symbols introduced by each labeled formula in the sequence. □

Definition. Let h be a sequence of labeled formulas. A *function symbol of* h is a function symbol that is either introduced by h or is a member of *PR*. The *language of* h is the set of all function symbols of h. The *theory of* h is the inductive closure of the union of *GZ* with the universal closures of the formulas of h and the induction axioms in the language of h. □

Convention. When our meaning is clear, we will feel free to confuse a sequence of labeled formulas with its theory. We will also feel free to talk about one sequence h_1 of labeled formulas *conservatively extending* another sequence h_0 of labeled formulas, to indicate that the theory of h_1 conservatively extends the theory of h_0. Of course, the order of elements in the sequences h_0 and h_1 is irrelevant for this notion. $\square$

Definition. The notion of *defaxiom-free* is defined by recursion, as follows. A labeled formula is *defaxiom-free* if its label is not `<defaxiom>` and moreover, if its label is `<constraint,` s`>` then s is defaxiom-free. A sequence of labeled formulas is *defaxiom-free* if each of its members is defaxiom-free. $\square$

The following is obvious (and well-known in the case of theories).

Lemma (Transitivity of Conservative Extension). If h_0, h_1, and h_2 are theories or sequences of labeled formulas such that h_2 conservatively extends h_1 and h_1 conservatively extends h_0, then h_2 conservatively extends h_0. $\square$

Next, we prepare for the introduction of the notion of *history* by starting with the syntactic requirements only, ignoring proof obligations.

Definition (Weak history). A *weak history* is a finite sequence h of labeled formulas meeting the following requirements.

(1) For each labeled formula in h whose label is not a defaxiom label, the set of function symbols introduced is non-empty.

(2) Every function symbol occurring in the formula of a labeled formula A of h must either be introduced by A or be a function symbol of the set of predecessors of A in h.

(3) The family of all sets of function symbols introduced by labeled formulas in h is pairwise disjoint. Moreover, each such set of function symbols is also disjoint from PR; thus, no introduced function symbol can occur in GZ.

(4) For each labeled formula A of h whose label has the form `<defchoose,` φ`,` $\vec{y}$`,` f`>`, the formula of A is the *Skolem axiom introducing* f for formula φ, with respect to the sequence $\vec{y}$ of variables. (See Section 4 for the definition.) Furthermore, if h is h_0, A, h_1, then φ is a formula in the language of h_0.

(5) If h is h_0, A, h_1, where A has label `<constraint,` s`>`, then: s is a defaxiom-free sequence of labeled formulas; h_0, s is a weak history; and the formula of A is in the language of h_0, s.

(6) If h is h_0, B, h_1 where B is a labeled formula with label `<defuns,` D`,` F`>`, then D is an definitional axiom over the theory of h_0. $\square$

Definition. A *history* is a weak history h that meets the following requirements in addition to those above.

(5') Extending (5): h_0, s is a history and A is a theorem of the theory h_0, s.

(6') Extending (6): D is an interpreter admissible definition of F over the theory of h_0. $\square$

Remark. We will do a number of proofs by induction on sequences of labeled formulas. Invariably these recursions are justified by considering first the number of constraint labels and then the length of the sequence. We leave this justification implicit in arguments below. $\square$

Proposition. If h is a weak history, then the language of h includes the language of the theory of h. $\square$

It is possible for the language of h to be a proper superset of the language of the theory of h. Consider for example what happens if we add a constrained function symbol with no axioms.

Implementation note. When an `encapsulate` event in ACL2 does not introduce any local functions, it is viewed simply as the sequence of non-local events contained within. This allows us to satisfy Property (1) above. That property could probably be safely omitted, but at any rate, ACL2 does not introduce "constraints" (non-definitional axioms) when there are no local functions in an `encapsulate`. This choice can reduce the proof obligations arising from functional instantiation, and can allow induction schemes that would otherwise not be used. We will say more about such details of the implementation of `encapsulate` in Appendix A.

We will use the following obvious lemma implicitly.

Lemma. (i) Every initial segment of a weak history is a weak history.

(ii) Every initial segment of a history is a history. $\Box$

Lemma (History Monotonicity). Suppose that h_0, h_1, h_2 is a history, and that h_0, h_1' is a history whose theory contains that of h_0, h_1, such that h_1 and h_1' introduce the same function symbols. Then h_0, h_1', h_2 is a history.

Proof: An easy induction on h_2. The key idea is that the proof obligations introduced during the processing of h_2 are still provable if we replace h_1 by the larger history h_1'. In particular, the Proposition on Preservation of Interpreter Admissibility (Subsection 5.3) justifies property (6') of a history. $\Box$

Next, we introduce an operation on sequences of labeled formulas that is motivated by the idea of replacing each constrained formula by the sequence of events justifying it.

Definition. The *expansion* of a sequence of labeled formulas is defined by recursion, as follows. The expansion of the empty sequence is the empty sequence. The expansion of h, A where A is labeled by `<constraint,`*s*`>` is the expansion of h, s. Finally, the expansion of h, A for any other labeled formula A is h', A where h' is the expansion of h. $\Box$

Remark (for readers familiar with ACL2). This notion of "expansion" formalizes the ACL2 `:puff` command in the case that there are no `local` events.

Lemma (Expansion). Let h_1 be the expansion of the sequence h_0 of labeled formulas. Then h_1 is a sequence of labeled formulas having the following properties.

(a) h_1 and h_0 introduce the same function symbols.

(b) The theory of h_0 is a subset of the theory of h_1, and hence for all sequences h and h' of labeled formulas, the theory of h, h_0, h' is a subset of the theory of h, h_1, h'.

(c) For every history h, if h, h_0 is a history, then h, h_1 is a history.

(d) If h_0 is defaxiom-free then h_1 is defaxiom-free.

(e) h_1 contains no constraint labels.

Proof: An easy induction. The History Monotonicity Lemma, (a), and (b) are used for the proof of (c). $\Box$

Our next goal is to show that histories provide conservative extensions. In fact, we prove a slightly stronger result that is useful later. The following lemma will be used in its proof.

Lemma (Conservativity of Defchoose). Suppose that h is a sequence of labeled formulas. Suppose also that A is a formula with label `<defchoose,` φ`,` $\vec{y}$`,` f`>`, where f is not in the language of h and φ is a formula in the language of h. Then h, A conservatively extends h.

Proof: Immediate from the conservativity result in Appendix B. □

We are now ready to prove a slight strengthening of a special case of the theorem concluding this section. Here, we only consider extensions obtained by adding events to the end of a given history.

Lemma (History Conservativity). If h_0, h_1 is a history and h_1 is defaxiom-free, then h_0, h_1 conservatively extends h_0. More generally, suppose also that h'_0 is a sequence of labeled formulas that includes h_0 as a subsequence, where no function symbol introduced by h_1 occurs in the theory of h'_0. Then h'_0, h_1 conservatively extends h'_0.

Proof. By parts (a), (c), and (d) of the Expansion Lemma, the hypotheses continue to hold if we replace h_1 by its expansion, which we temporarily call H_1. By part (b) of the Expansion Lemma, the theory of h'_0, H_1 includes the theory of h'_0, h_1. Hence it suffices to prove the theorem for H_1 in place of h_1, which by parts (d) and (e) of the Expansion Lemma consists entirely of formulas labeled by defun and defchoose. Without loss of generality, then, we assume that every formula of h_1 is labeled by defun or defchoose.

We proceed by induction on the length of h_1. The case that h_1 is empty is clear. Otherwise let us write h_1 as A, h'_1. We can apply the inductive hypothesis using: h_0, A for h_0; h'_0, A for h'_0; and h'_1 for h_1. Then, we may conclude that h'_0, A, h'_1 (which is h'_0, h_1) conservatively extends h'_0, A. Thus it remains only to show that h'_0, A conservatively extends h'_0, since then we are done by the transitivity of conservative extension. But since we have assumed that every formula of h_1 is labeled by defun or defchoose, this is immediate from the Conservativity of Definitions Lemma (Subsection 5.5) and the Conservativity of Defchoose Lemma (above). For the defun case we are using the hypothesis that h'_0 extends h_0, together with the fact that the proof obligation for interpreter admissibility of A is provable in h_0 (because h_0, h_1 is a history). □

Corollary (History Consistency). Every defaxiom-free history is consistent.

Proof. This is immediate from the History Conservativity Lemma, where h_0 is the empty history and h_1 is the given defaxiom-free history, since GZ has been assumed to be consistent (see Section 4). □

Definition. Fix a weak history h. We define the *set of ancestors of* a function symbol of h or a labeled formula of h as follows. Every function symbol in PR has the empty set of ancestors. Now fix a history and let A be a labeled formula in that history that introduces a function symbol f (perhaps among others). The *set of ancestors of f* (with respect to this history) is defined to be the set of ancestors of A, which is defined to be the union of the set of function symbols introduced by A with the set of ancestors of all function symbols occurring in the formula of A that are not introduced by A.

The *set of proper ancestors* of a labeled formula, or of a function symbol introduced by the labeled formula, is defined to be the result of removing function symbols introduced by the labeled formula from the set of ancestors. □

Note that the recursion above is justifiable by part (2) of the definition of weak history.

Definition. Suppose h is a sequence of labeled formulas, and suppose h' is a subsequence of h. We say that h' is *closed under ancestors (with respect to h)* provided the following two conditions hold.

(1) Every ancestor in h of every function symbol occurring in a formula of h' is a function symbol of h'.

(2) Every element of h labeled by defaxiom is a member of h'. □

The following proposition follows easily from the definition above and the definition of ancestors.

Proposition. If h' is closed under ancestors with respect to a weak history h, then every function symbol occurring in a formula of h' is a function symbol of h', i.e., is in *PR* or is introduced by h'. □

The following simple lemma is a key ingredient of the argument required for the History Conservativity Theorem that follows.

Lemma (Restriction). Suppose that h_0, s, h_1 is a sequence of labeled formulas such that h_0, s is a history, where s is defaxiom-free, and suppose that the function symbols introduced by s do not occur in the formulas of h_1. Then h_0, s, h_1 conservatively extends h_0, h_1.

Proof. Since h_0, s is a history and s is defaxiom-free, then by the History Conservativity Lemma, h_0', s conservatively extends h_0' for every sequence h_0' of labeled formulas that extends h_0 such that no function symbol introduced by s occurs in the theory of h_0'. In particular, h_0, h_1, s conservatively extends h_0, h_1, and this is just another way of stating the conclusion above. □

Theorem (History Conservativity). Suppose h is a history and h' is a subsequence of h that is closed under ancestors. Then h is a conservative extension of h'.

Proof: by induction on the number of labeled formulas in h that are not in h'. If that number is 0, then we are done. Otherwise, let A be the first labeled formula in h that is not in h'. Let h'' be the result of inserting A into h' so that the result is a subsequence of h; thus we may write h'' as h_0, A, h_1 where h' is h_0, h_1, and h_0, A is an initial subsequence of h. Clearly h'' is closed under ancestors, since h' is. By the inductive hypothesis, h is a conservative extension of h''. Since the relation of conservative extension is transitive, it suffices to show that h'' is conservative over h', i.e., that h_0, A, h_1 is conservative over h_0, h_1. Since h' is closed under ancestors, A is not labeled by a defaxiom label. Hence we are done by the Restriction Lemma, provided we can show that the function symbols introduced by A do not occur in the formulas of h_1. But this is clear from the Proposition above, since h_0, h_1 is closed under ancestors. □

7 Chronologies and Formal Results

As suggested in Section 1, we wish to formulate a well-behaved notion of *chronology* that formalizes the result of a user's interaction with ACL2. Chronologies will be defined to consist of axiomatic acts – which is the part forming a *history* as defined in the preceding section – and formulas allegedly first-order derivable from those axioms. Thus, we would like a chronology to be a sequence of labeled and unlabeled formulas such that its restriction to labeled formulas is a history, and such that every unlabeled formula is a theorem of the labeled formulas. However, users interact with ACL2 in more complex ways than simply introducing axioms (including definitions) and proving theorems: they may also use the structuring mechanisms provided by `local`, `encapsulate`, and `include-book`. In this section we formalize user interaction with ACL2 and key properties of it. The next (final) section connects this formalization with the implementation.

Definitions. Let s be a finite sequence of labeled and unlabeled formulas.

(i) $H(s)$ is the subsequence of s consisting of labeled formulas. Intuitively, we think of $H(s)$ as the history part of s, i.e., the axiomatic acts in s.

(ii) $\mathit{THM}(s)$ is the set of universal closures of unlabeled formulas of s. Intuitively, we think of $\mathit{THM}(s)$ as the proved theorems of s.

(iii) s is a *weak chronology* if $H(s)$ is a weak history and for every initial subsequence of s of the form s_0, A where A is an unlabeled formula, every function symbol occurring in A is a function symbol of $H(s_0)$. $\Box$

Definition. The class of *chronologies* is the least class of sequences that contains the empty sequence and is closed under the four operations given below.

- [**Labeled extension**]
  If s is a chronology and A is a labeled formula such that $H(s), A$ is a history, then s, A is a chronology.

- [**Unlabeled extension**]
  If s is a chronology and φ is a formula in the language of $H(s)$ that is provable from the union of the theory of $H(s)$ with $\mathit{THM}(s)$, then s,φ is a chronology.

- [**Delete**]
  If s is a chronology and s' is a weak chronology, where s' is a subsequence of s that contains all labeled formulas of s with label `<defaxiom>`, then s' is a chronology.

- [**Include**]
  Suppose that s_0 and s_1 are chronologies, that s_2 is the subsequence of s_1 obtained by deleting members of s_1 that belong to s_0, and that s_0, s_2 is a weak chronology. Then s_0, s_2 is a chronology.

□

Proposition. Every history is a chronology.

Proof: By an easy induction on length, using the [Labeled extension] rule. □

Implementation note. ACL2 lays down certain "command markers" that indicate the initial segments of a given chronology that it will accept as chronologies. However, this restriction is not necessary for correctness, so we do not model it here.

The following three lemmas are all that remain before we are ready to prove the main results about chronologies.

Lemma (History Sufficiency). Suppose that h is a weak history. Assume that for every initial segment h', A of h, there exists h'' such that h'', A is a history and the set of elements of h'' is a subset of the set of elements of h'. Then h is a history.

Proof: by induction on h. If h is empty then this is trivial. Otherwise, write h as h', A. By the inductive hypothesis, h' is a history. And, we are given that h is a weak history. Using the assumption of the theorem we may choose h'' such that h'', A is a history, where the set of elements of h'' is a subset of the set of elements of h'. Now clauses (5') and (6') from the definition of history follow easily, using the inductive hypothesis in order to guarantee satisfaction of the requirement of (5') that h', s be a history. □

Lemma (Combining). Suppose that h_0 and h_1 are histories, that h_2 is the subsequence of h_1 obtained by removing all elements of h_0 from h_1, and that h_0, h_2 is a weak history. Then h_0, h_2 is a history.

Proof. This is an an immediate consequence of the History Sufficiency Lemma. □

Lemma (Ancestors Preserve History). Suppose h is a history and h' is a subsequence of h that is a weak history and contains all elements of h labeled by defaxiom. Then h' is a history.

Proof. An easy argument, omitted here, uses (2) from the definition of weak history to show that h' is closed under ancestors with respect to h. Then the theorem follows by an easy induction on h', using the History Conservativity Theorem to guarantee that the proof obligations are met. Notice that it is critical here that we are using interpreter admissibility rather than measure admissibility, in order to guarantee that the proof obligations for definitional axioms are expressed in the sub-history. □

Main Lemma for Chronologies. Let s be a chronology. Then (1) $H(s)$ is a history, (2) s is a weak chronology, and (3) $THM(s)$ is a subset of the theory of $H(s)$.

Proof: by induction on the construction of the class of chronologies. The Ancestors Preserve History Lemma guarantees that the [Delete] rule preserves (1), while the History Conservativity Theorem guarantees that it preserves (3). The only other step requiring a bit of thought is the justification that $H(s_0, s_2)$ is a history in the application of the [Include] rule. But it is a weak history because s_0, s_2 is assumed to be a weak chronology, so this follows from the Combining Lemma. □

The following theorem is a consequence of the preceding lemma together with the History Conservativity Theorem of the preceding section. It captures what we really want to claim about chronologies.

Theorem (Correctness of Chronologies). Let s be a chronology. Then every theorem of s is a first-order consequence of the history of s. Furthermore, suppose that φ is proved in s, and suppose that h is any history contained in s that contains every labeled formula of s that is labeled by defaxiom or mentions a function symbol occurring in h. Then φ is provable in h. □

Corollary. Let s be a chronology such that $H(s)$ is defaxiom-free. Then the set of (universal closures of) formulas (labeled and unlabeled) of s is consistent.

Proof. By the Correctness of Chronologies Theorem, it suffices to show that $H(s)$ is consistent. This follows from the fact that $H(s)$ is a history (part (1) of the Main Lemma for Chronologies) together with the History Consistency Corollary. □

8 Implementation Correctness

In this section we give a high-level argument that our notion of "chronology" adequately models the informal notion of "ACL2 session," i.e., sequence of ACL2 events. Our goal is certainly *not* to prove correctness of routines in the ACL2 theorem prover's code; they are assumed to implement the logic, at the level of this paper. Rather, our goal in this section is to show that each such weak chronology is in fact a chronology. For then, the Correctness of Chronologies Theorem tells us that every alleged theorem is in fact a first-order consequence of the relevant axioms of the session.

To that end, we consider each event type and show that it *preserves chronologies*, that is, successful execution of such an event in a given chronology leads to a new chronology. Many ACL2 events have nothing to do with extending the logic, while many of the rest are essentially abbreviations for others; but we omit consideration of both such types of events from this paper. We turn our attention now to those that are left.

`Defaxiom` events preserve chronologies because of the [Labeled extension] rule, noting that if we are extending a chronology s, then $H(s)$ is a history by the Main Lemma for Chronologies.

The argument above for `defaxiom` also provides justification for definition events, including `defun` and `mutual-recursion` as well as `defchoose`. The [Labeled extension] rule justifies the claim that these events preserve chronologies, with the following four caveats. First, we also need the Correctness of Chronologies Theorem, which guarantees that it's sufficient to prove the necessary proof obligations using $\mathit{THM}(s)$ (as does the ACL2 system) in addition to $H(s)$. Second, we need the Interpreter Admissibility Theorem of Subsection 5.5, in order to conclude that measure admissibility (which is checked by the implementation) implies interpreter admissibility (as required by the definitions of history and chronology). Third, we need the Correctness of Functional Instantiation Theorem (in Appendix A), which allows us to use functional instantiation in the proof (as does the ACL2 system). And finally, we need the Canonical Measure Theorem of Subsection 5.5, together with the argument there, to justify ACL2's use of induction schemes in proofs.

That **`defthm`** events preserve chronologies follows similarly from the [Unlabeled extension] rule, again using the Correctness of Chronologies Theorem, the Correctness of Functional Instantiation Theorem, and the Canonical Measure Theorem.

Since ACL2 does not permit local **`defaxiom`** events, **`include-book`** preserve chronologies by the [Delete] rule (to delete local events) followed by the [Include] rule.

It remains to see that **`encapsulate`** events preserve chronologies. Suppose that s_1 is a chronology corresponding to an ACL2 session from which an **`encapsulate`** event is executed successfully, where s_2 is the sequence of events (including **`local`** events) introduced inside the **`encapsulate`** form. Thus we may assume (inductively) that s_1, s_2 is a chronology. Let φ be the conjunction of the events to be exported, i.e., those that are not marked **`local`**. Then $H(s_1, s_2) \vdash \varphi$, by the Correctness of Chronologies Theorem. By the Main Lemma for Chronologies we know that $H(s_1, s_2)$ is a history, i.e., $H(s_1), H(s_2)$ is a history. Let A be the labeled formula with label **`<constraint,`** s_2**`>`** and formula φ. Thus $H(s_1), A$ is a history, and hence s_1, A is a chronology by the [Labeled extension] rule. The implementation checks the requirement for a history that s_2 be defaxiom-free. In fact, ACL2 Version 2.0 allowed **`encapsulate`** forms with embedded **`defaxiom`** forms, and this "feature" led to a soundness bug that has been fixed. The bug was uncovered in the course of writing this paper.

APPENDICES

A Correctness of Functional Instantiation

In this appendix we prove the correctness of functional instantiation, in three stages. After introducing basic notions we prove a simplified version. This version is then generalized to a larger class of functional substitutions, and slightly strengthened in an obvious way to better match the ACL2 implementation. That second version is then strengthened to a version that accounts for an optimization made by the ACL2 implementation in its handling of `encapsulate` events.

The Nqthm version of functional instantiation is proved correct in [2].

A.1 Functional Substitutions

Definition. Let h be a history. A pseudo-function of h is either a function symbol of h, or an expression of the form `(LAMBDA` *vars term*`)` where *vars* is a list of distinct variables and every function symbol occurring in *term* is a function symbol of h. In the latter case we define the *arity* of the pseudo-function to be the length of *vars*, and variables occurring in *term* that do not belong to *vars* are called *free variables* of the pseudo-function. □

Definition. A *functional substitution* is a finite, arity-preserving function *fs* from function symbols to pseudo-functions. If in addition the range of *fs* consists entirely of function symbols, we call it a *simple* functional substitution. Otherwise, the set of *free variables of fs* is the union of the sets of free variables of the pseudo-functions in the range of *fs*.

If h is a (possibly weak) history and *fs* is a functional substitution that maps function symbols of h to pseudo-functions of h, then we may say that *fs* is a functional substitution *with respect to h*. □

Recall that *PR* is the set of function symbols occurring in *GZ*. The following proposition follows immediately from the definition above together with part (3) of the definition of weak history (Section 6).

Proposition. The domain of a functional substitution *fs* is disjoint from *PR*. □

Notation. $x\backslash fs$ is the functional instance of the term or formula x by the functional substitution *fs*. We may call this a *simple functional instance* when *fs* is simple. We omit details of this definition, which may be carried out by recursion in a straightforward manner as in [2]. □

A.2 Correctness for Simple Functional Substitutions

Our first lemma makes rigorous the following key idea. If a theorem is proved that involves function symbols not mentioned in the axioms from which it is proved, then the meanings of those functions symbols are irrelevant to its provability. Hence, it remains a theorem when those function symbols are replaced.

Lemma. Suppose that ψ is a theorem of a given first-order theory T and that *fs* is a simple functional substitution whose domain is disjoint from the set of function symbols of T. Then $\psi\backslash fs$ is a theorem of T.

Proof. A proof-theoretic argument can presumably be made, since $A\backslash fs = A$ for every axiom A of T, and the rules of inference remain valid when applying *fs* to each of their applications. Here, however, we give a model-theoretic argument. First, note that it suffices to assume that the domain and range of *fs* are disjoint, since we may write *fs* as the composition of two appropriate functional substitutions with that property. (Briefly: First map the symbols in the domain to distinct symbols occurring nowhere in sight, and then map those to the final values.) Now given any model of T (which also interprets the symbols in the range of *fs*, but not in the domain of *fs*) that satisfies the negation of $\psi\backslash fs$, we may expand this to a model of the negation of ψ by interpreting every function symbol in the domain of *fs* to be the interpretation of the corresponding function symbol in the range of *fs*. $\square$

Theorem. Suppose that *fs* is a simple functional substitution with respect to a history h and that φ is a theorem of h. Suppose further that h' is a subsequence of h that is closed under ancestors, such that φ is a formula of the language of h'. Finally, suppose that for every labeled formula A of h', $A\backslash fs$ is a theorem of h. Then $\varphi\backslash fs$ is a theorem of h.

Proof. Since φ is (by assumption) a theorem of h, it is also a theorem of h', by the History Conservativity Theorem. Fix a proof of φ from the theory of h' (which includes induction axioms). Thus, letting A be the conjunction of the axioms and induction axioms of h' used in the proof, we have that $(A \rightarrow \varphi)$ is a first-order theorem of GZ. By the immediately preceding proposition and lemma, it follows that $(A \rightarrow \varphi)\backslash fs$ is a first-order theorem of GZ, i.e., that $(A\backslash fs \rightarrow \varphi\backslash fs)$ is a theorem of GZ, hence of h. Hence $\varphi\backslash fs$ is a theorem of h, because each conjunct of $A\backslash fs$ is a theorem of h by hypothesis — except, a separate argument needs to be made for induction axioms. That is, we claim that the simple functional instance of an induction axiom of h' is an induction axiom of h. This is clear from the form of induction axioms, since every simple functional instance by *fs* of such a formula is of that form as well. $\square$

A.3 Correctness for Functional Substitutions

In the ACL2 implementation of functional instantiation, a functional substitutions is permitted to map a function symbol f to a pseudo-function `(LAMBDA` *vars term*`)`, where the length of *vars* is the arity of f. Moreover, this pseudo-function may include free variables. So, we need to remove the restriction to *simple* functional substitutions from the theorem above. The theorem below also restricts the proof obligations in an obvious way, to functional instances of theorems that contain at least one function symbol bound in the functional substitution.

But first we state a simple lemma that is used in the proof.

Lemma. let T_1 be an inductively complete theory and let C be a set of zero-ary function symbols (i.e., constants) that is disjoint from the language of T_1. Let T_2 be the inductive completion of T_1 with respect to the union of C with the language of T_1. Then for any formula φ_1 in the language of T_1 and formula φ_2 resulting from substituting constants from C for some free variables of φ_1, if φ_2 is a theorem of T_2 then φ_1 is a theorem of T_1.

Proof. Suppose that φ_1 is not a theorem of T_1. Thus by the Completeness Theorem, there is a model of T_1 that satisfies the negation of φ_1 for some assignment s of values to its variables. An expansion of that model to the language of T_2 may be obtained by interpreting members of C occurring in φ_2 by the values of the variables under s that they replace, and interpreting the rest of C arbitrarily. That expanded model with assignment s satisfies T_2 and the negation of φ_2. Hence, φ_2 is not a theorem of T_2. $\Box$

The following theorem justifies ACL2's generation of proof obligations for uses of functional instantiation, other than the optimizations discussed in the next subsection. The subsequence h' in the theorem below is taken in the implementation to include all ancestors of all function symbols occurring in φ or any defaxiom; see the ACL2 documentation [8] for "CONSTRAINT".

Theorem (Correctness of Functional Instantiation). Suppose that *fs* is a functional substitution with respect to a history h and that φ is a theorem of h. Suppose further that h' is a subsequence of h that is closed under ancestors, such that φ is a formula of the language of h'. Finally, suppose that for every labeled formula A of h' such that some function symbol in the domain of *fs* occurs in A, $A\backslash fs$ is a theorem of h and no free variable of *fs* occurs in A. Then $\varphi\backslash fs$ is a theorem of h.

Proof. First, observe that if no function symbol in the domain of *fs* occurs in a formula A, then $A\backslash fs$ is just A. Therefore, we can remove the restriction to formulas A for which some function symbol in the domain of *fs* occurs in A.

Next, we reduce this theorem to the case that there are no free variables of *fs*. For if there are, first extend h to a new history h_0 that introduces, without extending the theory, a distinct zero-ary function symbol for each variable free in *fs*. Technically, h_0 is the extension of h by the formula `T` (*true*) with label `<constraint,` *s*`>`, where (say) s defines each new zero-ary function to have value 0. Now consider the functional substitution fs_0 obtained from *fs* by replacing each variable free in *fs* by the corresponding new constant (i.e., call of a new zero-ary function). Note that the hypotheses of the theorem are satisfied with h_0 in place of h, with fs_0 in place of *fs*, and with h' and φ unchanged: for if $A\backslash fs$ is a theorem of h, then it is also a theorem of h_0 and hence so is its instance $A\backslash fs_0$. (Why is $A\backslash fs_0$ an instance of $A\backslash fs$? This is left to the reader, but we note that here is where the hypothesis is used about free variables of *fs*.) Application of the theorem in this case would allow us to conclude that $\varphi\backslash fs_0$ is a theorem of h_0. The preceding lemma then allows us to conclude that $\varphi\backslash fs$

is a theorem of h.

So, let us assume that *fs* has no free variables. For each pair $\langle f,$ `(LAMBDA` *vars term*`)`$\rangle$ in *fs*, extend h by a definition $g(vars) = term$, where g is a new function symbol. Let us call the resulting history h_1, and let us write fs_1 for the functional substitution obtained by replacing each expression `(LAMBDA` *vars term*`)` as above by the corresponding new function symbol g. We claim that the hypotheses of the theorem hold for h_1 and fs_1 in place of h and *fs* (and with h' and φ unchanged); and for this claim, we only need show that for A as before, $A\backslash fs_1$ is a theorem of h_1, given that $A\backslash fs$ is a theorem of h. But this is clear since $A\backslash fs_1$ and $A\backslash fs$ are logically equivalent in h_1. The theorem in the preceding subsection now applies: $\varphi\backslash fs_1$ is a theorem of h_1. But $\varphi\backslash fs_1$ is logically equivalent in h_1 to $\varphi\backslash fs$, so $\varphi\backslash fs$ is a theorem of h_1. By the History Conservativity Theorem, $\varphi\backslash fs$ is a theorem of h. □

A.4 Correctness in the Presence of Optimizations

The following documentation for topic "`CONSTRAINT`" from [8] discusses certain optimizations performed by the ACL2 implementation when generating proof obligations for a use of functional instantiation. Below, we argue the correctness of these optimizations.

> First, we focus only on non-trivial encapsulations [those that have non-empty signatures] that neither contain nor are contained in non-trivial encapsulations. (Nested non-trivial encapsulations are not rearranged at all: do not put anything in such a nest unless you mean for it to become part of the constraints generated.) Second, in what follows we only consider the non-`local` events of such an `encapsulate`, assuming that they satisfy the restriction of using no locally defined function symbols other than the signature functions. Given such an `encapsulate` event, move, to just in front of it and in the same order, all definitions and theorems for which none of the signature functions is ancestral. Now collect up all formulas (theorems) introduced in the `encapsulate` other than definitional axioms. Add to this set any of those definitional equations that is either subversive [non-tight, in the terminology of this paper] or defines a function used in a formula in the set. The conjunction of the resulting set of formulas is called the "constraint" and the set of all the signature functions of the `encapsulate` together with all function symbols defined in the `encapsulate` and mentioned in the constraint is called the "constrained functions." Assign the constraint to each of the constrained functions. Move, to just after the `encapsulate`, the definitions of all function symbols defined in the `encapsulate` that have been omitted from the constraint.

We now demonstrate correctness of these optimizations by defining two corresponding transformations and showing that they preserve chronologies. That is, each transformation maps any suitable chronology to a sequence of labeled and unlabeled formulas, and our task will be to prove that the result is a chronology. It should then be clear that we have justified the optimizations described in the documentation quoted above.

Each of our two transformation rules is intended to transform chronologies constructed by successfully executing an **`encapsulate`** event. The rules are parameterized as follows.

- Let s_1, s_2 be a chronology.
- Let A be first-order derivable from s_1, s_2 together with associated induction schemes. Label A by **`<constraint,`** $H(s_2)$**`>`**, and abuse notation by calling this labeled formula "A" as well. Note that by Part (3) of the Main Lemma for Chronologies (Section 7), s_1, A is a history.
- Let $B \in s_2$.
- Let s_3 be the result of deleting B from s_2.
- Let A' be the result of replacing the formula of A with any first-order consequence of A in the language of s_1, s_3, and replacing the label of A by **`<constraint,`** $H(s_3)$**`>`**.

Given the parameters just defined, the rules are as follows. The notion of *tight* is introduced in Section 5 in the definition there of "Tight Definability." The proposition following that definition allows us to claim for rule [Back] below that the canonical measure for B is definable over $H(s_1)$.

- [**Front**] Suppose that every ancestor of B is introduced either in B (hence B is labeled) or in $H(s_1)$. Then the new sequence is s_1, B, A'.
- [**Back**] Suppose that B is the last labeled formula in s_2 that introduces no ancestor of the labeled formula A'. Moreover, suppose B is labeled by defun or defchoose and that if B is a recursive definition, then B is tight with respect to the union of *PR* with the set of function symbols introduced in $H(s_1)$. Then the new sequence is s_1, A', B.

Proposition. The [Front] rule preserves chronologies.

Proof. Fix the parameters defined above, and suppose that every ancestor of B is introduced either in B or in $H(s_1)$.

First suppose that B is a labeled formula. By the Ancestors Preserve History Lemma (Section 7), the sequence $H(s_1), B$ is a history. It then follows by the History Sufficiency Lemma (Section 7) that the sequence $H(s_1), B, H(s_3)$ is a history. By definition of history, it follows that $H(s_1), B, A'$ is a history. Therefore s_1, B, A' is a chronology, by two applications of the [Labeled extension] rule.

Now suppose that B is an unlabeled formula. By the Correctness of Chronologies Theorem, B is a theorem of $H(s_1), H(s_2)$. By the History Conservativity Theorem (Section 6), B is a theorem of $H(s_1)$. Thus, s_1, B is a chronology by the [Unlabeled extension] rule. Since $H(s_1), A$ is a history by the Main Lemma for Chronologies, and since s_3 is equal to s_2, then clearly $H(s_1), A'$ is a history, and hence s_1, B, A' is a chronology by the [Labeled extension] rule. $\Box$

Proposition. The [Back] rule preserves chronologies.

Proof. Fix the parameters defined above, and suppose that B is the last labeled formula in s_2 that introduces no ancestor of the labeled formula A'. Since s_1, s_2 is a chronology, then $H(s_1), H(s_2)$ is a history by the Main Lemma for Chronologies. As mentioned above, if B is a recursive definition then tightness allows us to assert that its canonical measure is definable over $H(s_1)$; and hence by History Conservativity Lemma (Section 6), the measure theorem for B is provable in $H(s_1)$. Thus it suffices to prove that s_1, A' is a chronology, since then by the [Labeled extension] rule, s_1, A', B is a chronology. (Notice that we could not guarantee that [Labeled extension] applies if B were labeled by constraint.)

We claim that $H(s_3)$ is closed under ancestors. It suffices to check that labeled formulas B' occurring after B in $H(s_3)$ do not have as an ancestor any function symbol introduced by B. By choice of B, we know that any later B' of $H(s_3)$ introduces an ancestor of A'; thus by definition of ancestor, all functions introduced by B' are ancestors of A'. Hence no function introduced by B can be an ancestor of B', since otherwise that function would be an ancestor of A'. It follows that $H(s_3)$ is closed under ancestors.

As already noted above, $H(s_1), H(s_2)$ is a history; hence by the Ancestors Preserve History Lemma (Section 7), $H(s_1), H(s_3)$ is a history. Now (the formula of) A' is a theorem of the history $H(s_1), H(s_2)$ because A is such a theorem and A' is a first-order consequence of A, by hypothesis. The formula of A' is in the language of $H(s_1), H(s_3)$, since no function symbol of the formula of A' is introduced by B. Hence the formula of A' is a theorem of the history $H(s_1), H(s_3)$ by the History Conservativity Lemma. It follows that $H(s_1), A'$ is a history, by definition of history. Therefore s_1, A' is a chronology, by the [Labeled extension] rule, and we have met the final proof obligation. $\Box$

B Conservativity of Skolemization

We wish to check the conservativity of Skolemization in our context, where each time a function symbol is introduced, so are all the induction axioms about the symbol (and the existing function symbols). If we were dealing with a logic that precluded non-standard numbers, we could manage more simply, because there would be no need to worry about the induction axioms introduced when we add a new function symbol (a Skolem function). That is: every sort of induction scheme is automatically true when the natural numbers of the model are standard! But we do not want to leave the realm of first-order logic if we can avoid it; for example, that would eliminate the potential for integrating some non-standard analysis into the system[4], and it would require us to re-think our proof theory. Fortunately, we'll see that we can assume that there is a definable enumeration of the universe. This enumeration permits the explicit definition of Skolem functions, by choosing the least witnesses in the sense of this enumeration. The remainder of this Appendix works out this argument.

We thank Jim Schmerl for suggesting the key ideas that allow us to carry out the argument allowing for an enumeration of the model.

The definition of *Skolem axiom* is given in Subsection 4.

Notation. Let F be a set of function and relation symbols that includes the unary relation symbol `Nat`. We write $\mathit{IND}(\texttt{Nat}, F)$ for the theory containing all induction axioms in the language of F. $\square$

The following simple lemma is surely well known.

Lemma. Suppose that T_1 is a subtheory of the first-order theory T_2 such that for every model M of T_1, every sentence A true in M, and every finite subset T_2' of T_2, there is a model of T_2' satisfying A. Then T_2 is a conservative extension of T_1.

Proof. Suppose for a contradiction that T_2 is not a conservative extension of T_1; say, φ is in the language of T_1 and is a theorem of T_2, but φ is not a theorem of T_1. By the Completeness Theorem for first-order logic, let M be a model of T_1 that satisfies $\neg\varphi$. By hypothesis and the Compactness Theorem of first-order logic, we may choose of model of T_2 that satisfies $\neg\varphi$. Thus T_2 has a model not satisfying φ, so φ is not a theorem of T_2, which contradicts the choice of φ. $\square$

The next lemma is not at all obvious. If we were not concerned about induction axioms, it would be trivial to conservatively extend a theory by adding a bijection between the universe and its natural numbers, by the downward Lowenheim-Skolem theorem of first-order logic. Something fancier is needed, however, if we want to preserve induction.

Lemma. Suppose that T is a first-order theory in the finite language F containing PR, which contains GZ as well as $\mathit{IND}(\texttt{Nat}, F)$. Then it is conservative to extend T to a theory T' by adding an axiom introducing a new unary function symbol g, asserting that g maps the universe 1-1 onto `Nat`, together with $\mathit{IND}(\texttt{Nat}, F \cup \{g\})$.

Proof sketch. We sketch two arguments, both of which are quite technical and model-theoretic, and both of which were suggested by Jim Schmerl. Let M be a model of T. In each case we show how to expand M (at least for M

[4]The addition of non-standard analysis to ACL2 is a major part of the doctoral research currently being undertaken by Ruben Gamboa.

countable, which is enough by the downward Lowenheim-Skolem theorem) to a model of the desired assertion on g together with $\mathit{IND}(\mathtt{Nat}, F \cup \{g\})$ (or, in the second argument, an arbitrary finite subset of this set). Thus, the preceding lemma yields the desired conservativity claim, by applying it to T and T'. The first argument requires background in models of arithmetic, while the second depends on a much deeper result than any needed for the first argument.

The more standard and direct argument uses a variant of Cohen's forcing technique from set theory, adapted to models of arithmetic. Thus, fix a countable model M of T. The partial order here consists of 1-1 "finite" (in the sense of M) functions from an initial segment of the interpretation of `Nat` in M, that are coded in M. A standard argument shows that a function through this partial order, generic with respect to all dense sets definable with parameters in M, gives the desired expansion of M. Details are omitted here. The main ideas (given here only briefly, for those familiar with forcing) are to prove first the usual truth lemma, and then to show that if p forces that $\varphi(\mathrm{x})$ defines an inductive subset of `Nat`, and if n is in `Nat`, then the set of conditions that force all predecessors of n to satisfy φ is dense below p. Hence if p is in the generic set, then a member of that dense set is also in the generic set, and hence φ holds for all predecessors of n in the generic model. Since n is arbitrary, then φ holds for all elements of `Nat` in the generic model, and the induction scheme has been verified.

The second argument uses a theorem of Jim Schmerl [17]. That theorem tells us that given any sentence φ in the language of Peano Arithmetic extended by F, any model M of $\mathit{IND}(\mathtt{Nat}, F)$ that satisfies φ, and any finite subset T' of T, there is a model M_1 of φ satisfying T' that is definable inside M and is contained in the interpretation in M of `Nat`, such that the interpretation in M_1 of `Nat` is isomorphic, via an isomorphism j definable in M, to the interpretation in M of `Nat`. (If necessary, it is certainly no problem to make a definitional extension of T before carrying out this argument, allowing the coding up finite sequences in the manner required by Schmerl's theorem.) Because M_1 is definable in M, it follows follows that M_1 is also a model of $\mathit{IND}(\mathtt{Nat}, F \cup \{g\})$, for any function g on M_1 definable in M. Now since M_1 is contained in the interpretation of `Nat` in M, there exists a 1-1 enumeration of the elements of M_1 that is definable in M. By composing it appropriately with the aforementioned definable isomorphism j, we have a 1-1 function g from M_1 onto the interpretation of `Nat` in M_1, which is definable in M and thus expands M_1 to a model of $\mathit{IND}(\mathtt{Nat}, F \cup \{g\})$, as argued above. □

The following theorem implies the Conservativity of Defchoose Lemma from Section 6, whose proof had been deferred.

Theorem. Suppose that T is a first-order theory in the language F, and that T contains the relativization of Peano Arithmetic to `Nat` as well as $\mathit{IND}(\mathtt{Nat}, F)$. Then it is conservative to extend T by adding any Skolem axiom introducing a new function symbol f together with $\mathit{IND}(\mathtt{Nat}, F \cup \{f\})$.

Proof. The preceding lemma allows us to extend the given theory conservatively to include a function g mapping the universe 1-1 onto N, together with $\mathit{IND}(\mathtt{Nat}, F \cup \{g\})$. Thus, by transitivity of conservativity it suffices to extend this new theory in the manner indicated. But this is easy because in the indicated theory with g, we may explicitly define the desired Skolem function f. We do so in simply by picking the least witness, i.e., among all appropriate values $g(n)$ the one chosen is that for which n is least (if any witness exists; else, 0, say). $\square$

References

[1] R. S. Boyer and J S. Moore. *A Computational Logic*, Academic Press: New York, 1979.

[2] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, 1991, pp. 7–26.

[3] R. S. Boyer, M. Kaufmann, and J S. Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancement, *Computers and Mathematics with Applications*, **5**(2) (1995) 27–62.

[4] B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, M. Srivas and A. Camilleri (eds.), Springer-Verlag, November, 1996, pp. 275–293.

[5] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*, Academic Press: London, 1997.

[6] B. Brock and J S. Moore. A Mechanically Checked Proof of a Comparator Sort Algorithm, URL `http://www.cs.utexas.edu/users/-moore/publications/csort/index.html` (submitted for publication) 1999.

[7] D. A. Greve, D. S. Hardin and M. M. Wilding, Efficient Simulation Using a Simple Formal Processor Model, Technical Report, Advanced Technology Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498, April, 1998.

[8] M. Kaufmann and J S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual.* URL: http://-www.cs.utexas.edu/users/moore/acl2/acl2-doc.html#User's-Manual.

[9] M. Kaufmann and J S. Moore. High-Level Correctness of ACL2: A Story. URL http://www.cs.utexas.edu/users/moore/acl2/reports/-story.txt, October, 1995.

[10] M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. URL http://www.cs.utexas.edu/users/moore/acl2/reports/-km97a.ps.

[11] M. Kaufmann and J Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. In *IEEE Transactions on Software Engineering* **23**(4), April, 1997, pp. 203–213.

[12] M. Kaufmann. ACL2 Support for Verification Projects. In *Proceedings 15th Int'l Conf. Automated Deduction*, C. Kirchner and H. Kirchner (eds.), LNAI 1421, Springer-Verlag, July, 1998, pp. 220–238.

[13] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*, Automated Reasoning Series, Kluwer Academic Publishers, 1996.

[14] J Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the AMD5$_K$86 Floating-Point Division Program. *IEEE Trans. Comp.* **47**(9), Sept. 1998, pp. 913–926. See also URL http://-devil.ece.utexas.edu/~lynch/divide/divide.html.

[15] D. Russinoff, "A Mechanically Checked Proof of Correctness of the AMD5$_K$86 Floating-Point Square Root Microcode," *Formal Methods in System Design Special Issue on Arithmetic Circuits*, 1997.

[16] D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithms of the AMD-K7^{TM} Processor URL `http://www.onr.com/-user/russ/david/k7-div-sqrt.html`.

[17] J. Schmerl. A reflection principle and its applications to nonstandard models. *J. Symbolic Logic* **60**, 1137–1152, December 1995.

[18] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

[19] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.

[20] Y. Yu. *Automated Proofs of Object Code for a Widely used Microprocessor*, Technical Report 92, Computational Logic, Inc., 1717 W. 6th, Austin, TX 78703, May, 1993. See URL http://www.cli.com/reports/.