Efficient execution in an automated reasoning environment

DAVID A. GREVE

Rockwell Collins Advanced Technology Center, Cedar Rapids, IA, USA

MATT KAUFMANN

Department of Computer Sciences, University of Texas at Austin, Austin, TX, USA
(*url:* http://www.cs.utexas.edu/users/kaufmann/)

PANAGIOTIS MANOLIOS

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
(*url:* http://www.cc.gatech.edu/home/manolios/)

J STROTHER MOORE

Department of Computer Sciences, University of Texas at Austin, Austin, TX, USA
(*url:* http://www.cs.utexas.edu/users/moore/)

SANDIP RAY

Department of Computer Sciences, University of Texas at Austin, Austin, TX, USA
(*url:* http://www.cs.utexas.edu/users/sandip/)

JOSÉ LUIS RUIZ-REINA

Dep. de Ciencias de la Computación e Inteligencia Artificial, Univ. de Sevilla, Seville, Spain
(*url:* http://www.cs.us.es/~jruiz/)

ROB SUMNERS

Advanced Micro Devices, Inc., Sunnyvale, CA, USA

DARON VROON

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
(*url:* http://www.cc.gatech.edu/home/vroon/)

MATTHEW WILDING

Rockwell Collins Advanced Technology Center, Cedar Rapids, IA, USA
(*url:* http://hokiepokie.org/)

Abstract

We describe a method that permits the user of a mechanized mathematical logic to write elegant logical definitions while allowing sound and efficient execution. In particular, the features supporting this method allow the user to install, in a logically sound way, alternative executable counterparts for logically defined functions. These alternatives are often much more efficient than the logically equivalent terms they replace. These features have been implemented in the ACL2 theorem prover, and we discuss several applications of the features in ACL2.

1 Introduction

This paper is about a way to permit the functional programmer to prove efficient programs correct. The idea is to allow the provision of two definitions of the program: an elegant definition that supports effective reasoning by a mechanized theorem prover, and an efficient definition for evaluation. A bridge of this sort, between clear logical specifications and efficient execution methods, is sometimes called "semantic attachment" of the executable code to the logical specification.

We describe an approach that has been implemented to support *provably correct* semantic attachment of efficient code within the framework of the ACL2 theorem prover. ACL2 is a logic based on functional Common Lisp (Steele, 1990). The logic is supported by a mechanized theorem proving environment in the Boyer–Moore tradition (Boyer & Moore, 1997). The acronym ACL2 stands for "A Computational Logic for Applicative Common Lisp." ACL2 has been used to mechanically reason about some of the largest commercial systems that have ever undergone formal verification (Brock *et al*., 1996; Brock & Hunt, 1999; Russinoff *et al*., 2005).

It is perhaps surprising to see a focus on semantic attachment in the context of ACL2 precisely because the logic is based on an efficient functional programming language, where the "default" semantic attachment is provided by the compiler. But logical perspicuity and execution efficiency are often at odds, as demonstrated by numerous examples in this paper.

Despite our focus on ACL2, we believe the techniques described here are of interest to any system that aims to support mechanized reasoning about programs in a functional programming language. We demonstrate the feasibility of supporting efficient reasoning about functional programs without having to give up execution efficiency.

This paper is a reflection of the importance that the ACL2 community places on efficient execution in the context of automated reasoning. To put our work in context, we start the remainder of this section with a description of the history of ACL2 focusing primarily on the need for efficient executability in industrial-strength automated reasoning projects. We then provide a brief overview of the ACL2 logic, its relationship with Common Lisp, and the features already implemented in the theorem prover to support execution. We then describe what this paper is about in greater detail.

1.1 A brief history of ACL2

We describe the history of ACL2 to make three points. First, mechanized formal methods now have a place in the design of digital artfacts. Second, formal models are much more valuable if they can not only be analyzed but also executed. This is a powerful argument for the use of an axiomatically described functional programming language supported by a mechanized theorem prover. Furthermore, industrial test suites put severe strain on the speed and resource bounds of functional models. Third, the starting point for this work on semantic attachment was a system already honed by decades of focus on efficient functional execution in a logical setting.

ACL2 descends from the Boyer–Moore Pure Lisp Theorem Prover, produced in Edinburgh in the early 1970s (Boyer & Moore, 1975). That system supported a

first-order mathematical logic based on a tiny subset of Pure Lisp. Constants were represented by variable-free applications of constructor functions such as `cons`, and ground terms were reduced to constants via an interpreter that doubled as a simplifier for symbolic expressions. Pressure to handle larger examples, specifically the operational semantics of the BDX 930 flight control computer in the late 1970s (Goldberg *et al.*, 1984), led to the abandonment of ground constructor terms as the representation of constants and the adoption of semantically equivalent quoted constants. At the same time, automatic semantic attachment was introduced to enable evaluation of recursively defined functions on such constants by invoking code produced by a translator from Boyer–Moore logic into the host Lisp and thence into machine code by the resident compiler (Boyer & Moore, 1979, 1981, 1997). This version of the Boyer-Moore theorem prover was called Nqthm.

By the mid-1980s, the Boyer–Moore community was tackling such problems as the first mechanically checked proof of Gödel's incompleteness theorem (Shankar, 1994) and the correctness of a gate-level description of an academic microprocessor (Hunt, 1994). These projects culminated in the late 1980s with the "verified stack" of Computational Logic, Inc. (Bevier *et al.*, 1989), a mechanically checked proof of a hierarchy of systems with a gate-level microprocessor design at the bottom, several simple verified high-level language applications at the top, and a verified assembler, linker, loader, and compiler in between. By the end of the 1980s, researchers in industry were attempting to use Nqthm to describe commercial microprocessor design components and exploit the formal descriptions both to verify properties and to simulate those designs by executing definitions in the Boyer–Moore logic.

In 1989, the ACL2 project was started, in part to address the executability demands made by the community. Instead of a small home-grown Pure Lisp, the ACL2 language extends a large subset of applicative (functional) Common Lisp. It can be built on top of most Common Lisp implementations as of this writing, and its compiler is the compiler of the underlying Common Lisp. Models of digital systems written in ACL2 can be analyzed with the mechanical theorem prover and also executed on constants. This duality has enabled industrial researchers to use functional Common Lisp to describe designs.

An ACL2 model of a Motorola digital signal processor, which was mechanically verified to implement a certain microcode engine, ran three times faster on industrial test data than the previous simulation engine (Brock & Hunt, 1999). At Advanced Micro Devices, the Register Transfer Logic (RTL) for the elementary floating-point operations on the AMD Athlon processor[1] was mechanically verified with ACL2 to be IEEE compliant. But before the modeled RTL was subjected to proof, it was executed on more than 80 million floating-point test vectors and the results were compared (identically) against the output of AMD's standard simulator (Russinoff & Flatau, 2000). Subsequently, proofs uncovered design bugs; the RTL was corrected and verified mechanically before the processor was fabricated. At Rockwell Collins, Greve *et al.* (2000) defined an ACL2 model of the microarchitectural design of the

[1] AMD, the AMD logo and combinations thereof, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

world's first silicon Java Virtual Machine, was used as the simulation engine and executed at about 50% of the speed of the previously written C simulator. Liu and Moore (2003) described another ACL2 model of the Java Virtual Machine, capable of executing many bytecode programs and including support for multiple threads, object creation, method resolution, dynamic class loading, and bytecode verification.

1.2 Syntax and semantics

Having motivated our interest in an axiomatically described functional programming language supported by a mechanized theorem prover, we now give a brief introduction to ACL2. Here we principally focus on the features of ACL2 that are relevant to this paper. The reader interested in learning ACL2 is referred to the ACL2 home page (Kaufmann & Moore, 2006), which contains extensive hypertext documentation on the theorem prover. In addition, two previous papers (Kaufmann & Moore, 1997, 2001) lay out the logical foundations of ACL2.

The syntax of the ACL2 logic is that of Lisp. For example, in ACL2, we write `(+ (expt 2 n) (f x))` instead of the more traditional $2^n + f(x)$. Terms are used instead of formulas. For example,

```
(implies (and (natp x) (natp y) (natp z) (natp n) (> n 2))
         (not (equal (+ (expt x n) (expt y n))
                     (expt z n))))
```

is Fermat's Theorem in ACL2 syntax. The syntax is quantifier free. Formulas may be thought of as universally quantified on all free variables. Fermat's Theorem may be read "for all natural numbers x, y, z, and $n > 2$, $x^n + y^n \neq z^n$." Case is generally unimportant; `expt`, `EXPT`, and `Expt` denote the same symbol. A semicolon (`;`) starts a comment for the remainder of the current line.

A commonly used data structure in Lisp is the list, which is represented as an ordered pair, <*head*, *tail*>, or in *dotted pair* `Lisp notation`, (*head* `.` *tail*). The Lisp primitive `car` returns the first component *head* of an ordered pair or list, and `cdr` returns the second component *tail* of an ordered pair and (hence) the tail of a list.

ACL2 provides macros whereby the user can introduce new syntactic forms by providing translators into the standard forms. *Macros* are functions that operate on the list structures representing expressions. For example, `(list` $x_1 x_2 \ldots x_n$`)` is translated to `(cons` x_1 `(cons` $x_2 \ldots$ `(cons` x_n `nil)...))` by defining `list` as a macro. Similarly, `cond` is a macro that translates

```
(cond (c1 value1)
      (c2 value2)
      ...
      (ck valuek))
```

to

```
(if c1 value1
    (if c2 value2
        (... (if ck valuek nil) ...))),
```

which returns $value_i$ for the least i such that c_i is true (*i.e.*, any value other than the "false" value `nil`), and otherwise returns `nil`. The expression `(let ((`var_1 $form_1$`) ... (`var_k $form_k$`)) `*expr*`)` represents the value of *expr* in an environment where each var_i is bound to $form_i$ in parallel; and `let*` is similar, except that the bindings are interpreted sequentially. The forms `mv` and `mv-let` implement multiple-valued functions in ACL2. In particular, `(mv` $\alpha_1 \ldots \alpha_n$`)` returns a "vector" of n values and `(mv-let (`$v_1 \ldots v_n$`)` α β`)` binds the variables v_i to the n values returned by α and then evaluates β. The meanings of most other Lisp primitives used in this paper should be clear from the context.

The applicative subset of Common Lisp provides a model of the ACL2 logic. One of the key attractions of ACL2 is that most ground expressions in the logic are executable, in the sense that they can be reduced to constants by direct execution of compiled code for the function definitions as opposed to, say, symbolic evaluation via the axioms.[2] This makes it possible to test ACL2 models on concrete data. Thus, ACL2 models can serve as simulation engines and can be formally analyzed to establish properties.

Consider the following recursive definition of a function that computes the length of a given list. Note that `defun` is the ACL2 (and Lisp) command for introducing definitions; here, we are defining `lng` to be a function of one argument, `x`, with the indicated body.

```
(defun lng (x)
  (if (endp x) 0 (+ 1 (lng (cdr x)))))
```

When such a definition is admitted to the logic, a new axiom is added; in this case:

Definitional Axiom

```
(equal (lng x) (if (endp x) 0 (+ 1 (lng (cdr x))))).
```

The so-called *definitional principle* requires the proof that the recursion in the definition is well founded, which in turn establishes that there exists a unique function satisfying the equation to be added as an axiom. The intention is that the resulting definition provides a conservative extension of the existing theory, and hence preserves consistency (Kaufmann & Moore, 2001). This intention explains the purpose of such a proof obligation. For example, without this check, the following "definition" with nonterminating recursion could be used to prove a contradiction.

```
(defun bad (x)
  (not (bad x)))
```

The proof obligation for a recursive definition also establishes that all calls of the function terminate (provided the machine has sufficient resources). ACL2 uses a default well-founded relation and guesses an appropriate *measure* to be applied to the function's arguments that is to decrease for each recursive call, but the user is able to override these defaults.

[2] We say "most" because it is possible to introduce undefined but constrained function symbols. See Section 4.3.

1.3 An interactive automatic theorem prover

The ACL2 subset of Common Lisp is formalized in a set of axioms and rules of inference that are, in turn, implemented in an automatic theorem prover. The prover applies a variety of symbolic manipulation techniques, including rewriting and mathematical induction. The theorem prover is automatic in the sense that no user input is expected once a proof attempt begins.

But in a more fundamental sense, the theorem prover is interactive. Its behavior is largely determined by the previously proved lemmas in its database at the beginning of a proof attempt. The user essentially programs the theorem prover by stating lemmas for it to prove, to use automatically in subsequent proofs. For example, an equality lemma can be used as a rewrite rule, an implication concluding with an equality can be used as a conditional rewrite rule, etc. Every lemma is tagged with pragmatic information, describing how the lemma is to be used operationally.

The theorem prover is invoked by the user to prove lemmas and theorems. But it is also invoked by the definitional principle, `defun`, to prove that a measure decreases in recursion and to establish certain type-like conditions on definitions, discussed further below. Thus, user guidance, in the form of appropriate lemma development, plays a role in the definition of new functions.

In an industrial-scale proof project, thousands of lemmas might have to be proved to lead the theorem prover to the proof of the target conjecture. However, ACL2 comes with a set of precertified "books" (files) containing hundreds of definitions and thousands of lemmas relating many of them. The user can include any of these books into a session to help configure the database appropriately. Commonly used books include those on arithmetic, finite sets, and record-like data structures.

Interesting proof projects require that the user intimately understand the problem being attacked and why the conjecture is a theorem. In short, effective users approach the theorem prover with a proof in mind and code that proof into lemmas developed explicitly for the conjecture, while leveraging the precertified books for background information. The theorem prover is more like an assistant that applies and checks the alleged proof strategy, forcing the user to confront cases that had escaped preliminary analysis. This process is very interactive and can be time consuming. Logs of failed proof attempts lead the user to discover new relationships and new conditions that often lead to re-statements of the main conjecture. A successful proof project is essentially a collaboration between the user and the theorem prover.

1.4 Guards and guard verification

A successful ACL2 definition adds a new axiom and defines (and generally compiles) the new function symbol in the host Common Lisp. For example, the above `defun` for `lng` is executed directly in Common Lisp. We refer to this program as the *Common Lisp counterpart* of the logical definition. Because Common Lisp is a model of the ACL2 axioms, ACL2 may exploit the Common Lisp counterpart and the host Lisp execution engine as follows: When a ground application of the defined symbol arises during the course of a proof or when the user submits a form to

the ACL2 read-eval-print loop, its value under the axioms may be computed with the Common Lisp counterpart in the host Lisp. For example, should `(lng '(1 2 3 4 5))` arise in a proof, ACL2 can use the Common Lisp counterpart of `lng` to compute `5` in lieu of deriving that value by repeated reductions using instantiation of the definitional axioms.

This simple story is complicated by the fact that not all Common Lisp functions are defined on all inputs, but the ACL2 axioms uniquely define each primitive. For example, the function `endp` is defined in Common Lisp to return `t` ("true") if its argument is the empty list, `nil` ("false") if its argument is an ordered pair, and is not defined otherwise. This allows the Common Lisp implementor to compile the test as a very fast pointer equality ("`eq`") comparison against the unique address of the empty list. However, `(endp 7)` is undefined in Common Lisp; implementations typically cause an error or, when code is compiled, may give unexpected results.

The Common Lisp standard (Steele, 1990) implicitly introduces the notion of "intended domain" of the primitives. The intended domain for `endp` consists of the ordered pairs and the empty list. ACL2 formalizes this notion with the idea of *guards*. The guard of a function symbol is an expression that checks whether the arguments are in the intended domain. It is permitted for ACL2 to invoke the Common Lisp counterpart of a function only if the arguments have been guaranteed to satisfy the guard.

ACL2 provides a way for the user to declare the guard of a defined function. In particular, we could define `lng` as follows:

```
(defun lng (x)
  (declare (xargs :guard (true-listp x)))
  (if (endp x) 0 (+ 1 (lng (cdr x)))))
```

where `(true-listp x)` is defined to recognize *true-lists*, which are lists that are terminated by the empty list, `nil`.

```
(defun true-listp (x)
  (if (consp x)
      (true-listp (cdr x))
    (eq x nil)))
```

ACL2 also provides a means, called *guard verification*, of proving that the guards on the input of a function ensure that all the guards in the body are satisfied. In principle, guard verification consists of two automated steps: (a) generating the *guard conjectures*, and (b) proving them to be theorems. The guard on both `(endp x)` and `(cdr x)` is that `x` is either a cons pair or a `nil`, which we write as `(cons-or-nilp x)`. The guard on `(+ i j)` is `(and (acl2-numberp i) (acl2-numberp j))`. The guard conjectures for `lng` are thus:

```
(and (implies (true-listp x)
              (cons-or-nilp x))          ; from (endp x) and (cdr x)
     (implies (and (true-listp x) (not (endp x)))
              (true-listp (cdr x)))      ; from (lng (cdr x))
     (implies (and (true-listp x) (not (endp x)))
```

```
(and (acl2-numberp 1)      ; from (+ 1 (lng ...))
     (acl2-numberp (lng (cdr x))))))
```

These are generated and proved after the definition of `lng` is admitted.

Thus, when the ACL2 theorem prover encounters `(lng '(1 2 3 4 5))`, it checks that the guard for `lng` is satisfied, that is, `(true-listp '(1 2 3 4 5))`. Since this is true and the guards for `lng` have been verified, we know that all evaluation will stay within the intended domains of all the functions involved. Thus, ACL2 is free to invoke the Common Lisp definition of `lng` to compute the answer `5`.

On the other hand, if ACL2 encounters `(lng '(1 2 3 4 5 . 7))`, a list that is terminated with the atom `7` instead of the empty list, the guard check fails and ACL2 is not permitted to invoke the Common Lisp counterpart. The value of the term is computed by other means, for example, application of the axioms during a proof, or by an alternative "safe" Common Lisp function that performs appropriate run-time guard and type checking at the cost of some efficiency. ACL2 defines such a function in Common Lisp, the so-called *executable counterpart*.

In general, ACL2 evaluation always calls the executable counterpart to evaluate a function call. But if the guard of the function has been verified and the call's arguments satisfy the function's guard, then the executable counterpart will invoke the more efficient Common Lisp counterpart to do the evaluation.

Note that by verifying the guards of a function, it is possible to execute code that is free of run-time type checks, without imposing logical or syntactic restrictions. However, we have found that it considerably simplifies the reasoning process to keep guards out of the logic (*i.e.*, out of the definitional axioms). For further details about guards and guard verification, see the ACL2 online documentation available from the ACL2 home page (Kaufmann & Moore, 2006).

Guard verification is but one of several features of ACL2 designed to allow the efficient execution of ground terms while preserving the axiomatic semantics of the language. Another such feature is the provision of single-threaded objects (Boyer & Moore, 2002), which allow destructive modification of some data structures. Still another feature, related to guards, is ACL2's support for Common Lisp inline-type declarations (and their proofs of correctness), which permits Common Lisp compilers to produce more efficient code assuming the declared types for the intermediate expressions.

1.5 What this paper is about

The novel idea in this paper is the use of proof to verify semantic attachments that are defined by the user. We see that ACL2's guard verification mechanism is the vehicle that manages this proof obligation. We introduce constructs `mbe` ("must be equal") and `defexec` that, while simple, are powerful tools for separating logical and execution needs. The presence of a general-purpose theorem prover allows logical definitions and executable code to be arbitrarily different in form, where one can use the full deductive power of the prover to relate them.

Suppose we have verified the guards of `lng` and encounter an application of `lng` to a true-list of length 10,000. The guard check would succeed and the Common

Lisp counterpart would be invoked. But since it is defined recursively, we are likely to get a stack overflow. Although the given definition of `lng` is mathematically elegant, for the purpose of efficient execution, it would have been better to define it as follows:

```
(defun lng (x)
  (declare (xargs :guard (true-listp x)))
  (lnga x 0))
```

where

```
(defun lnga (x a)
  (declare (xargs :guard (and (true-listp x) (integerp a))))
  (if (endp x) a (lnga (cdr x) (+ 1 a)))).
```

Since the function `lnga` is tail recursive, good Common Lisp compilers will compile this function into a simple loop with no stack allocation on recursive function calls. The first recursive definition of `lng` we presented in the paper is not tail recursive and would cause stack allocation on each recursive call.

One of the claimed advantages of ACL2 is that models permit both execution and formal analysis. But this presents a quandary. If we define `lng` so as to favor analysis, we may make it impossible to execute on examples of interesting scale. And if we define it to favor execution, we complicate formal proofs, perhaps quite significantly.

This paper presents an approach that allows the ACL2 user to have it both ways. In particular, we introduce two constructs `defexec` and `mbe` in the ACL2 theorem prover that make it possible to write:

```
(defexec lng (x)
  (declare (xargs :guard (true-listp x)))
  (mbe :logic
       (if (endp x) 0 (+ 1 (lng (cdr x))))
       :exec
       (lnga x 0))).
```

This definition incurs, in addition to normal termination and guard verification obligations, an additional proof obligation that the Common Lisp (`:exec`) counterpart will return the same answer as the logical (`:logic`) definition. More precisely, the guard verification obligation is extended by this additional proof obligation. Henceforth, when the theorem prover is reasoning about the function `lng`, it will use the original, elegant definitional equation. But when ground applications satisfying the guard arise, the tail-recursive "definition" is used (assuming that guard verification has already been completed).

While at first glance, this may appear to be the only reason to use `defexec` and `mbe`, we present several other contexts in this paper where the use of `defexec` and `mbe` affords an elegant solution in ACL2. For example, one problem that arises in the definition of some complex recursive functions is the need to introduce additional tests for the purpose of proving that the function terminates on all values of the parameters—a requirement for function admission in the logic—but these

additional tests *must* be optimized away to permit efficient execution. Consider the formal definition of an operational semantics for a nontrivial computing machine. The semantics may be well defined only on states satisfying a complicated global invariant, so that invariant must be checked in the definition to ensure admissibility. But checking the invariant at every step of subsequent execution is prohibitively expensive. By using the mechanisms described here, the state invariant can be checked once and then execution on ground applications no longer does the check—provided the "invariant" has been proved to be invariant. We illustrate this point in Section 4.2.

1.6 Related work

Weyhrauch (1980) coined the term *semantic attachment* for the mechanism in the FOL theorem prover by which the user could attach programs to logical theories. The programs were to be partial models of the theories. Manipulation of terms in the theories could be guided by computing with their semantic attachments. Thus, for example, the machine integer `0` could be attached to the logical constant function `zero` and the program for adding `1` to an integer could be attached to the Peano successor function, `succ`. Then, properties of `succ(succ(zero()))` could be computed via these attachments. In its original implementation, there was no provision for establishing the soundness of the attachments; the motivation of the work was to explore artificial intelligence and reasoning in particular.

Semantic attachment was an approach to the more general problem of *reflection*, which has come to denote the use of computation in a metatheory to derive theorems in a theory. Harrison (1995) provides an excellent survey of reflection.

For obvious reasons, when soundness is considered of great importance, work on reflection (which is often computation on ground terms in a formal metatheory) leads to the study of the relation between formal terms and the means to compute their values. This insight on reflection has been used in Nqthm and ACL2 to develop a notion of a program designed to compute the value of a given defined function on explicit constants (Boyer & Moore, 1981). This program is often referred to as the *executable counterpart* of the defined function; in ACL2, the executable counterpart calls the Common Lisp counterpart when the guards have been verified. The need to evaluate verified term transformers ("metafunctions") on ground constants, representing terms in the logic, has made it imperative to provide for both the efficient representation of ground terms (*e.g.*, `'(0 1)` as the "explicit value" of a ground term such as `(cons (zero) (cons (succ (zero)) nil))`) and the efficient computation of defined functions on those values. Indeed, this is a key facility that has permitted the Boyer–Moore provers to deal with large constants and encouraged the development of significant work in the operational semantics of microprocessors, virtual machines, and programming languages. These developments have also forced the implementors of ACL2 to support the theorem prover on Common Lisp (Steele, 1990) rather than a home-grown Pure Lisp that Nqthm supported, thereby exploiting the advantage of diverse development environments with efficient optimizing compilers.

Recently, many state-of-the-art theorem provers have adopted means of efficient computation on ground constants. For instance, execution capabilities have been added to Coq (Paulin-Mohring & Werner, 1993), HOL (Gordon *et al.*, 2003), Nuprl (Allen *et al.*, 1990), and PVS (Shankar, 1999; Crow *et al.*, 2001). Generally speaking, the features described here provide the ACL2 user with finer grained control over the code that is executed to compute ground terms. This is not unexpected, since ACL2 is more closely integrated to a production programming language than most other theorem provers, resulting in heavier execution performance demands by its industrial users.

Since the initial development of this paper, several ACL2 applications have used `mbe` and `defexec`. Cowles *et al.* (2003) implemented fast matrix algebra operations using `mbt`, which is a derivative of `mbe`. Matthews and Vroon (2004) also used `mbt` to define an efficient machine simulator. Davis (2004) implemented efficient finite set theory operations using `mbe`. Finally, a number of nontrivial applications of the `mbe` and `defexec` are described in an expanded version of this paper that is available as a University of Texas Technical Report (Greve *et al.*, 2006). These applications include algorithms for ordinal arithmetic and an efficient implementation of a unification algorithm.

1.7 Organization of this paper

The rest of this paper begins with a detailed description of the `mbe` and `defexec` features in the next section. Sections 3 and 4 provide extensive example applications of `mbe` and `defexec`. We conclude in Section 5.

The applications described in this paper can be broadly divided into two categories. Section 3 presents examples in which a function's natural definition is inefficient for execution and needs to be replaced with a suitable alternative definition for efficiency. Section 4 presents examples in which a natural definition is sufficient for execution, but is ineffective for reasoning in the logic. For clarity, the examples we illustrate here are, for the most part, pedagogical. However, as mentioned in the last paragraph of Section 1.6, the extended technical report (Greve *et al.*, 2006) provides many other nontrivial applications of `mbe` and `defexec`. We refer to this technical report as the "TR." While discussing examples in this paper, we often point to corresponding more complex applications described in the TR. The TR also provides more detailed explanations of some of the examples presented here.

ACL2 contains input files, such as the `books/defexec/` directory of the ACL2 distribution, in support of many of the applications in this paper. The information in this paper is intended to be consistent with those files, although we take liberties when appropriate, for example, omitting `declare` forms for brevity.

2 Attaching executable counterparts: `mbe` and `defexec`

Every defined function in ACL2 is automatically given an *executable counterpart* based on the definition. As mentioned in the preceding section, the executable counterpart calls the Common Lisp counterpart when the guards have been verified.

In the preceding section, we briefly introduced `mbe`, which allows the user to attach alternative executable code to logic forms. In this section, we describe `mbe` in some detail. We also introduce the `defexec` macro, which provides a way to prove termination of executable counterparts provided by `mbe`. Both `mbe` and `defexec` were introduced into Version 2.8 of ACL2 (March 2004).

We keep the description here relatively brief. For more details, we refer the reader to the hypertext ACL2 documentation available from the ACL2 distribution and from the ACL2 home page (Kaufmann & Moore, 2006). In particular, the `mbe` documentation topic provides a link to documentation for a macro `mbt` ("must be true"), which may be more convenient than `mbe` for some applications.

2.1 `MBE`

In the logic, (`mbe :logic` *logic_code* `:exec` *exec_code*) is equal to *logic_code*; the value of *exec_code* is ignored. However, in the execution environment of the host Lisp, it is the other way around: this form macroexpands simply to *exec_code*.

The guard proof obligations generated for the above call of `mbe` are (`equal` *logic_code exec_code*) together with those generated for *exec_code*. It follows that *exec_code* may be evaluated in Common Lisp to yield a result, if evaluation terminates, that is provably equal in the ACL2 logic to *logic_code*. These proof obligations can be easy to prove or arbitrarily hard, depending on the differences between *exec_code* and *logic_code*.

We now illustrate `mbe` using the following definition of a list length function, `lng`. This example was presented in the previous section, except that here we use `defun` instead of `defexec`, the latter being a feature to which we return later. The function `lnga` was defined in the previous section using tail recursion.

```
(defun lng (x)
  (declare (xargs :guard (true-listp x)))
  (mbe :logic
       (if (endp x) 0 (+ 1 (lng (cdr x))))
       :exec
       (lnga x 0)))
```

The above definition has the logical effect of introducing the following axiom, exactly as if the above `mbe` call were replaced by just its `:logic` part.

Definitional Axiom

```
(equal (lng x)
       (if (endp x) 0 (+ 1 (lng (cdr x))))).
```

On the other hand, after guards have been verified for `lng`, ACL2 evaluates calls of `lng` on true-list arguments by using the following definition in Common Lisp, obtained by replacing the `mbe` call above by its `:exec` part.

```
(defun lng (x)
  (lnga x 0))
```

Guard verification for lng presents the following proof obligations:

```
(and (implies (true-listp x)
              (true-listp x)) ; from (lnga x 0)
     (implies (true-listp x)
              (integerp 0))   ; from (lnga x 0)
     (implies (true-listp x)
              (equal (if (endp x) 0 (+ 1 (lng (cdr x))))
                     (lnga x 0))))           ; from the mbe call
```

The first two are trivial to prove. But the third, which comes from the mbe call, requires a key lemma relating lng and lnga. This lemma cannot even be stated until lng is admitted. Thus, the guard verification must be postponed by extending the above declare form:

```
(declare (xargs :guard (true-listp x) :verify-guards nil))
```

After lng is admitted (without guard verification), the following key lemma can be stated by the user and is proved automatically by induction.

```
(defthm lnga-is-lng
  (implies (integerp n)
           (equal (lnga x n)
                  (+ n (lng x)))))
```

Guard verification for lng then succeeds. After guard verification, but only then, calls of lng in ACL2 generate corresponding calls in Common Lisp of lng, and hence of lnga. (Before guard verification, calls of lng are evaluated by interpreting the definitional equation derived from the :logic part of the mbe.)

2.1.1 Remarks on MBE *implementation*

Mbe is defined as a macro. The form (mbe :logic *logic_code* :exec *exec_code*) expands in the logic to the function call (must-be-equal *logic_code exec_code*). Indeed, the guard we have been referring to for (mbe :logic *logic_code* :exec *exec_code*) is really the guard for (must-be-equal *logic_code exec_code*).

ACL2 gives special treatment to calls of must-be-equal in several places, so that from the perspective of the ACL2 logic, the ACL2 user is unlikely to see any difference between (mbe :logic *logic_code* :exec *exec_code*) and *logic_code*. For example, the proof obligations generated for admitting a function treat the above mbe term simply as *logic_code*. For those familiar with ACL2, we note that function expansion, :use hints, :definition rules, induction schemes, termination (admissibility) proofs, and generation of constraints for functional instantiation also treat the above mbe call as if it were replaced by *logic_code*. So, why not simply define the macro mbe to expand in the logic to its :logic code? We need the call of function must-be-equal for the generation of guard proof obligations.

Special treatment of must-be-equal is also given in creation of executable counterparts, evaluation within the ACL2 logic, and signature checking when translating to internal form. Although the idea of mbe is essentially rather straightforward, much

care has been taken to implement this feature to keep the user view simple while providing useful heuristics in the prover and sound implementation for the logic.

2.2 DEFEXEC

Evaluation of functions defined using `mbe` need not terminate, not even given unlimited computing resources. Consider the following silly example:

```
(defun silly (x)
  (declare (xargs :guard t))
  (mbe :logic (integerp x)
       :exec  (silly x)))
```

ACL2 has no problem admitting this function. Its guard verification goes through trivially because the `mbe` call generates this trivial proof obligation:

```
(equal (integerp x) (silly x))
```

However, evaluation of, say, `(silly 3)` causes a stack overflow, because the Common Lisp definition of `silly`, using the `:exec` part of the above definition, is essentially as follows:

```
(defun silly (x)
  (silly x))
```

Although it can sometimes be useful to introduce functions that do not terminate on all inputs, even of appropriate "type," nevertheless one often prefers a termination guarantee. We turn now to a mechanism that guarantees termination (given sufficient time and space), even for functions that use `mbe`.

Definitions made with the `defexec` macro have the same effect for evaluation as ordinary definitions (made with `defun`), but impose proof obligations that guarantee termination of calls of their executable counterparts on their intended domains. For example, if we use `defexec` instead of `defun` in the ACL2 definition of `silly` above that calls `mbe`, then ACL2 will reject that definition.

`Defexec` has the same basic syntax as the usual ACL2 definitional command, `defun`, but with a key additional requirement: the body of the definition must be a call of `mbe`. `Defexec` then generates an additional proof obligation guaranteeing termination of the `:exec` part under the assumption that the guard is true. This can be a nontrivial requirement if the definition is recursive.

Consider the following form:

```
(defexec fn (x)
  (declare (xargs :guard guard))
  (mbe :logic logic_code
       :exec  exec_code))
```

In addition to the corresponding `defun` (where `defexec` above is replaced by `defun`), this form generates the following *local* definition for the ACL2 theorem prover. Because it is `local`, the definition is ignored by Common Lisp; it is only used by the ACL2 logical engine, as described below.

```
(local (defun fn (x)
         (declare (xargs :verify-guards nil))
         (if guard exec_code nil)))
```

Thus, ACL2 must succeed in applying its usual termination analysis to *exec_code*, but where the guard is added as a hypothesis in each case. For example, if *exec_code* contains a recursive call of the form `(fn (d x))`, then ACL2 will have to prove that `(d x)` is "smaller than" `x` in the sense of an appropriate "measure," under the hypothesis of *guard*. ACL2 provides default notions of "smaller than" and "measure," but these can be supplied for the *exec_code* by way of an `xargs` or `exec-xargs` declaration; we refer the reader to the full documentation for these and other details.

3 Optimizing for execution

This section focuses on examples where the natural definition is modified to achieve efficient execution. We start by considering a simple list-sorting problem in Section 3.1; `mbe` and `defexec` allow us to use an efficient *in situ* quicksort for execution and a natural insertion sort algorithm for the purpose of reasoning. In Section 3.2, we then consider uses that optimize certain facets of functional evaluation. The TR discusses a more nontrivial example, namely, the use of `mbe` to attach efficient algorithms for ordinal arithmetic to logically elegant definitions. Both the efficient and elegant algorithms were devised by authors Manolios and Vroon, using a succinct representation of the ordinals up to ϵ_0 (Manolios & Vroon, 2003, 2006). In addition, they have been integrated into the ACL2 logic and form the basis of a powerful library of theorems for reasoning about ordinal arithmetic (Manolios & Vroon, 2004), which is now used to prove that user-submitted function definitions terminate.

3.1 Sorting a list

Consider the problem of sorting a list. The standard insertion sort algorithm is simple but inefficient, whereas an in-place quicksort can be efficient but complex. In this section, we illustrate the use of `mbe` to write a sorting function whose logical definition uses the simpler algorithm and whose definition for execution uses the more efficient algorithm.

The following simple insertion sort function serves as the logical view of sorting a list. Here, `<<` is a total order on the ACL2 universe (Manolios & Kaufmann, 2002).

```
(defun insert (e x) ; insert e into sorted list x
  (if (or (endp x) (<< e (car x)))
      (cons e x)
    (cons (car x) (insert e (cdr x)))))

(defun isort (x)    ; build up sorted list by insertion
  (if (endp x) () (insert (car x) (isort (cdr x)))))
```

Defining an efficient in-place quicksort requires the fast random access and fast random (destructive) update of an array. ACL2 supports the use of efficient array operations by the use of so-called *single-threaded objects* or *stobjs* (Boyer & Moore, 2002). Stobjs are declared by a special form `defstobj`, which takes a list of field descriptors, where each field can either be a single Lisp object or a resizable array of Lisp objects. For instance, the following declaration creates a stobj named `qstor` containing a single array field `objs`:

```
(defstobj qstor (objs :type (array t (0)) :resizable t))
```

A `defstobj` introduces functions for accessing and updating the fields in the stobj and resizing array fields. In the logic, these functions are defined as corresponding operations on lists representing the stobj array structure. However, under the hood, these functions perform fast array access and update operations. ACL2 imposes syntactic restrictions on functions that operate on stobjs to guarantee that only one reference to the stobj is ever created and that every function that modifies a stobj returns that stobj. The restrictions ensure that execution using destructive updates on arrays is consistent with the constructive list semantics in the logic.

Ray and Sumners (2002) present an efficient in-place implementation of quicksort in ACL2 using stobjs, which is similar to the classical imperative implementation of the algorithm. In particular, they define a function `sort-qs` that takes the above stobj `qstor` and two indices `lo` and `hi`, and sorts the portion of the array in the `objs` field of `qstor` between `lo` and `hi` (inclusive). Given this implementation, we can define a function `qsort`, which implements an efficient quicksort on lists, as follows:

```
(defun qsort (x)
  (with-local-stobj qstor
     (mv-let (result qstor)
         (let* ((size (length x))
                (qstor (resize-array size qstor))
                (qstor (load-list x 0 size qstor))
                (qstor (sort-qs 0 (1- size) qstor))
                (result (extract-list 0 (1- size) qstor)))
           (mv result qstor))  ; must return modified stobj
       result)))
```

The function `qsort` creates a "local" stobj `qstor`, allocates the stobj array, loads the array with the elements of the list, calls `sort-qs` to sort the array recursively in place, and finally copies the sorted array back to a list, which it then returns. The form `with-local-stobj` creates a stobj locally inside a function call, freeing the memory when the function returns.

The functions `isort` and `quicksort` are equal under the assumption that the list being sorted is a true-list.

```
(defthm qsort-equivalent-to-isort
  (implies (true-listp x)
           (equal (qsort x)
                  (isort x))))
```

With this theorem proven, we can now define our intended `defexec` function named `sort-list` for sorting lists with a guard assuming that the input list is a true-list.

```
(defexec sort-list (x)
  (declare (xargs :guard (true-listp x)))
  (mbe :logic (isort x) :exec (qsort x)))
```

Thus, while the optimized `qsort` is used for execution, the simple `isort` function is used for logical purposes. Using the logical definition, it is straightforward to prove that the function does indeed sort, that is, returns an ordered permutation of its input. To prove a theorem about `sort-list`, we simply prove the corresponding theorem about `isort` without considering the efficient implementation. For example, the following theorem specifies that `sort-list` is idempotent and is trivial to prove.

```
(defthm sort-list-idempotent
  (equal (sort-list (sort-list x)) (sort-list x)))
```

The price we pay for getting both execution speed and logical elegance is the proof of equivalence—a nontrivial one-time cost. Also, one can implement even more efficient versions for execution purposes to handle situations when the in-place quicksort becomes costly, for instance, by optimizing for cases when the list is almost sorted. `Mbe` allows us to optimize the `:exec` body for these cases without affecting the logical view of `sort-list` and the resulting proofs involving `sort-list`.

List sorting, of course, is one very trivial instance of the general approach in which `defexec` is used for separation of concerns that allows the use of an optimized definition for execution while still making it possible to use a logically simple definition for reasoning purposes. The approach has also been applied to define a propositional satisfiability checker in ACL2, where the logical view of the checker is provided by simply characterizing the notion of satisfiability using quantification, whereas the executable definition is implemented using Binary Decision Diagrams (Sumners, 2000).

3.2 Fine-grained optimization using `defexec`

We now consider another use of `mbe` and `defexec`, namely, as effective tools for providing fine-grained optimizations. In particular, we use them to implement function inlining, result memoization, and fast simulation of models of computing systems in ACL2.

3.2.1 Inlined functions

Executing a function call incurs the overhead for managing a call stack that stores the values of parameters, results, and local variables. While the penalty for a single function call is nominal, the total cost for all of the function calls in an execution can be substantial. Most modern compilers provide support for *inlining* function calls. *Inlining a function* is essentially the replacement of the call of a function with the body of the function under a substitution of parameters.

There is a standard approach to achieve the effect of inlining in ACL2. Consider a nonrecursive function f whose execution suffers from the cost of the overhead of function calls. Instead of defining this function, one can define a *macro* with a body that produces the definition of f. Since a macro is expanded before logical processing by the theorem prover or execution by the host Common Lisp, this removes the cost of function calls for execution. However, this approach is inefficient for reasoning in the logic because, unlike functions, macros are "syntactic sugar" to the logic. If an algorithm is modeled as a function, then the user can prove lemmas about that function and use them to guide proofs. On the other hand, macros are immediately expanded when a form is processed, and thus never appear in the logic. For instance, in the case of f above, suppose we want to define a new function g that calls f, and assume that we want to prove a lemma `L` about g that does not require reasoning about the code for f. If f were defined *as a function*, we could then instruct the theorem prover not to expand its body while proving `L`; however, if f is a macro, then we lose such control.

The dichotomy between the needs to inline function calls for execution and to preserve function calls for reasoning is resolved with the use of `defexec`. To support function inlining, we implement two macros: `defun-inline` and `defun-exec`. Users use `defun-inline` instead of `defun` if they intend for the function to be inlined, and `defun-exec` in place of `defun` otherwise. The two macros generate `mbe` forms, allowing us to address both logical and execution needs.

How are the macros implemented? We first define a function `exec-term` that takes a term and replaces every function call (*fn* ...) with (*fn*`-exec` ...). The `defun-inline` and `defun-exec` macros called with name *fn* and body *bdy* generate a `defexec` form with name *fn*, whose `:logic` definition is exactly *bdy*, and `:exec` definition is the result of applying `exec-term` to *bdy*. The forms also generate a macro with the name *fn*`-exec`, but in the case of `defun-exec`, this new macro simply expands to a call of *fn*, whereas for `defun-inline`, it expands to the application of `exec-term` to *bdy*.

Using `defun-inline` and `defun-exec` macros, a user can limit the cost of function calls during execution without losing the flexibility to control term expansion during proofs. As an example, consider the following definitions of functions `foo` and `bar` where we wish to inline all calls of `foo`. Then we can write the following two forms:

```
(defun-inline foo (x) (f (h x)))
(defun-exec bar (x) (foo x))
```

This generates the following functions and macros that achieve the intended effect of removing the function call of `foo` in the execution bodies of functions that call `foo` while leaving `foo` as a function in the logic. We assume that `f` and `h` have already been defined using `defun-inline` or `defun-exec`.

```
(defun foo (x)
  (mbe :logic (f (h x)) :exec (f-exec (h-exec x))))
(defmacro foo-exec (x)
  (list 'f-exec (list 'h-exec x)))
```

```
(defun bar (x)
  (mbe :logic (foo x) :exec (foo-exec x)))
(defmacro bar-exec (x)
  (list 'bar x))
```

3.2.2 *Function memoization*

Another common optimization encountered in functional languages is the memoization of function results. *Function memoization* entails the efficient storage and retrieval of the results of previous function calls and requires the ongoing access and maintenance of a table storing previous results. For efficiency, we use a stobj named `memo-tbl` to store previously computed results. The details of the implementation of the stobj and the functions to store and retrieve results from the stobj are not relevant to this paper. Instead, we focus on the usage of `defexec` in supporting memoization through an abstraction (macro) `defun-memo`, which generates two `defuns` along with several additional definitions and theorems to prove relevant properties of the functions. `Defun-memo`, when called with argument *fn*, generates a function named *fn*`-memo` that includes an additional parameter, namely, the stobj `memo-tbl`. *fn*`-memo` returns the result of the computation and a `memo-tbl`, which has been updated to incorporate this result if it is not found in the existing `memo-tbl` using macro `previous-rslt`. The `memo` functions that are generated call only other `memo` functions in order to pass the `memo-tbl` around to each function that is subsequently called. We tie these `memo` functions with the logical definitions by generating a `defexec` that creates a local `memo-tbl` stobj and calls the corresponding `memo` function. For instance, the call `(defun-memo foo (x) (f (h x)))` generates the following definitions (among many other theorems and definitions):

```
(defun foo-body (x memo-tbl)
  (mv-let (r memo-tbl) (h-memo x memo-tbl)
    (f-memo r memo-tbl)))

(defun foo-memo (x memo-tbl)
  (mv-let (exists rslt) (previous-rslt (foo x) memo-tbl)
    (if exists (mv rslt memo-tbl)
      (mv-let (r memo-tbl) (foo-body x memo-tbl)
        (let ((memo-tbl (update-rslt (foo x) r memo-tbl)))
          (mv r memo-tbl))))))

(defexec foo (x)
  (mbe :logic (f (h x))
       :exec (with-local-stobj memo-tbl
               (mv-let (rslt memo-tbl)
                   (foo-body x memo-tbl)
                 rslt))))
```

The function `foo-body` performs the evaluation of the body of `foo` with the additional access and update of previously computed results in the `memo-tbl`. The `foo-body` and `foo-memo` functions call other `memo-tbl` functions for functions that

the user specifies for memoization. The `defexec` form for each function uses a local stobj `memo-tbl` for the execution body, but has the desired body on the logical side.

3.2.3 Efficient machine simulators

As a final application of `defexec` for providing fine-grained user control, we discuss briefly its use for generating appropriate logical and executable definitions for a simple simulator for computing runs of system models. For more details, see the corresponding section in Greve *et al.* (2006).

We define a macro called `defsimulator` that takes a list of state variables along with terms defining the next-state value for each variable. Consider the following example in which the macro `defsimulator` is called to define a simple system.

```
(defsimulator simple (pc ra rb)
  (next-pc (cond ((and (eq (instr pc) 'bra) (= rb 0)) ra)
                 (t (1+ pc))))
  (next-ra (cond ((eq (instr pc) 'add) (+ ra rb))
                 ((integerp (instr pc)) (instr pc))
                 (t ra)))
  (next-rb (cond ((eq (instr pc) 'mov) ra)
                 ((eq (instr pc) 'cmp) (if (> ra rb) 1 0))
                 (t rb))))
```

Here `(instr pc)` defines some mapping from program counter values to instructions that serves as the definition of the program that will execute on the simple system. This example `simple` system has three state variables, named `pc`, `ra`, and `rb`. This is a trivial processor model with a program counter `pc` and two registers `ra` and `rb`. Each variable stores an integer counter value that is updated at every step to be the value defined by evaluating the `next-pc`, `next-ra`, or `next-rb` term using the current values for the state variables `pc`, `ra`, and `rb`. For the sake of reasoning in the logic, we prefer to define the state variables as functions of time—where time in this case is natural valued and specified by the parameter *n*. For example, the following definition of `pc` is generated for the `:logic` code of an `mbe` call:

```
(defun pc (n)
  (if (zp n) (initial-pc)
    (let ((pc (pc (1- n))) (ra (ra (1- n))) (rb (rb (1- n))))
      (cond ((and (eq (instr pc) 'bra) (= rb 0)) ra)
            (t (1+ pc))))))
```

We may then define a function `(machine-state n)` that returns a list of the state variables at time `n`: `(list (pc n) (ra n) (rb n))`. In systems with larger numbers of state variables, this approach to defining state variables as functions of *n* affords readable terms involving state variables and efficient, elegant reasoning about the properties of individual state variables that only require the expansion of

the function definitions for the state variables upon which the property depends; see for example, Russinoff *et al.* (2005) for a nontrivial example. The use of functions of time to represent the values of state variables can also be extended with additional parameters to elegantly handle vectors and hierarchy. However, for execution it is preferable to define a function `(run-state n state-vars)` that iterates for *n* steps, updating an array (a field of stobj `state-vars`) by storing the values of the state variables computed at each step.

The macro `defsimulator` creates the desired logic and executable definitions (and proofs showing their correspondence). The final "result" of this expansion of the `defsimulator` macro is the definition of `machine-state` given below. In the logic, `machine-state` computes a simple list composed of the values of `pc`, `ra`, and `rb` at time *n*. The execution body of `machine-state` includes the creation of a local stobj and the appropriate call of `run-state` and accumulation of the results into a list matching the result defined in the logic.

```
(defexec machine-state (n)
  (mbe :logic (list (pc n) (ra n) (rb n))
       :exec (with-local-stobj state-vars
               (mv-let (rslt state-vars)
                   (let ((state-vars (run-state n state-vars)))
                     (mv (list (pc-val state-vars)
                               (ra-val state-vars)
                               (rb-val state-vars))
                         state-vars))
                 rslt))))
```

4 Optimizing for proof

In the applications of the `mbe` and `defexec` features presented in the last section, the primary goal was to retain a natural and logically elegant definition of a function for reasoning in the logic, while attaching a more efficient definition for execution. Attaching efficiently executable functions to a logically elegant definition forms the key target application of these features. However, there are situations in which the more natural definition is efficient, but needs to be modified in order to facilitate *logical reasoning*. In this section, we show examples of such applications.

The necessity for a more complex logical definition with a natural, efficient body arises in practice for several reasons. First, our goal might be to define functions with nice algebraic properties that enable creation of elegant rewrite rules; the logical definition necessary to ensure such properties might be cluttered. Second, the function might be *reflexive*, that is, the natural definition might involve nested recursive calls; it is difficult to admit such functions using the definitional principle without cluttering the definition with so-called "termination tests." Third, the natural definition might be *partial*, that is, it might specify the value of the function only in some specific domain; since ACL2 is a logic of total functions, additional logical machinery is necessary to admit such definitions.

4.1 Normalized association lists

Our first example illustrates how the logical definitions of functions might be cluttered for the purpose of deriving nice algebraic properties. Consider the problem of defining functions `mget` and `mset` for accessing and updating elements in an association list. An association list in Lisp is essentially a list of pairs (*key* . *value*), which can be thought of as a finite function mapping each *key* to the corresponding *value*. The function `(mget a m)` takes a key `a` and a mapping `m` and returns the value currently associated with `a` in `m` or returns `nil` if no value is associated with `a` in `m`. The function `(mset a v m)` returns a new mapping that associates the key `a` with value `v` but otherwise preserves all associations in the mapping `m`.

For logical reasoning, it is convenient if we can define `mget` and `mset` such that the following are theorems.

1.
```
(defthm mget-of-mset
    (equal (mget a (mset b v m))
           (if (equal a b) v (mget a m))))
```

2.
```
(defthm mset-eliminate
    (equal (mset a (mget a m) m) m))
```

3.
```
(defthm mset-subsume
    (equal (mset a u (mset a v m))
           (mset a u m)))
```

4.
```
(defthm mset-normalize
    (implies (not (equal a b))
             (equal (mset b v (mset a u m))
                    (mset a u (mset b v m)))))
```

Notice that the conditions 1–3 have no hypothesis, and none of the theorems contains a hypothesis restricting `m` to be a well-formed association list. The theorems can thus be treated as elegant rewrite rules.

However, defining `mget` and `mset` so that these conditions are theorems is nontrivial. Here, we provide an overview of the steps involved in the definitions.

To get the last three properties mentioned above, we need a normalized representation for the finite mappings. We define a well-formed mapping to be a list of key-value pairs where the keys are strictly ordered by the total order `<<` (cf. Section 3.1). Furthermore, to satisfy `mset-eliminate`, we add the requirement that no key-value pair may have the value of `nil`—where `nil` is the default return value for `mget`. This notion of well-formed mapping is recognized by the following function `well-formed-map`:

```
(defun well-formed-map (m)
  (declare (xargs :guard t))
  (or (null m)
```

```
(and (consp m)
     (consp (car m))
     (well-formed-map (cdr m))
     (cdar m)
     (or (null (cdr m))
         (<< (caar m) (caadr m))))))
```

It is straightforward to define recursive functions `mset-wf` and `mget-wf` that satisfy the desired properties with the additional hypothesis of `(well-formed-map m)`. Each function recurs through the list of pairs until it finds the position in the list where the key fits (relative to the `<<` order on keys) and performs the appropriate return of associated value or update of the mapping. Finally, to remove this additional "well-formedness" hypothesis, we use a generic method discovered by Sumners (Kaufmann & Sumners, 2002). The method involves defining two functions `acl2->map` and `map->acl2`, so that `acl2->map` transforms an ACL2 object into a well-formed map and `map->acl2` inverts this transformation. The paper shows how to use these transformations to define functions that satisfy the desired theorems.

However, what about execution efficiency? The definitions of functions `mset-wf` and `mget-wf` are not optimized for execution, and the additional calls of the translation functions `acl2->map` and `map->acl2` are expensive. However, we needed these transition functions because we wanted the theorems described above to hold unconditionally; for execution, we can avoid them by placing appropriate conditions on the guard. The guard is defined as follows: We choose a "bad" key that we never expect to arise in the use of `mget` and `mset`. We then define two predicates `good-key` and `good-map` as follows: `(good-key a)` returns `T` if and only if `a` is not the single bad key chosen; `(good-map m)` is essentially `(well-formed-map m)`, with the additional requirement that none of the keys are the bad key. Under these hypotheses, we can show that the functions `acl2->map` and `map->acl2` are identity functions; we therefore can define efficient versions `mget-fast` and `mset-fast` that take advantage of this efficient guard to treat `m` essentially as an already normalized association list. Finally, we define `mget` and `mset` with `mbe`, to achieve both the algebraic properties and efficient execution as follows:

```
(defun mget (a m)
  (declare (xargs :guard (good-map m)))
  (mbe :logic (mget-wf a (acl2->map m))
       :exec (mget-fast a m)))

(defun mset (a v m)
  (declare (xargs :guard (and (good-key a) (good-map m))))
  (mbe :logic (map->acl2 (mset-wf a v (acl2->map m)))
       :exec (mset-fast a v m)))
```

The guard obligation for `mbe` (cf., Section 2.1) produces the following proof obligation for the definitions above, which are easy to discharge on the basis of the above argument.

```
(implies (good-map x)
         (equal (mget-wf a (acl2->map x))
                (mget-fast a x)))
```

If the domain of application allows us to strengthen further the guards for `mset` and `mget`, then many further optimizations would be possible. For example, if the domain were restricted to mapping with keys that were numbers, then we could use faster tests for equality and the ordering `<<` would reduce to `<` on numbers, which is a much faster test to compute. If we could assume that the key passed into `mset` was less than the least key in `m`, then we could simplify `mset` to be the following:

```
(defun mset-new (a v m)
  (declare (xargs :guard (and (good-key a) (good-map m)
                              (or (null m) (<< a (caar m))))))
  (mbe :logic (mset a v m) :exec (cons (cons a v) m)))
```

4.2 Reflexive functions: Adding tests for termination

In the preceding section, we saw how it can be useful to clutter function definitions in order to obtain elegant logical properties of those functions. In contrast, we now study a class of function definitions whose very admission to the ACL2 logic requires cluttering them with extra tests. Consider the following definition:

```
(defun weird-identity (x)
  (if (and (integerp x) (< 0 x))
      (+ 1 (weird-identity (weird-identity (- x 1))))
    0))
```

As discussed in Section 1.2, there is a termination proof obligation that requires `(weird-identity (+ -1 x))` to be suitably smaller than positive integer `x`. Unfortunately, it is clearly impossible to carry out any such proof until this definition has been admitted, that is, until the following axiom has been added:

```
(equal (weird-identity x)
       (if (and (integerp x) (< 0 x))
           (+ 1 (weird-identity (weird-identity (- x 1))))
         0))
```

The definition above is *reflexive*: it contains a recursive call with an argument that itself contains a recursive call. As seen above, the inner recursive call can occur in the proof obligation for admitting this function.

The experienced ACL2 user knows that a solution to this problem is to add an extra test for termination as follows:

```
(defun weird-identity-logic (x)
  (if (and (integerp x) (< 0 x))
```

```
    (let ((rec-call (weird-identity-logic (- x 1))))
      (if (and (integerp rec-call)
               (<= 0 rec-call)
               (< rec-call x))
          (+ 1 (weird-identity-logic rec-call))
        'do-not-care))
  0))
```

However, we would prefer to evaluate calls of a reflexive function without the additional termination tests. We realize this preference by using `mbe` as follows:

```
(defun weird-identity (x)
  (declare (xargs :guard (and (integerp x) (<= 0 x))))
  (mbe :logic
       (weird-identity-logic x)
       :exec
       (if (and (integerp x) (< 0 x))
           (+ 1 (weird-identity (weird-identity (- x 1))))
         0)))
```

The necessary proof obligations above are easily discharged once we have proved the following lemma:[3]

```
(implies (and (integerp x) (<= 0 x))
         (equal (weird-identity-logic x)
                x))
```

Note of course that the example described above is merely pedagogical; the `:exec` code for `weird-identity` could have simply been `x`, as in fact the proof obligation above demonstrates. However, nontrivial reflexive definitions arise in practice. The TR describes such a case study, namely, a sophisticated implementation of a unification algorithm using term dags (Ruiz-Reina *et al.*, 2006). Furthermore, Greve and Wilding (2003) described the use of the same approach in an efficient implementation of a path-finding algorithm in a graph.

Finally, we return to a point made about invariants in Section 1.5. The extra test in the definition of `weird-identity-logic` can be viewed as an invariant on the "state" `x`, assuming that the initial state satisfies the guard. The lemma above is sufficient to guarantee that this is truly an invariant, and hence can be optimized away for execution on states `x` satisfying the guard. See the aforementioned examples of linear pathfinding and unification for more elaborate examples of the insertion of invariants for termination.

4.3 Executable tail-recursive partial functions

As a final application of `mbe` and `defexec` in optimizing natural executable definitions for logical reasoning, we consider its use in efficiently executing tail

[3] ACL2 does all proofs automatically for the two definitions and the lemma.

recursive partial functions. Lisp programmers often write tail-recursive functions that terminate only on some specific intended domain. In this section, we show how to preserve the natural (partial) definition of tail-recursive equations by using `mbe` to associate it with an appropriate function introduced for the logic.

Consider the problem of introducing the following "definition" of tail-recursive factorial.

```
(equal (trfact n a)
       (if (equal n 0)
           a
         (trfact (- n 1) (* n a))))
```

Notice that the equation uniquely specifies the value of the `trfact` if and only if n is a non-negative integer; the recursion does not terminate if n is a negative integer, a nonintegral rational, or non-numeric. However, recall from Section 1.2 that the definitional principle of ACL2 can be used to introduce a recursive definition if and only if the recursion is well founded, that is, terminates for *all* inputs. Hence, we cannot use this principle to introduce the equation above as a definitional axiom.

Such nonterminating tail-recursive equations can arise in nontrivial contexts, for example, in formalizing microprocessor interpreters or low-level procedural programming languages (Moore, 2003). For example, the formal language interpreter is often defined in ACL2 by specifying a function `step` such that, given a machine state `s`, `(step s)` returns the state after executing one instruction from state `s`. One might then wish to formalize execution of the interpreter by the function `stepw` as follows:

```
(equal (stepw s)
       (if (halted s)
           s
         (stepw (step s))))
```

The equation above defines a unique value of `(stepw s)` only for those machine states `s` for which the interpreter terminates (*i.e.*, reaches a `halted` state).

ACL2 provides a generic mechanism, called the *encapsulation principle*, to introduce functions with axioms that do not fully specify the return value for all inputs. For instance, we can use encapsulation as follows to introduce a unary function `foo` constrained only to return a natural number:

```
(encapsulate
 (((foo *) => *))
 (local (defun foo (x) 1))
 (defthm foo-is-natural
   (natp (foo x))))
```

The first line `(((foo *) => *))` in the form above specifies that `foo` is a function of a single argument and returns a single value. The `defthm` command specifies the formula `(natp (foo x))` as a constraint on `foo`. To ensure consistency, one must exhibit that there exists *some* function, called a *local witness*, that satisfies the alleged constraints; in this case, the function that always returns `1` serves as a local witness.

Once the encapsulate event has been executed, the local witness is "forgotten" and foo is axiomatized to be a unary function satisfying only the specified constraints.

The encapsulation principle can be used to introduce tail-recursive partial functions in ACL2. In particular, Manolios and Moore (2003) show that given *any* tail-recursive equation, one can always define a local witness constrained to satisfy the equation. Using this observation, they define a macro called defpun that makes it possible to introduce equations such as trfact described above as follows:

```
(defpun trfact (n a)
  (if (equal n 0)
      a
    (trfact (- n 1) (* n a))))
```

The macro expands into an encapsulate form that introduces a local witness constrained to satisfy the defining equation. Unfortunately, however, because of the use of encapsulation, the defining equation is introduced as a *property* or *constraint* on the function trfact; no meaningful executable counterpart is provided to the host Common Lisp. Thus, even for arguments on which the recursion terminates, one cannot evaluate the function other than possibly by symbolic expansion of the defining equation. We remedy this situation with mbe and defexec.

Our solution is to define a new macro defpun-exec (Ray, 2004) that allows us to write the following form:

```
(defpun-exec trfact (n a)
  (if (equal n 0)
      a
    (trfact (- n 1) (* n a)))
  :guard (and (natp n) (natp a)))
```

In the logic, the effect is the same as that of defpun above, namely, the introduction of function trfact constrained to satisfy its defining equation. However, for arguments satisfying the guard, defpun-exec enables *evaluation* of the equation. Thus, we can evaluate (trfact 3 1) to 6.

How does defpun-exec work on the above example? First, it introduces a new function trfact-logic using defpun.

```
(defpun trfact-logic (n a)
  (if (equal n 0)
      a
    (trfact-logic (- n 1) (* n a))))
```

Next, it introduces the following form via defexec.

```
(defexec trfact (n a)
 (declare (xargs :guard (and (natp n) (natp a))
 (mbe :logic (trfact-logic n a)
      :exec (if (equal n 0) a (trfact (- n 1) (* n a))))))
```

The use of `defexec` rather than `defun` generates proof obligations that ensure the termination of the `:exec` body on the domain specified by its guard. With this form, the definitional axiom of `trfact` is merely the following:

```
(equal (trfact n a) (trfact-logic n a))
```

Since `trfact-logic` is constrained to satisfy exactly the same tail-recursive equation as the `:exec` code for `trfact` above, the guard obligation for `mbe`, namely, that the `:logic` and `:exec` forms be provably equal, is trivial. Finally, `defpun-exec` introduces the following trivial-to-prove theorem, which verifies that `trfact` also satisfies the desired defining equation.

```
(defthm trfact-def
  (equal (trfact n a)
         (if (equal n 0)
             a
           (trfact (- n 1) (* n a))))
  :rule-classes :definition)
```

The keyword `:definition` in the `:rule-classes` argument for the `defthm` command is a directive to the ACL2 theorem prover asking it to use this theorem as a defining equation for `trfact` for reasoning purposes. On the other hand, since `trfact` is *defined* rather than *constrained*, we can now perform efficient, nonlooping evaluation of `trfact` calls on inputs that satisfy its guard, using the `:exec` code in the underlying Common Lisp.

5 Conclusion

In this paper, we have discussed the need to combine efficient functional programming constructs with mechanized proof support. Our motivating examples come from industrial applications of the ACL2 system, in which hardware and software of industrial interest have been formally modeled. Those models have been used as efficient simulation engines or rapid prototypes and have also been subjected to mechanically checked proofs to establish properties of interest. The dual use of formal models—execution and proof—increases their value but puts great stress on the programming/logical language because there is frequently a tension between logical elegance and execution efficiency.

The main point of this paper is to show the utility of the feature `mbe` ("must be equal"), which allows the user to define a function in two different but provably equivalent ways to resolve this tension between execution and proof. Because of the presence of a theorem prover within the system, the two alternatives may be arbitrarily different as long as the user can guide the system to a proof of their equivalence under the hypotheses governing their use.

The obvious application of `mbe` is to provide both elegant and efficient definitions of elementary functions such as length, factorial, list reverse, and list sorting, and on more interesting applications such as ordinal arithmetic and record structure operations. `Mbe` is often so used.

However, this paper highlights less obvious uses. In particular, we noted that the principle of definition, which is necessary to guard against the introduction of unsoundness, may require the inclusion of run-time tests that can be shown to be unnecessary once the properties of the newly defined concept have been established. Using the new feature, we show how such run-time tests can be eliminated after the fact.

As another highlighted use of `mbe`, we show how it can be used to provide executable counterparts for some partially defined constrained functions. Until the introduction of `mbe` into the ACL2 system, it was not possible to compute the values of any constrained functions (except by symbolic deduction). In particular, we show how executable counterparts can be provided for partial tail-recursive functions. This is an important class of functions: most operational models of state machines, microprocessors, and low-level procedural programming languages are given by an iterated state-transition system that can naturally be expressed tail-recursively and whose termination is not guaranteed. We anticipate that the provisioning of partial functions with executable counterparts will hasten their adoption by the ACL2 community and will simplify system modeling in ACL2.

The most important lesson of this paper is perhaps that functional programming languages can benefit greatly from a focus on mechanically checked proofs. First, such a focus enables the dual use of functional formal models, and thus encourages the adoption of functional programming by user communities (such as microprocessor design teams) that do not traditionally use the paradigm. Second, the presence of a mechanical theorem prover can allow the user great flexibility in attaining efficient code while presenting correct definitions.

Acknowledgments

This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591, as well as the National Science Foundation under Grant Nos. ISS-0417413, CCF-0429924, and CCF-0438871. The work of the sixth author is partially supported by the Spanish Ministry of Education and Science project TIN2004-03884, which is cofinanced by FEDER funds. We thank Vernon Austel for a key idea that helped in the design of `mbe`. We also thank the anonymous referees for helpful expository suggestions.

References

Allen, S., Constable, R., Howe, D. & Aitken, W. (1990) The semantics of reflected proof. In: *Fifth Annual IEEE Symposium on Logic in Computer Science*, Mitchell, J. (ed). IEEE Computer Society Press, pp. 95–105.

Bevier, W. R., Hunt, Jr., W. A., Moore, J S. & Young, W. D. (1989) Special issue on system verification. *J. Automated Reasoning* **5**(4), 409–530.

Boyer, R. S. & Moore, J S. (1975). Proving theorems about pure Lisp functions. *J. ACM* **22**(1), 129–144.

Boyer, R. S. & Moore, J S. (1979). *A Computational Logic*. New York: Academic Press.

Boyer, R. S. & Moore, J S. (1981). Metafunctions: Proving them correct and using them efficiently as new proof procedures. *Pages 103–184 of:* Boyer, R. S. & Moore, J S. (eds), *The Correctness Problem in Computer Science*. New York: Academic Press.

Boyer, R. S. & Moore, J S. (1997) *A Computational Logic Handbook*, 2nd ed. New York: Academic Press.

Boyer, R. S. & Moore, J S. (2002) Single-threaded objects in ACL2. In: *Practical Aspects of Declarative Languages : 4th International Symposium, PADL 2002*, Krishnamurthi, S. & Ramakrishnan, C. R. (eds). Lecture Notes in Computer Science, vol. 2257. New York: Springer-Verlag. pp. 9–27 Available at: `http://www.cs.utexas.edu/users/moore/publications/stobj/main.ps.Z`

Brock, B. & Hunt, Jr., W. A. (1999) Formal analysis of the Motorola CAP DSP. In: *Industrial-Strength Formal Methods in Practice*, Hinchey, M. & Bowen, J. (eds), New York: Springer-Verlag, pp. 81–116.

Brock, B., Kaufmann, M. & Moore, J S. (1996) ACL2 theorems about commercial microprocessors. In: *Proceedings of the 1st International Conference on Formal Methods in Computer-Aided Design (FMCAD 1996)*. Srivas, M. & Camilleri, A. (eds). LNCS, vol. 1166. New York: Springer-Verlag, pp. 275–293.

Cowles, J., Gamboa, R. & van Baalen, J. (2003) Using ACL2 arrays to formalize matrix algebra. In: *Proceedings of the 4th International Workshop on the ACL2 Theorem Prover and Its Applications*, Hunt, Jr., W. A., Kaufmann, M. & Moore, J S. (eds), Available at: `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`

Crow, J., Owre, S., Rushby, J., Shankar, N. & Stringer-Calvert, D. (2001) *Evaluating, Testing, and Animating PVS Specifications*. Tech. rept. Menlo Park, CA: Computer Science Laboratory, SRI International. Available at: `http://www.csl.sri.com/users/rushby/-papers/attach.pdf`

Davis, J. (2004) Finite set theory based on fully ordered lists. In: *Proceedings of the 5th International Workshop on the ACL2 Theorem Prover and Its Applications*, Kaufmann, M. & Moore, J S. (eds). Available at: `http://www.cs.utexas.edu/users/moore/acl2/-workshop-2004/`

Goldberg, J., Kautz, W., Mellear-Smith, P. M., Green, M., Levitt, K., Schwartz, R. & Weinstock, C. (1984) *Development and Analysis of the Software Implemented Fault Tolerance (Sift) Computer*. Tech. rept. NASA Contractor Report 172146. Hampton, VA: NASA Langley Research Center.

Gordon, M., Hurd, J. & Slind, K. (2003) Executing the formal semantics of the Accellera property specification language by mechanised theorem proving. In: *Proceedings of the 12th International Conference on Correct Hardware Design and Verification Methods*, Geist, D. (ed). Lecture Notes in Computer Science, vol. 2860. New York: Springer-Verlag, pp. 200–215.

Greve, D. & Wilding, M. (2003) Using MBE to speed a verified graph pathfinder. In: *Proceedings of the 4th International Workshop on the ACL2 Theorem Prover and Its Applications*, Hunt, Jr., W. A., Kaufmann, M. & Moore, J S. (eds). Available at: `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`

Greve, D., Wilding, M. & Hardin, D. (2000) High-speed, analyzable simulators. In: *Computer-Aided Reasoning: ACL2 Case Studies*, Kaufmann, M., Manolios, P. & Moore, J S. (eds). New York: Kluwer Academic Press, pp. 113–136.

Greve, D. A., Kaufmann, M., Manolios, P., Moore, J S., Ray, S., Ruiz-Reina, J. L., Sumners, R., Vroon, D. & Wilding, M. (2006) *Efficient Execution in an Automated Reasoning Environment*. Tech. rept. TR-06-59. Department of Computer Sciences, University of Texas at Austin. Available at: `http://www.cs.utexas.edu/users/moore/publications/acl2-papers.-html#Utilities`

Harrison, J. (1995) *Metatheory and Reflection in Theorem Proving: A Survey and Critique*. Tech. rept. CRC-053. SRI International Cambridge Computer Science Research Centre. Available at: `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.html`

Hunt, Jr., W. A. (1994) *FM8501: A Verified Microprocessor*. Lecture Notes in Artificial Intelligence, vol. 795. Yew York: Springer-Verlag.

Kaufmann, M. & Moore, J S. (1997) *A precise description of the ACL2 logic*. Tech. rept. Department of Computer Sciences, University of Texas at Austin. See URL: `http://-www.cs.utexas.edu/users/moore/publications/acl2-papers.html#Foundations`.

Kaufmann, M. & Moore, J S. (2001) Structured theory development for a mechanized logic. *J. Automated Reasoning* **26**(2), 161–203.

Kaufmann, M & Moore, J S. (2006) *ACL2 Home Page*. Available at: `http://www.cs.-utexas.edu/users/moore/acl2`

Kaufmann, M. & Sumners, R. (2002) Efficient rewriting of data structures in ACL2. In: *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications*, Borrione, D., Kaufmann, M. & Moore, J S. (eds). Available at: `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/`

Liu, H. & Moore, J S. (2003) Executable JVM model for analytical reasoning: A study. In: *IVME '03: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*. Yew York: ACM Press, pp. 15–23.

Manolios, P. & Kaufmann, M. (2002) Adding a total order to ACL2. In: *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications*, Borrione, D., Kaufmann, M. & Moore, J S. (eds). Available at: `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/`

Manolios, P. & Moore, J S. (2003) Partial functions in ACL2. *J. Automated Reasoning* **31**(2), 107–127.

Manolios, P. & Vroon, D. (2003) Algorithms for ordinal arithmetic. In: *19th International Conference on Automated Deduction – CADE-19*, Baader, F. (ed). Lecture Notes in Artificial Intelligence, vol. 2741. New York: Springer-Verlag, pp. 243–257.

Manolios, P. & Vroon, D. (2004) Integrating reasoning about ordinal arithmetic into ACL2. *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design – FMCAD 2004*. Lecture Notes in Computer Science, vol. 3312. New York: Springer-Verlag.

Manolios, P. & Vroon, D. (2006) Ordinal arithmetic: Algorithms and mechanization. *J. Automated Reasoning*, **34**(4), 1–37.

Matthews, J. & Vroon, D. (2004) Partial clock functions in ACL2. *Proceedings of the 5th International Workshop on the ACL2 Theorem Prover and Its Applications*, Kaufmann, M. & Moore, J S. (eds). Available at: `http://www.cs.utexas.edu/users/moore/acl2/-workshop-2004/`

Moore, J S. (2003) Inductive assertions and operational semantics. In: *Correct Hardware Design and Verification Methods – CHARME 2003*, Geist, D. & Tronci, E. (eds). Lecture Notes in Computer Science, vol. 2860. New York: Springer-Verlag. pp. 289–303.

Paulin-Mohring, C. & Werner, B. (1993) Synthesis of ML programs in the system Coq. *J. Symbolic Comput.*, **15**, 607–640.

Ray, S. (2004) Attaching efficient executability to partial functions in ACL2. In: *Proceedings of the 5th International Workshop on the ACL2 Theorem Prover and Its Applications*, Kaufmann, M. & Moore, J S. (eds). Available at: `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/`

Ray, S. & Sumners, R. (2002) Verification of an in-place quicksort in ACL2. In: *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications*, Borrione,

D., Kaufmann, M. & Moore, J S. (eds). Available at: http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/

Ruiz-Reina, J. L., Martín, F. J., Alonso, J. A. & Hidalgo, M. J. (2006) Formal correctness of a quadratic unification algorithm. *J. Automat. Reason.*, **37**(1-2), 67–92.

Russinoff, D., Kaufmann, M., Smith, E. & Sumners, R. (July 2005) Formal verification of floating-point RTL *at amd* using the ACL2 theorem prover. In: *Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Simonov, N. (ed). Available at: http://sab.sscc.ru/imacs2005/papers/T2-I-94-1021.pdf

Russinoff, D. M. & Flatau, A. (2000) RTL verification: A floating-point multiplier. In: *Computer-Aided Reasoning: ACL2 Case Studies*, Kaufmann, M., Manolios, P. & Moore, J S. (eds). New York: Kluwer Academic Publishers. pp. 201–232.

Shankar, N. (1994) *Metamathematics, Machines, and Gödel's Proof*. Cambridge: Cambridge University Press.

Shankar, N. (1999) *Efficiently Executing* PVS. Project report. Menlo Park, CA: Computer Science Laboratory, SRI International,.

Steele, Jr., G. L. (1990) *Common Lisp the Language*. 2nd ed. Woburn, MA: Digital Press.

Sumners, R. (2000) Correctness proof of a BDD manager in the context of satisfiability checking. In: *Proceedings of the 2nd International Workshop on the ACL2 Theorem Prover and Its Applications*, Kaufmann, M. & Moore, J S. (eds). Available at: http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/

Weyhrauch, R. (1980) Prolegomena to a theory of mechanized formal reasoning. *Artif. Intell. J.* **13**(1), 133–170.