Efficient Execution in an Automated Reasoning Environment

David A. Greve[*] Matt Kaufmann[†] Panagiotis Manolios[‡]
J Strother Moore[§] Sandip Ray[¶] José Luis Ruiz-Reina[‖]
Rob Sumners[**] Daron Vroon[††] Matthew Wilding[‡‡]

December 18, 2006

Abstract

We describe a method to permit the user of a mathematical logic to write elegant logical definitions while allowing sound and efficient execution. We focus on the ACL2 logic and automated reasoning environment. ACL2 is used by industrial researchers to describe microprocessor designs and other complicated digital systems. Properties of the designs can be formally established with the theorem prover. But because ACL2 is also a functional programming language, the formal models can be executed as simulation engines. We implement features that afford these dual applications, namely formal proof and execution on industrial test suites. In particular, the features 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. We discuss several applications of these features.

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,

[*]Rockwell Collins Advanced Technology Center
[†]Dept. of Computer Sciences, Univ. of Texas at Austin
[‡]College of Computing, Georgia Institute of Technology
[§]Dept. of Computer Sciences, Univ. of Texas at Austin
[¶]Dept. of Computer Sciences, Univ. of Texas at Austin
[‖]Dep. de Ciencias de la Computación e Inteligencia Artificial, Univ. de Sevilla
[**]Advanced Micro Devices, Inc.
[††]College of Computing, Georgia Institute of Technology
[‡‡]Rockwell Collins Advanced Technology Center

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 [54]. The logic is supported by a mechanized theorem proving environment in the Boyer-Moore tradition [9]. The acronym ACL2 stands for "A Computational Logic for Applicative Common Lisp." We briefly describe ACL2 and its relationship to Common Lisp, establishing some relevant background.

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 efficiency.

1.1 A Brief History of ACL2

We briefly describe the history of ACL2 to make three points. First, mechanized formal methods now have a place in the design of digital artifacts. Second, formal models are much more valuable if they can not only be analyzed but 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 strains 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 [6]. 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 like `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 formal operational semantics of the BDX 930 flight control computer in the late 1970s, 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 so that recursively defined functions could be evaluated on such constants via the invocation of code produced by a translator from Boyer-Moore logic into the host Lisp (and thence into machine code by the resident compiler)[8, 7, 9]. 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 [52] and the correctness of a gate-level description of an academic microprocessor [26]. These projects culminated in the late 1980s with the "verified stack" of Computational Logic, Inc., [5], 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 to exploit those 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 [11]. At Advanced Micro Devices, the RTL for the elementary floating-point operations on the AMD Athlon^{TM} processor[1] was mechanically verified with ACL2 to be IEEE compliant. But before the modeled RTL was subjected to proof it was executed on over 80 million floating-point test vectors and the results were compared (identically) against the output of AMD's standard simulator [50]. Subsequently, proofs uncovered design bugs; the RTL was corrected and verified mechanically before the processor was fabricated. At Rockwell Collins, Greve *et al* [24] defined an ACL2 model of the microarchitectural design of the world's first silicon Java Virtual Machine which was used as the simulation engine and executed at about 50% of the speed of the previously written C simulator. Liu and Moore [33] describe 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 as needed for reading this paper. For more thorough treatments of the ACL2 logic, see [28, 29].

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,

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

```
(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 document should be clear from 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. Execution times can be roughly comparable to C.

[2]We say "most" because it is possible to introduce undefined but constrained function symbols. See Section 4.4.

Thus, ACL2 models can often serve as practical 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 [29]. This intention explains the purpose of such a proof obligation. For example, without this check, the following "definition" with non-terminating 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.

1.3 An Interactive Automatic Theorem Prover

The ACL2 subset of Common Lisp is formalized in a set of axioms and rules of inference which 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 starts.

But in a more fundamental sense, the theorem prover is interactive. Its behavior is largely determined by the previously proved lemmas in its data base 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 pre-certified "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 pre-certified 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 which often lead to restatements 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 [54] 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 `nil`, which we will 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, *i.e.*, `(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, *e.g.*, 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 runtime 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 [30].

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 [10], 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 will 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. While 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, which 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 will present several other contexts in this paper where the use of `defexec` and `mbe` affords an elegant solution in ACL2. For example, one problem which 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 non-trivial 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 with a simple pedagogical example in Section4.2. More complex examples may be found in the linear pathfinding example presented in [23] and a unification algorithm (see Section 4).

1.6 Putting This Work in Context

Weyhrauch [58] 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 [25] 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. It was in precisely this context that Boyer and Moore [8] introduced the notion of a program designed to compute the value of a given defined function on explicit constants. Such a program is now known 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, forced the implementors of Nqthm 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. It was this facility that permitted Nqthm 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 in turn led to increased demand for efficient computation and the eventual abandonment of the home-grown Nqthm version of Pure Lisp. This also led to the decision to base ACL2 [27] on Common Lisp [54] with a wide array of development environments with efficient optimizing compilers. The decision to build ACL2 on top of Common Lisp created the need for the formulation of guards as a means to ensure the correspondence between the axioms and the runtime environment.

Since then, many theorem provers have adopted means of efficient computation on ground constants (see for example [53, 1, 42, 18, 22]). 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 much more closely integrated to a production programming language than the theorem provers cited above and the execution performance demands made by its industrial users are consequently heavier.

Finally, since the initial development of this paper, several other ACL2 applications have used `mbe` and `defexec`. Cowles *et al* [17] implement fast matrix algebra operations using `mbt` which is a derivative of `mbe`. Matthews and Vroon [40] also use `mbt` to define an efficient machine simulator. Davis [19] implements efficient finite set theory operations using `mbe`.

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`.

The applications we describe in this paper can be broadly divided into two categories. Section 3 provides applications in which a function's natural definition is inefficient for execution and hence needs to be replaced suitably to obtain the desired efficiency. Section 4 provides applications in which a natural definition is sufficient for execution purposes, but is ineffective for reasoning in the logic.

We conclude the paper in Section 5.

ACL2 contains input files in support of the applications in this paper, in the `books/defexec/` directory of the ACL2 distribution. 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 [30]. 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 will evaluate 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`.)

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 applied 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` is essentially as follows, using the `:exec` part of the above definition.

```
(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 non-trivial 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 used only 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 in order 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 which optimize certain facets of functional evaluation. In Section 3.3 we then consider a more elaborate application, which uses `mbe` to develop efficient functions for computing results of ordinal arithmetic.

3.1 Sorting a List

Consider the problem of sorting a list. The standard insertion sort algorithm is simple but inefficient, while 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 will serve as the logical view of sorting a list. Here, `<<` is a total order on the ACL2 universe [34].

```
(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* [10]. 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 which 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 [45] 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` as follows, which implements an efficient quicksort on lists.

```
(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 non-trivial 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 which 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 while the executable definition is implemented using Binary Decision Diagrams [55].

3.2 Fine-grained Optimization using `defexec`

In this section we show that `mbe` and `defexec` can also be used as effective tools for providing fine-grained optimizations. In particular, we will 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 which 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 non-recursive function f whose execution suffers from the cost of function call overheads. Instead of defining this function, one can define a *macro* with a body that produces the code for 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 preserve 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, while 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 which achieve the intended effect of removing the function call of `foo` in the execution bodies of functions which 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 will use a stobj named `memo-tbl` in order 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 will focus on the usage of `defexec` in supporting memoization through an abstraction (macro) `defun-memo`, which generates two `defun`s 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` which 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 which are generated only call other `memo` functions in order to pass the `memo-tbl` around to each function. We tie these `memo` functions with the logical definitions by generating a `defexec` which 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 will call other `memo-tbl` functions for functions which 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 its use for generating appropriate logical and executable definitions for

a simple simulator for computing system models. To facilitate this we define a macro called `defsimulator` which takes a list of state variables along with terms defining the next-state value for each variable. We illustrate its use with an example. Consider the following call of the macro `defsimulator` which defines 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 which serves as the definition of the program which 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 which 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`. The following is generated for the `:logic` code of an `mbe` call, and the `mutual-recursion` wrapper informs ACL2 that, as the name implies, the functions defined within its scope are mutually recursive.

```
(mutual-recursion
(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))))))
(defun ra (n)
  (if (zp n) (initial-ra)
    (let ((pc (pc (1- n))) (ra (ra (1- n))) (rb (rb (1- n))))
       (cond ((eq (instr pc) 'add) (+ ra rb))
             ((integerp (instr pc)) (instr pc))
             (t ra)))))
(defun rb (n)
  (if (zp n) (initial-rb)
    (let ((pc (pc (1- n))) (ra (ra (1- n))) (rb (rb (1- n))))
      (cond ((eq (instr pc) 'mov) ra)
            ((eq (instr pc) 'cmp) (if (> ra rb) 1 0))
            (t rb)))))
)
```

```
(defun machine-state (n)
  (list (pc n) (ra n) (rb n)))
```

The function `(machine-state n)` returns a list composed of the value of each state variable at time `n`. In systems with larger numbers of state variables, this approach to defining state variables as functions of `n` affords more readable terms involving state variables and efficient, elegant reasoning about the properties of individual state variables which only require the expansion of the function definitions for the state variables upon which the property depends; see for example [49] for a non-trivial example. The use of functions of time to represent the values of state variables can also be extended with additional parameters to handle elegantly arrays and hierarchy.

While the definition of state variables as functions of time is effective in the logic, it is inefficient for the purpose of execution. For execution, it is preferable to update an array storing the values of the state variables at each step and to iterate through the desired number of steps for a given run up to time `n`. This preference leads to the following efficient execution definition of the function `(machine-state n)`, which uses a stobj to store the values of `pc`, `ra`, and `rb` at each step.

```
(defstobj state-vars pc-val ra-val rb-val)

(defun step-state-vars (state-vars)
  (let* ((pc (pc-val state-vars))
         (ra (ra-val state-vars))
         (rb (rb-val state-vars))
         (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)))
         (state-vars (update-pc-val next-pc state-vars))
         (state-vars (update-ra-val next-ra state-vars))
         (state-vars (update-rb-val next-rb state-vars)))
      state-vars))

(defun run-state-vars (n state-vars)
  (if (zp n) state-vars
    (let ((state-vars (step-state-vars state-vars)))
      (run-state-vars (1- n) state-vars))))

(defun run-state (n state-vars)
  (let* ((state-vars (update-pc-val (initial-pc) state-vars))
```

```
        (state-vars (update-ra-val (initial-ra) state-vars))
        (state-vars (update-rb-val (initial-rb) state-vars)))
   (run-state-vars n state-vars)))
```

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))))
```

3.3 Efficient Ordinal Arithmetic

In this section we will consider a more elaborate example of the use of `mbe`, namely in building an efficient and powerful library for reasoning about ordinal arithmetic. The ordinal numbers are an extension of the natural numbers into the transfinite. They were introduced by Cantor over 100 years ago and are at the core of modern set theory [12, 13, 14]. The ordinal numbers are important tools in logic, *e.g.*, after Gentzen's proof of the consistency of Peano arithmetic using the ordinal number ϵ_0 [21], proof theorists routinely use ordinals and ordinal notations to establish the consistency of logical theories [51, 57].

In computing science, the ordinals are used to prove termination of systems, an important component of total correctness proofs for transformational systems [2]. Even in the context of *reactive systems*, non-terminating systems that engage in on-going interactions with an environment (*e.g.*, operating systems and network protocols), termination proofs are important as they are used to prove *liveness properties*, *i.e.*, that some desired behavior is not postponed forever. Proving termination amounts to showing that a relation is well-founded [3]. From a basic theorem of set theory, the Axiom of Choice implies that every well-founded relation can be extended to a total order that is isomorphic to an ordinal. Thus, the ordinal numbers provide a general setting for establishing termination proofs.

The ACL2 system makes critical use of the ordinals: every function defined using the definitional principle must be shown to terminate using the ordinals up to ϵ_0. Up through Version 2.7, the ACL2 system provided a notation for representing these ordinals, a function to check if an object represents an ordinal in this notation, and a function for comparing the magnitude of two ordinals, but had only very limited support for reasoning about and constructing ordinals. In fact, while the set theoretic definitions of arithmetic operations were given by Cantor in the 1800's, algorithms for arithmetic operations on ordinal notations were not studied in any comprehensive way until recently, when Manolios and Vroon provided efficient algorithms, with complexity analyses, for ordinal arithmetic on the ordinals up to ϵ_0, using a notational system that is exponentially more succinct than the one used in ACL2 Version 2.7 [36, 39]. The above notations and algorithms were implemented in the ACL2 system, their correctness was mechanically verified, and a library of theorems developed that can be used to significantly automate reasoning involving the ordinals [37]. The library substantially increases the extent to which ACL2 can automatically reason about the ordinals, *e.g.*, it has been used to give a constructive proof of Dickson's lemma [56].

Starting with ACL2 Version 2.8, ordinals are now denoted using the new notation (which we introduce below) and a new, improved library for reasoning about the ordinals is provided [38]. The new library allows us to discharge automatically all the proof obligations involving the ordinals in the proof of Dickson's lemma, mentioned previously. There are many issues in developing such a library, but here we focus on the following very important consideration. The definitions of the ordinal arithmetic operations serve two purposes. They are used to reason about expressions in the ground (variable-free) case, by computation, and they are used to develop the various rewrite rules appearing in the libraries. In the first case, the definitions are used for computation, whereas in the second case, we use the definitions to reason symbolically. We use `mbe` in a way that allows us to compute efficiently in the ground case and to reason effectively in the general case. Here we discuss how this is accomplished. To keep the presentation here self-contained, we first provide a brief overview of ordinals and recount the current ordinal notations in ACL2. We then show how `mbe` allows us to use simple definitions of ordinal arithmetic operations for reasoning purposes and algorithmically more efficient ones for computation.

3.3.1 Set Theoretic Ordinals

We start with a brief review of the theory of ordinals [20, 32, 51]. A relation, $\prec$, is *well-founded* if every decreasing sequence is finite. A *woset* is a pair $\langle X, \prec\rangle$, where X is a set, and $\prec$ is a *well-ordering*, a total, well-founded relation, over X. An *ordinal* is a woset $\langle X, \prec\rangle$ such that for all $a \in X$, $a = \{x \in X \mid x \prec a\}$. It follows that if $\langle X, \prec\rangle$ is an ordinal and $a \in X$, then a is an ordinal and that $\prec$ is equivalent to $\in$. We will use lower case Greek letters to denote ordinals and $<$ to denote the ordering.

Given two wosets, $\langle X, \prec\rangle$ and $\langle X', \prec'\rangle$, a function $f : X \to X'$ is said to be

an *isomorphism* if it is a bijection and for all $x, y \in X$, $x \prec y$ iff $f(x) \prec' f(y)$. Two wosets are said to be *isomorphic* if there exists an isomorphism between them. A basic result of set theory states that every woset is isomorphic to a unique ordinal. Given a woset $\langle X, \prec \rangle$, we will denote the ordinal to which it is isomorphic as $Ord(X, \prec)$. Since every well-founded relation can be extended to a woset, we see that the theory of the ordinals is the most general setting for proving termination.

Given an ordinal, α, we define its *successor*, denoted α' to be $\alpha \cup \{\alpha\}$. There is clearly a minimal ordinal, $\emptyset$, which is commonly denoted by 0. The next smallest ordinal is $0' = \{0\}$ and is denoted by 1. Next we have $1' = \{0, 1\}$, which is denoted by 2. Continuing in this manner, we obtain all the natural numbers. A *limit ordinal* is an ordinal > 0 that is not a successor. The set of natural numbers, denoted ω, is the smallest limit ordinal.

3.3.2 Ordinal Arithmetic

In this section we define addition, subtraction, multiplication, and exponentiation for the ordinals. A discussion of these operators and their properties can be found in texts on set theory [20, 32, 51]. We start by defining ordinal addition. Informally, the idea is that $\alpha + \beta$ corresponds to the ordinal obtained by appending a copy of β after α.

Definition 1 *$\alpha + \beta = Ord(A, <_A)$ where $A = (\{0\} \times \alpha) \cup (\{1\} \times \beta)$ and $<_A$ is the lexicographic ordering on A.*

Note that addition is not commutative, *e.g.*, $1 + \omega = \omega$, whereas $\omega < \omega + 1$.

We now define ordinal multiplication. Informally, the idea is that $\alpha \cdot \beta$ corresponds to the ordinal obtained by making β copies of α.

Definition 2 *$\alpha \cdot \beta = Ord(A, <_A)$ where $A = \beta \times \alpha$ and $<_A$ is the lexicographic ordering on A.*

Note that commutativity and distributivity from the right do not hold for multiplication, *e.g.*, $2 \cdot \omega = \omega$, whereas $\omega < \omega \cdot 2$; also, $(\omega + 1) \cdot \omega = \omega \cdot \omega$, whereas $\omega \cdot \omega < \omega \cdot (\omega + 1) = \omega \cdot \omega + \omega$.

We define ordinal exponentiation using transfinite recursion. There are three cases: either the exponent is 0, or it is a successor ordinal, or it is a limit ordinal.

Definition 3 *Given any ordinal, α, we define exponentiation using transfinite recursion: $\alpha^0 = 1$, $\alpha^{\beta+1} = \alpha^\beta \cdot \alpha$, and for β a limit ordinal, $\alpha^\beta = \bigcup_{\xi<\beta} \alpha^\xi$.*

Using the ordinal operations, we can construct a hierarchy of ordinals: $0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots, \omega \cdot 2, \omega \cdot 2 + 1, \ldots, \omega^2, \ldots, \omega^3, \ldots, \omega^\omega, \ldots$, and so on. The ordinal $\omega^{\omega^{\omega^{\cdots}}}$ is called ϵ_0, and it is the smallest ordinal, α, for which $\omega^\alpha = \alpha$; such ordinals are called ϵ-ordinals.

3.3.3 Ordinal Notations

The theory of ordinal notations was initiated by Church and Kleene [15] and is recounted in Chapter 11 of Roger's book on computability [46]. A system of notations for ordinals up to some ordinal α consists of a constructive, syntactic way of denoting each and every ordinal less than α. Since we require notations to be strings from a countable alphabet, by a simple counting argument we see that if there is a system of notations for ordinal α, then α is countable. Notations are important in proof theory because they are used to obtain constructive proofs [51, 57]. Notations are important for our purposes because they allow us to compute.

The notations we consider deal with the ordinals less than ϵ_0 and are based on the Cantor's normal form theorem for the ordinals [51]. It states that for every ordinal $\alpha \neq 0$, there are unique $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_n$ $(n \geq 1)$ such that $\alpha = \omega^{\alpha_1} + \cdots + \omega^{\alpha_n}$. For every ordinal $\alpha < \epsilon_0$, we have that $\alpha < \omega^\alpha$, as ϵ_0 is the smallest ϵ-ordinal. Therefore, we can add the restriction that $\alpha > \alpha_1$ for such ordinals. This is essentially the representation of the ordinals used in ACL2 Version 2.7 [27]. However, since $\omega^\alpha \cdot k + \omega^\alpha = \omega^\alpha \cdot (k+1)$, we can collect like terms, giving us a more compact normal form. More specifically, for every ordinal $\alpha \in \epsilon_0$, there are unique $n, p \in \omega, \alpha_1 > \cdots > \alpha_n > 0$, and $x_1, \ldots, x_n \in \omega \backslash \{0\}$ such that $\alpha > \alpha_1$ and $\alpha = \omega^{\alpha_1} x_1 + \cdots + \omega^{\alpha_n} x_n + p$.

This is the representation of the ordinals that is used in ACL2 as of Version 2.8. It is exponentially more succinct than ACL2's previous notation, as can be seen by considering ordinals of the form $\omega \cdot k$, where $k \in \omega$. The previous notation requires $O(k)$ bits to represent such an ordinal, whereas our notation requires $O(\log k)$ bits.

The reason why a succinct notation is important is that we are interested in efficient algorithms. If the number of bits required to represent ordinals in one notation is potentially exponentially greater than the number of bits required with an alternate notation, then even linear time algorithms for the first notation can take longer to run than polynomial time algorithms for the second notation.

3.3.4 Ordinals in ACL2

Cantor Normal Form gives us a natural way to define *ACL2 ordinals*, our representation of the (set-theoretic) ordinals in ACL2: as lists of exponent-coefficient pairs. More precisely, given an ordinal α with Cantor Normal Form $\sum_{i=1}^{n} \omega^{\alpha_i} k_i + p$, the function *CNF*, below, returns the corresponding ACL2 ordinal, where `make-ord`, is an ACL2 function that constructs an infinite ACL2 ordinal, given its first exponent, its first coefficient and the rest of it.

$$CNF(\alpha) = \begin{cases} p & \text{if } n = 0 \\ \texttt{(make-ord}\ \ CNF(\alpha_1)\ \ k_1\ \ CNF(\sum_{i=2}^{n} \omega^{\alpha_i} k_i + p)\texttt{)} & \text{otherwise} \end{cases}$$

Descriptions of `make-ord` and other functions used to define the ACL2 ordinals are given in Figure 1. The definitions of these functions are not given here, but are covered in an earlier paper [37].

Function	Description
`(natp x)`	true iff `x` is a natural number.
`(posp x)`	true iff `x` is a positive integer.
`(finp x)`	true iff `x` is a finite ordinal.
`(infp x)`	true iff `x` is an infinite ordinal.
`(o-first-expt x)`	given $CNF(\sum_{i=1}^{n} \omega^{\alpha_i} x_i + p)$, returns $CNF(\alpha_1)$.
`(o-first-coeff x)`	given $CNF(\sum_{i=1}^{n} \omega^{\alpha_i} x_i + p)$, returns x_1.
`(o-rst x)`	given $CNF(\sum_{i=1}^{n} \omega^{\alpha_i} x_i + p)$, returns $CNF(\sum_{i=2}^{n} \omega^{\alpha_i} x_i + p)$.
`(make-ord x k y)`	returns the ordinal $\omega^{\mathtt{x}}\mathtt{k} + \mathtt{y}$ in our notation.
`(omega-term x k)`	returns the ordinal $\omega^{\mathtt{x}}\mathtt{k}$ in our notation.
`(first-term x)`	given $CNF(\sum_{i=1}^{n} \omega^{\alpha_i} x_i + p)$, returns $CNF(\omega^{\alpha_1} x_1)$.
`(natpart x)`	given $CNF(\sum_{i=1}^{n} \omega^{\alpha_i} x_i + p)$, returns p.
`(limitpart x)`	given $CNF(\sum_{i=1}^{n} \omega^{\alpha_i} x_i + p)$, returns $CNF(\sum_{i=1}^{n} \omega^{\alpha_i} x_i)$.
`(limitp x)`	true iff `x` represents a limit ordinal.
`(o< x y)`	true iff $\alpha < \beta$, where $CNF(\alpha) = \mathtt{x}$ and $CNF(\beta) = \mathtt{y}$.
`(o<= x y)`	true iff $\alpha \leq \beta$, where $CNF(\alpha) = \mathtt{x}$ and $CNF(\beta) = \mathtt{y}$.
`(ocmp x y)`	returns `lt`, `gt`, or `eq` if `x` is $<$, $>$, or $=$ `y`, respectively.

Figure 1: Basic Ordinal Functions

In the next section, we will give the complexity of various operations on ACL2 ordinals. We have disregarded the time complexity of operations on natural numbers in our analysis in order to focus on the interesting aspects of our algorithms, namely those pertaining to the structure of the ACL2 ordinals. The following two functions will play an important role in this regard.

```
(defun len (x)
  (if (finp x)
      0
    (+ 1 (len (o-rst x)))))
```

```
(defun sz (x)
  (if (finp x)
      1
    (+ (sz (o-first-expt x))
       (sz (o-rst x)))))
```

By this standard, the first ten functions listed in Figure 1 run in constant time, `natpart`, `limitpart`, and `limitp` run in linear time in `(len x)`, and `o<`, `o<=`, and `ocmp` run in time $O(\min(\texttt{(sz x)}, \texttt{(sz y)}))$ [36].

3.3.5 Attaching Executable Functions to Definitions of Ordinal Operations

We show how to attach efficient executable functions to simple definitions of ordinal operations, using `mbe`. The operations we consider are ordinal multiplication and exponentiation, and we compare the simple `:logic` definitions, used for symbolic manipulation, with the efficient `:exec` definitions, used for computation. To simplify the presentation we ignore guards and their verification (see Section 1.4).

```
(defun ob+ (x y)
  (let* ((fe-x (o-first-expt x))   (fco-x (o-first-coeff x))
         (fe-y (o-first-expt y))   (fco-y (o-first-coeff y))
         (cmp-fe (ocmp fe-x fe-y)))
    (cond ((and (finp x) (finp y)) (+ x y))
          ((or (finp x) (eq cmp-fe 'lt)) y)
          ((eq cmp-fe 'gt) (make-ord fe-x fco-x (ob+ (o-rst x) y)))
          (t (make-ord fe-y (+ fco-x fco-y) (o-rst y))))))

(defun dropn (n a)
  (cond ((or (finp a) (zp n)) a)
        (t (dropn (1- n) (o-rst a)))))

(defun count1 (x y)
  (cond ((finp x) 0)
        ((o< (o-first-expt y) (o-first-expt x))
         (+ 1 (count1 (o-rst x) y)))
        (t 0)))

(defun count2 (x y n) (+ n (count1 (dropn n x) y)))

(defun padd (x y n)
  (cond ((or (finp x) (zp n)) (o+ x y))
        (t (make-ord (o-first-expt x) (o-first-coeff x)
                     (padd (o-rst x) y (1- n))))))

(defun pmult (x y n)
  (let* ((fe-x (o-first-expt x))   (fco-x (o-first-coeff x))
         (fe-y (o-first-expt y))   (fco-y (o-first-coeff y))
         (m    (count2 fe-x fe-y n)))
    (cond ((or (equal x 0) (equal y 0)) 0)
          ((and (finp x) (finp y)) (* x y))
          ((finp y) (make-ord fe-x (* fco-x fco-y) (o-rst x)))
          (t (make-ord (padd fe-x fe-y m)
                       fco-y
                       (pmult x (o-rst y) m))))))

(defun ob* (x y)
  (mbe
   :logic (let ((fe-x (o-first-expt x))   (fco-x (o-first-coeff x))
                (fe-y (o-first-expt y))   (fco-y (o-first-coeff y)))
            (cond ((or (equal x 0) (equal y 0)) 0)
                  ((and (finp x) (finp y)) (* x y))
                  ((finp y) (make-ord fe-x (* fco-x fco-y) (o-rst x)))
                  (t (make-ord (o+ fe-x fe-y) fco-y (ob* x (o-rst y))))))
   :exec (pmult x y 0)))
```

Figure 2: Ordinal Multiplication

Ordinal multiplication is presented in Figure 2. Multiplication depends on `ob+`, a function that adds ACL2 ordinals, *i.e.*, if `x` equals $CNF(\alpha)$ and `y` equals $CNF(\beta)$, then `(ob+ x y)` equals $CNF(\alpha + \beta)$. Given arguments `x` and `y`, `ob+` traverses `x` until an exponent is found in `x` that is less than or equal to the first exponent of `y`. Note that the complexity of `(ocmp x y)` is $O(\min(\texttt{(sz x)}, \texttt{(sz y)}))$ [36]. Therefore, the running time of `ob+` is $O(\min(\texttt{(sz x)}, \texttt{(len x)} \cdot \texttt{(sz (o-first-expt y))}))$, because in the worst case we have to compare every exponent of `x` with the first exponent of `y`. In addition to `ob+`, we define `o+`, a macro that accepts an arbitrary number of arguments and macro-expands to calls of `ob+`. Similarly, we have `o*` for `ob*` and `o^` for `ob^`.

Now consider the running time of the `:logic` definition of `(ob* x y)`. In the worst case, `x` and `y` are both infinite ordinals. When this happens, we traverse `y`, adding the first exponent of `x` with each exponent of `y`. Thus, the running time of this algorithm is

$$O(\min(\texttt{(len (o-first-expt x))} \cdot \texttt{(sz y)}, \texttt{(len y)} \cdot \texttt{(sz (o-first-expt x))}))$$

This algorithm is inefficient. The problem arises because we are adding on the left some ordinal, `c` (the first exponent of `x`), to each of a decreasing sequence of ordinals $(\mathtt{d}_1, \mathtt{d}_2, \ldots, \mathtt{d}_n)$ (the exponents of `y`). Using the addition algorithm, we find that for each $\mathtt{d}_i$, `(o-first-expt` $\mathtt{d}_i$`)` is compared to each exponent of `c` until the first exponent of `c` such that `(o-first-expt` $\mathtt{d}_i$`)` is $\geq$ this exponent is found. But since the $\mathtt{d}_i$'s are decreasing, we know that `(o-first-expt` $\mathtt{d}_i$`)` $\geq$ `(o-first-expt` $\mathtt{d}_{i+1}$`)`. Therefore, if the jth exponent of `c` is greater than `(o-first-expt` $\mathtt{d}_i$`)`, it must be greater than `(o-first-expt` $\mathtt{d}_{i+1}$`)`. This means that simply adding each element of the decreasing sequence to `c` is inefficient. If we can keep track of how many exponents of `c` we went through before adding $\mathtt{d}_i$, we can just skip over those when we add $\mathtt{d}_{i+1}$. This is what `padd` allows us to do. We take advantage of `padd` in the `:exec` version of `ob*`, which calls `pmult`, which in turn calls `padd`. The function `(padd x y n)` appends the first `n` elements of `x` to the result of adding the rest of `x` to `y`. Comparing this to `ob+`, it is easy to see that if the first `n` exponents of `x` are greater than the first exponent of `y`, `(padd x y n)` = `(ob+ x y)`.

The complexity of `(pmult x y s)` is $O(\texttt{(len (o-first-expt x))} \cdot \texttt{(len y)} + \texttt{(sz (dropn (o-first-expt x) s))} + \texttt{(sz y)})$ when `s` $\leq$ `(count1 x y)`. Thus the complexity of the `:exec` version of `(ob* x y)` is $O(\texttt{(len (o-first-expt x))} \cdot \texttt{(len y)} + \texttt{(sz (o-first-expt x))} + \texttt{(sz y)})$ [36].

Improving the complexity of ordinal exponentiation is somewhat more difficult. The definition of exponentiation is given in Figure 3. The inefficiency in the `:logic` version is due to the repeated use of multiplication. Since we know some very specific things about the ordinals we are multiplying, we can find more efficient ways of calculating these products (as was the case with the sums calculated in multiplication).

For example, consider the case where `x` is infinite and `y` is finite. By the `:logic` definition of `ob^`, we multiply `x` by `(ob^ x (1- y))`. But notice that `(ob* a b)` traverses `b`, adding the first exponent of `a` to every exponent of `b`

```
(defun o^1 (x y)
  (cond ((finp y) (expt x y))
        ((equal (o-first-expt y) 1)
                (omega-term (o-first-coeff y) (expt x (o-rst y))))
        (t (let ((fe-y  (o-first-expt y))
                 (fco-y (o-first-coeff y))
                 (z     (o^1 x (o-rst y))))
             (omega-term (make-ord (o- fe-y 1) fco-y (o-first-expt z))
                         (o-first-coeff z))))))

(defun o^2 (x y)
  (cond ((zp y) 1)
        ((= y 1) x)
        (t (o* (omega-term (o* (o-first-expt x) (1- y)) 1) x))))

(defun o^3h (a p n q)
  (cond ((zp q) p)
        (t (padd (o* (o^2 a q) p) (o^3h a p n (1- q)) n))))

(defun o^3 (x q)
  (cond ((= q 0) 1)
        ((= q 1) x)
        ((limitp x) (o^2 x q))
        (t (let ((z (limitpart x))
                 (n (len x)))
             (padd (o^2 z q) (o^3h z (natpart x) n (1- q)) n)))))

(defun o^4 (x y)
  (o* (omega-term (o* (o-first-expt x) (limitpart y)) 1)
      (o^3 x (natpart y))))

(defun ob^ (x y)
  (mbe :logic (let ((fe-x  (o-first-expt x))
                    (fe-y  (o-first-expt y))
                    (fco-y (o-first-coeff y)))
                (cond ((or (and (finp y) (not (posp y))) (equal x 1)) 1)
                      ((equal x 0) 0)
                      ((finp y) (o* x (ob^ x (1- y))))
                      ((finp x)
                       (if (equal fe-y 1)
                           (o* (omega-term fco-y 1) (ob^ x (o-rst y)))
                         (o* (omega-term (omega-term (o- fe-y 1) fco-y) 1)
                             (ob^ x (o-rst y)))))
                      (t (o* (omega-term (o* fe-x (first-term y)) 1)
                             (ob^ x (o-rst y))))))
       :exec (cond ((or (equal y 0) (equal x 1)) 1)
                   ((equal x 0)                  0)
                   ((and (finp x) (finp y))      (expt x y))
                   ((finp x)                     (o^1 x y))
                   ((finp y)                     (o^3 x y))
                   (t                            (o^4 x y)))))
```


Figure 3: Ordinal Exponentiation

until we get to the natural number at the end of `b`. Now suppose `x` is a limit ordinal. That is, suppose that the natural number at the end of `x` is 0. Then multiplying `x` by itself `y` times simply amounts to adding the first exponent of `x` to every exponent in `x`, `y`-1 times. By the left distributive property of ordinal multiplication over ordinal addition, we can pull out the `y`-1 copies of `(o-first-expt x)`. This is what `o^2` does.

When `x` is not a limit ordinal, but `y` is still finite, things get a little more involved. Since `x` is not a limit ordinal, `(ob^ x (1- y))` is not a limit ordinal. Hence, when we multiply `x` by `(ob^ x (1- y))`, we eventually need to deal with the case where we are multiplying an infinite ordinal and a natural number. When this happens, we leave the first exponent of `x` the same, multiply the first coefficient of `x` by the natural number, and append the rest of `x`. Thus, every time we multiply by `x`, we multiply `x` by the `limitpart` of `(ob^ x (1- y))` (like we do in `o^2`), and then add the product of `x` and the `natpart` of `(ob^ x (1- y))`. This is what `o^3` does.

The case where `x` is a positive natural number and `y` is an infinite ordinal is straightforward and is handled by `o^1`.

Finally, when we are raising an infinite successor ordinal to an infinite successor ordinal power, we take advantage of the following property: $\alpha^{\beta+\gamma} = \alpha^{\beta}\alpha^{\gamma}$. We break up `y` into the sum of a limit ordinal and a natural number. The correctness of `o^4` is then apparent from the definition of multiplication.

Using several results about `len` and `sz` applied to `(ob* a b)` and `(ob^ a b)`, we can show that the complexity of the `:logic` definition of exponentiation, when using the efficient `:exec` version of `ob*` is as follows.

$$O\left(\begin{array}{l}\texttt{(natpart b)}\cdot\texttt{(len a)}\cdot\texttt{(len b)}\cdot\\ [\texttt{(sz (o-first-expt (o-first-expt a)))}\cdot\texttt{(len b)}+\texttt{(sz a)}+\texttt{(sz b)}]\\ +\texttt{(natpart b)}^2[\texttt{(sz (o-first-expt a))}\cdot\texttt{(len a)}+\texttt{(sz a)}]\end{array}\right)$$

On the other hand, Manolios and Vroon [36] show that the complexity of the `:exec` version of `ob^` is

$$O\left(\begin{array}{l}\texttt{(natpart b)}[\texttt{(len a)}\cdot\texttt{(len b)}+\texttt{(len (o-first-expt a))}\cdot\texttt{(len a)}\\ +\texttt{(sz a)}]+\texttt{(sz (o-first-expt (o-first-expt a)))}\cdot\texttt{(len b)}+\texttt{(sz b)}\end{array}\right)$$

This is a significant improvement in performance.

Thus by using `mbe`, each operation can have two definitions associated with it. The `:logic` definitions tend to be simple, easy to reason about, and ideally suited for symbolic reasoning, whereas the `:exec` definitions tend to be algorithmically more efficient, but also more complicated. ACL2 allows us to use the `:logic` definitions for reasoning and the `:exec` definitions for computation, all in a seamless, provably sound way.

4 Optimizing for Proof

In the applications of `defexec` presented above, 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. Other situations exist in which the more natural definition is efficient, but needs to be modified in order to facilitate logical reasoning. In such situations, `mbe` can be used to preserve the use of the natural definition for execution purposes.

Section 4.1 gives an example showing how an efficient executable partial function can be extended to a less efficient total function that has nicer properties in the logic. Because of `mbe`, the logical definition can be executed using the efficient (partial) definition.

Section 4.2 then shows how to admit a so-called "reflexive" function by adding necessary termination tests to the logical definition. For a more elaborate example, see the discussion of a linear path-finding algorithm in [23].

Section 4.3 presents a unification algorithm with supporting functions whose termination proofs rely on guarding each recursive call of the logical definition with an expensive check, for example that a graph is acyclic. Such examples thus add extra tests to the logic definition in order to prove termination, as for reflexive functions mentioned above, except that the motivation is partiality rather than reflexivity.

Finally, we return in Section 4.4 to the theme of executable partial functions by giving a generic method for admitting those that are tail-recursive.

4.1 Normalized Association Lists

Our first 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 which 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 non-trivial. Here we provide an overview of the steps involved in the definitions.

In order to get the last three properties 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). Further, in order 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` which 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 [31]. 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 which 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 function because we wanted the theorems 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 as follows, to achieve both the algebraic properties and efficient execution:

```
(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 based on 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 which 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. By 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, *i.e.*, 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, that 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)
```

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

```
x))
```

Note of course, that the example above is merely pedagogical; the `:exec` code for `weird-identity` could have simply been `x`, as in fact the proof obligation above demonstrates. However, non-trivial reflexive definitions arise in practice. The TR describes such a case study, namely a sophisticated implementation of a unification algorithm using term dags [48]. Furthermore, authors Greve and Wilding describe the use of the same approach in an efficient implementation of a path finding algorithm in a graph [23].

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 Unification on Term Dags

The next example shows a verified ACL2 implementation of a syntactic first-order unification algorithm that uses directed acyclic graphs to represent terms. There exist several previously published unification algorithms verifications (see [43, 47], for example) but none of them use directed acyclic graphs. In this implementation, some of the auxiliary functions involved in the algorithm need computationally expensive conditions in their logical definitions, which can be safely removed for execution on their intended domains (by means of `defexec` and `mbe`).

We start with a brief overview of unification. An *equation* is a pair of first-order terms, denoted as $t_1 \approx t_2$, and a *system of equations* is a finite set of equations. A substitution σ is a *solution* of $t_1 \approx t_2$ if $\sigma(t_1) = \sigma(t_2)$ and it is a solution of a system of equations S if it is a solution of every equation in S. Given two substitutions σ and δ, we say that σ is more general than δ if there exists a substitution γ such that $\delta = \gamma \circ \sigma$ (here $\circ$ denotes functional composition). A *most general solution* of a system S is a solution of S that it is more general than any other solution. We say that two terms t_1 and t_2 are *unifiable* if there exists a solution (called a *unifier*) of the system $\{t_1 \approx t_2\}$. A *most general unifier* (*mgu* in the sequel) of t_1 and t_2 is a most general solution of that system. A *unification algorithm* is an algorithm that decides whether two given terms are unifiable, and in that case it returns a most general unifier. A complete description of the theory of unification can be found in [4].

The unification algorithm implemented is based essentially on the relation $\Rightarrow_u$ given by the transformation rules in Figure 4 (known as the *Martelli--Montanari transformation system*). This set of rules acts on pairs of systems of equations of the form $S; U$ (what we call a *unification problem*). Intuitively, the system S can be seen as a set of equations still to be solved, and the sys-

Delete:	$\{t \approx t\} \cup R; U \Rightarrow_u R; U$
Occur-check:	$\{x \approx t\} \cup R; U \Rightarrow_u \bot$ if $x \in \mathcal{V}(t)$ and $x \neq t$
Eliminate:	$\{x \approx t\} \cup R; U \Rightarrow_u \theta(R); \{x \approx t\} \cup \theta(U)$ if $x \in X$, $x \notin \mathcal{V}(t)$ and $\theta = \{x \mapsto t\}$
Decompose:	$\{f(s_1, ..., s_n) \approx f(t_1, ..., t_n)\} \cup R; U \Rightarrow_u \{s_1 \approx t_1, ..., s_n \approx t_n\} \cup R; U$
Clash:	$\{f(s_1, ..., s_n) \approx g(t_1, ..., t_m)\} \cup R; U \Rightarrow_u \bot$ if $n \neq m$ or $f \neq g$
Orient:	$\{t \approx x\} \cup R; U \Rightarrow_u \{x \approx t\} \cup R; U$ if $x \in X$, $t \notin X$

Figure 4: Martelli–Montanari transformation system

tem U as a (partially) computed unifier[4]. The symbol $\bot$ represents unification failure. Beginning with the unification problem $S; \emptyset$, these rules can be applied iteratively (in a "don't care" nondeterministic manner), until either $\bot$ or a pair of systems of the form $\emptyset; U$ is obtained. It can be proved that this process must terminate and that S has a solution if and only if $\bot$ is not derived; in that case U is a most general solution of S. Thus, a unification algorithm can be designed by choosing a suitable data structure to represent first-order terms, and a strategy to apply the rules, beginning with the unification problem $\{t_1 \approx t_2\}; \emptyset$ (where t_1 and t_2 are the input terms).

In order to implement a unification algorithm in ACL2, a naive representation for terms could be to use lists denoting the terms in prefix notation (except variables, represented by atomic objects). For example, the term $f(x, f(g(x), f(x, y)))$ is represented by the list `'(f x (f (g x) (f x y)))`. Nevertheless, with this prefix representation a unification algorithm may be inefficient in some situations. Note that every application of the **Eliminate** rule to a unification problem represented in prefix form needs to reconstruct the instantiated systems of equations, and the problem can be even worse if the system contains repeated variables. The standard approach to deal with this problem is to use directed acyclic graphs (*dags* in the following) to represent terms. For example, the graph in Figure 5 represents the equation $f(x, f(g(x), f(x, y))) \approx f(g(y), f(u, f(g(y), y)))$. Nodes are labeled with function and variable symbols, and outgoing edges connect every non-variable node with dags representing its immediate subterms. We can naturally identify the root node of a term dag with the whole term. Note also that there is a certain amount of *structure sharing*, at least for the repeated variables:

To implement a unification algorithm with this term dag representation, the main idea is never to build new terms but only to update pointers destructively. In particular, the **Eliminate** rule can be implemented by adding a pointer linking the variable with the term to which this variable is bound; in that way

[4]We will identify a system of equations of the form $\{x_1 \approx t_1, \ldots, x_n \approx t_n\}$, where the x_i are variables, with the substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. If none of the x_i appear in any of the t_j, we say that the system is in *solved form*. Note that every system in solved form is an mgu of itself.

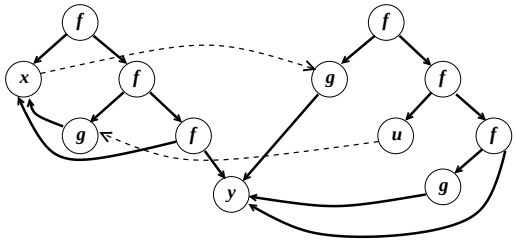

Figure 5: *Dag* representation of $f(x, f(g(x), f(x, y))) \approx f(g(y), f(u, f(g(y), y)))$

no reconstruction of the term is required in the application of a substitution. In the graph above, these pointers are represented by dashed arrows. The binding for a variable can be determined by following the pointers traversing the graph depth first, from left to right. In this case, the substitution represented is $\{x \mapsto g(y), u \mapsto g(g(y))\}$, which is an mgu of both terms.

4.3.1 A Formally Verified ACL2 Implementation

We now describe the ACL2 implementation of a unification algorithm on term dags, which is mainly based on the Pascal implementation given in section 4.8 of Baader and Nipkow's book [3], which in turn is based on the exposition by Corbin and Bidoit [16]. The main difference is that instead of using records with pointers, we use a single-threaded object. It should be noted that this implementation, although linear in space, may still require exponential time in some cases. For the sake of clarity, we will not introduce the technical improvements that can make the algorithm quadratic in time. The interested reader may consult [48], where the complete development of a formally verified quadratic unification algorithm is described.

The stobj used is a structure called `terms-dag` with only one field: an array called `dag`, whose size can be modified dynamically. This array is used to store the unification problem in dag form:

```
(defstobj terms-dag
  (dag :type (array t (0)) :resizable t))
```

Once the `terms-dag` stobj is defined, the expressions `(dagi` i `terms-dag)` and `(update-dagi` i v `terms-dag)` respectively access and update (with value v) the i-th cell of the `dag` array. These operations are done in constant time and the update is destructive (at the price of syntactic restrictions on the use of `terms-dag`). Each node in the graph is represented by a cell in the `dag` array of the stobj. Therefore, a node in the graph can be identified with an array index. Each cell stores the label and the successors of each node, in the following way:

- If node i represents an unbound variable x, then the i-th cell of the array contains an ordered pair of the form `(`x `. t)`.
- If node i represents an instantiated variable, then the i-th cell of the array contains an index n pointing to the root node of the term to which the variable is bound.
- If node i is the root node of a non-variable term $f(t_1, \ldots, t_n)$, then the i-th cell of the array contains an ordered pair of the form `(`f `.` l`)`, where l is the list of the indices corresponding to the root nodes of $t_1, \ldots, t_n$.

For example, if we store the term $equ(f(x, f(g(x), f(x, y))), f(g(y), f(u, f(g(y), y))))$ in the stobj, the cells of the `dag` array are:

0	1	2	3	4	5	6	7	8
(EQU . (1 9))	(F . (2 3))	(X . T)	(F . (4 6))	(G . (5))	2	(F . (7 8))	2	(Y . T)
(F . (10 12))	(G . (11))	8	(F . (13 14))	(U . T)	(F . (15 17))	(G . (16))	8	8
9	10	11	12	13	14	15	16	17

We can naturally identify an array index with the term represented by the subgraph whose root node is stored in the corresponding array cell. Taking advantage of this idea, we define a function `dag-transform-mm-st` in Figure 6 that applies one step of the transformation relation $\Rightarrow_u$ to a unification problem in dag form.

Let us explain the behavior of `dag-transform-mm-st`, although in the next subsection we will describe some of its auxiliary functions. In addition to the stobj, the function receives as input a non-empty system `S` of equations to be unified and a substitution `U` partially computed. The key point here is that `S` and `U` *only contain indices* pointing to the terms stored in `terms-dag`. In particular, `S` is a list of pairs of indices, and `U` is a list of pairs of the form `(`x `.` n`)` where x is a variable symbol and n is an index pointing to the node for which the variable is bound. We say that `S` is an *indices system* and `U` an *indices substitution*, and both together with the contents of `terms-dag` form what we call a *dag unification problem*. Depending on the pair of terms pointed to by the indices of the first equation of `S`, one of the rules of $\Rightarrow_u$ is applied. The function returns a multivalue `(mv ...)` with the following components, obtained as a result of the application of one step of transformation: the resulting indices system of equations to be solved, the resulting indices substitution, a boolean (if $\bot$ is obtained, this value is `nil`, else `t`) and the stobj `terms-dag`. Note that only when **Eliminate** is applied is the stobj updated, causing the corresponding variable to point to the corresponding term.

With `dag-transform-mm-st` as its main component, we can define the unification algorithm. In short, the main function, called `dag-mgu` and omitted here, receives as input two terms in prefix form and uses `terms-dag` as a local stobj; after storing these terms as directed acyclic graphs in the stobj (previously resizing the `dag` array properly), it iteratively applies the function `dag-transform-mm-st` until either non-unifiability is detected or there are no more equations to be solved. In this last case, the returned substitution (in

```
(defun dag-transform-mm-st (S U terms-dag)
  (declare (xargs :stobjs terms-dag))
  (let* ((equ (car S)) (R (cdr S))
         (t1 (dag-deref-st (car equ) terms-dag))
         (t2 (dag-deref-st (cdr equ) terms-dag))
         (p1 (dagi t1 terms-dag))
         (p2 (dagi t2 terms-dag)))
    (cond
     ((= t1 t2) (mv R U t terms-dag))                    ;DELETE
     ((dag-variable-p p1)
      (if (occur-check-st t1 t2 terms-dag)
          (mv nil nil nil terms-dag)                     ;OCCUR-CHECK
        (let ((terms-dag (update-dagi t1 t2 terms-dag)))
;ELIMINATE
          (mv R (cons (cons (dag-symbol p1) t2) U) t terms-dag))))
     ((dag-variable-p p2)
      (mv (cons (cons t2 t1) R) U t terms-dag))         ;ORIENT
     ((not (eql (dag-symbol p1) (dag-symbol p2)))
      (mv nil nil nil terms-dag))                        ;CLASH1
     (t (mv-let (pair-args bool)
                (pair-args (dag-args p1) (dag-args p2))
                (if bool                                 ;DECOMPOSE
                    (mv (append pair-args R) U t terms-dag)
                  (mv nil nil nil terms-dag)))))))      ;CLASH2
```

Figure 6: One step of transformation for unification

prefix form) is built from the final contents of `dag`, following the pointers of the instantiated variables.

Thus, the function `dag-mgu` returns two values: the first output value is a boolean indicating whether the terms are unifiable or not and, in case of unifiability, the second is the mgu. For example `(dag-mgu '(f x (f (g x) (f x y))) '(f (g y) (f u (f (g y) y))))` returns the multivalue `(t ((u . (g (g y))) (x . (g y))))` and `(dag-mgu '(f y x) '(f (k x) y))` returns `(nil nil)`.

The guard of the function `dag-mgu` is very simple, and it only requires that the two input terms have to be ACL2 objects representing well-formed terms in prefix form. This well-formedness property is defined by a function called `term-p`. The following theorems are proved in ACL2, showing that the function `dag-mgu` computes the most general unifier of two terms if and only if the terms are unifiable. Here the function `instance` implements the application of a substitution to a term, and `subs-subst` implements the "more general than" relation between substitutions:

```
(defthm dag-mgu-completeness
  (implies (and (term-p t1) (term-p t2)
                (equal (instance t1 sigma) (instance t2 sigma)))
           (first (dag-mgu t1 t2))))


(defthm dag-mgu-soundness
  (implies (and (term-p t1) (term-p t2)
                (first (dag-mgu t1 t2)))
           (equal (instance t1 (second (dag-mgu t1 t2)))
                  (instance t2 (second (dag-mgu t1 t2))))))


(defthm dag-mgu-most-general-solution
  (implies (and (term-p t1) (term-p t2)
                (equal (instance t1 sigma) (instance t2 sigma)))
           (subs-subst (second (dag-mgu t1 t2)) sigma)))
```

Note that the theory used to state the properties deals with terms represented in prefix notation. Also the input and the output of the function `dag-mgu` are terms and substitutions in prefix notation. But it has to be emphasized that internally, the main process is carried out on term dags. A description of the formal proof of these theorems in ACL2 can be found in [48].

4.3.2 Attaching Executable Functions

Although the algorithm described above terminates for every pair of ACL2 objects representing terms in prefix form (*i.e.*, on the intended domain described by its guard), some of the auxiliary functions used by the algorithm are not terminating in general. Consider, for instance the process of *dereferencing* the

```
(defexec dag-deref-st (h terms-dag)
  (declare (xargs :stobjs terms-dag
                  :guard (and (natp h)
                              (< h (dag-length terms-dag))
                              (term-graph-p-st terms-dag)
                              (dag-p-st terms-dag)) ...) ...)
  (mbe :logic
       (if (dag-p-st terms-dag)
           (let ((p (dagi h terms-dag)))
             (if (integerp p) (dag-deref-st p terms-dag) h))
         'undef)
       :exec
       (let ((p (dagi h terms-dag)))
         (if (integerp p) (dag-deref-st p terms-dag) h))))
```

Figure 7: Dereferencing

pair of indices corresponding to the selected equation in the transformation step (Figure 6). This process is implemented by the function `dag-deref-st`, where `(dag-deref-st` i `terms-dag)` is the result of following a chain of instantiated variable nodes in the graph stored in `terms-dag`, starting in node i, until an unbound variable node or a non-variable node is found. Clearly, there are situations in which this process may not terminate, as a consequence of the possibility of the existence of cycles (the simplest case is that the i-th cell could contain the number i). As discussed at the end of Subsection 1.2, a non-terminating process cannot be directly defined as a function in ACL2.

Consequently, in order to get the function admitted in the ACL2 logic we must explicitly introduce in its logical definition a condition that ensures its termination. This condition is implemented by a function `dag-p-st` checking whether the graph stored in the stobj is acyclic. Roughly speaking, the function `dag-p-st` does a depth-first search for cycles in a similar way to the linear pathfinding algorithm presented in the previous section. Having defined the function `dag-p-st`, we may now define the function `dag-deref-st` by means of `defexec`, as shown in Figure 7.

Note that the logical definition of the function includes the condition `(dag-p-st terms-dag)`, needed to be accepted by the ACL2 principle of definition. This test is computationally expensive and, even worse, it would have to be evaluated *in every recursive call* of the function `dag-deref-st`, making the logical definition impractical for execution. The use of `mbe` guarantees that once the guards are verified, the `:exec` body, without the expensive acyclicity test, can be safely used for execution when the function is called on arguments

satisfying its guard.

The guard of the function ensures that the cells of the `dag` array have contents of one of the three expected types described in the previous subsection (this condition is implemented by the function `term-graph-p-st`). The guard also requires the graph stored in the stobj to be acyclic. Since the graph is not updated during the process performed by `dag-deref-st`, and the guard trivially implies the condition `(dag-p-st terms-dag)` removed in the `:exec` body, then the guard verification of the function `dag-deref-st` is straightforward.

The call to `defexec` also generates proof obligations that ensure the termination of the `:exec` body on the domain specified by its guard. In this case, this proof obligation is similar to the one generated to show termination of the logical definition. This termination proof is not trivial and it is necessary to provide a measure (omitted here) that decreases in every recursive call. Roughly speaking, this measure counts the number of reachable nodes from the given node `h`.

There are two other auxiliary functions used by the unification algorithm that traverse the graph stored in the stobj. One of them is the function `occur-check-st` that decides if a given variable occurs in the term pointed by a given index. The other is the function that in the final step of the algorithm reconstructs the mgu in prefix form, following the pointers of the instantiated variables. Both functions are defined using `defexec` in a similar way to the above definition of `dag-deref-st`, including the acyclicity check in the logical definition and removing it in the executable body.

But non-termination situations can arise even if no traversal of the graph is performed. Consider a dag unification problem having `'((0 . 1))` as the (indices) system of equations to be solved, and such that `(dagi 0 terms-dag)`=`'(f . (0))` and `(dagi 1 terms-dag)`=`'(f . (1))`. In this case, the function `dag--transform-mm-st` of Figure 6 would apply the **Decompose** rule, obtaining the same dag unification problem, and thus an iterative application of that function would not terminate on this example.

Nevertheless, it is possible to define conditions on the dag unification problem that prevent these non-terminating situations. Essentially, these conditions ensure that the initial dag unification problem represents a corresponding unification problem represented in prefix form. These considerations lead to the definition of a function `dag-solve-system-st` (Figure 8) that iteratively applies `dag-transform-mm-st` until either non-unifiability is detected or there are no more equations to be solved.

The condition `(well-formed-upl-st S U terms-dag)` checks the well-formedness properties of the input dag unification problem and it is sufficient to prove termination of the function. The definition of `well-formed-upl-st`, omitted here, includes the `term-graph-p-st` and the `dag-p-st` conditions on the `term-dag` stobj. It also ensures that the variables of the graph are shared, and that `S` and `U` are, respectively, a well-formed indices system and an indices substitution. As in the previous cases, this condition is computationally expensive and makes the logical definition of the function impractical for execution. Fortunately, we can safely remove this expensive well-formedness condition in

```
(defexec dag-solve-system-st (S U bool terms-dag)
  (declare (xargs :stobjs terms-dag
                  :guard (well-formed-upl-st S U terms-dag)
...)...)
  (mbe :logic
       (if (well-formed-upl-st S U terms-dag)
           (if (or (not bool) (endp S))
               (mv S U bool terms-dag)
             (mv-let (S1 U1 bool1 terms-dag)
                     (dag-transform-mm-st S U terms-dag)
                     (dag-solve-system-st S1 U1 bool1
terms-dag)))
         (mv nil nil nil terms-dag))
       :exec
       (if (or (not bool) (endp S))
           (mv S U bool terms-dag)
         (mv-let (S1 U1 bool1 terms-dag)
                 (dag-transform-mm-st S U terms-dag)
                 (dag-solve-system-st S1 U1 bool1 terms-dag)))))
```

Figure 8: Solving dag unification problems

the executable body, by means of `mbe`.

In this case the guard verification is more difficult than in the previous case. The reason is that one step of transformation could apply the **Eliminate** rule and change the contents of the stobj. Thus, it has to be proved that the well-formedness condition of the dag unification problem is preserved by the rules of transformation, as established by the following theorem. Note that in particular, this theorem requires a proof that the updated graph obtained after applying the **Eliminate** rule is still acyclic:

```
(defthm well-formed-upl-st-preserved-by-dag-transform-mm-st
  (implies (and (well-formed-upl-st S U terms-dag) (consp S))
           (mv-let (S1 U1 bool1 terms-dag)
                   (dag-transform-mm-st S U terms-dag)
                   (well-formed-upl-st S1 U1 terms-dag)))
```

The termination proofs for both the logical definition and the executable body of `dag-solve-system-st` are the same: they can be achieved by explicitly providing a measure (omitted here) on the dag unification problem, that decreases with respect to a lexicographic well-founded relation. Essentially, the measure constructs the associated unification problem in prefix form and applies a measure that was previously proved suitable for the prefix version of the algorithm. See [48] for details.

Finally, it is worth pointing out that although some guards of the auxiliary functions presented here are computationally expensive (since they contain the acyclicity test), the guard of the main function `dag-mgu` is very simple and only checks that the two input ACL2 objects are well-formed terms in prefix form. So only this initial check has to be done: the guard verification mechanism ensures that subsequent calls of the auxiliary functions are on arguments satisfying their guards, so no subsequent guard checking is done at run time and the tests for cycles are never evaluated.

4.4 Executable Tail-Recursive Partial Functions

As a final application of `mbe` and `defexec` in optimizing natural executable definitions for logical reasoning, we will consider its use in efficiently executing tail-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 non-integral 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 non-terminating tail-recursive equations can arise in non-trivial contexts, for example in formalizing microprocessor interpreters or low-level procedural programming languages [41]. 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 [35] 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` which makes it possible to introduce equations such as `trfact` 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` [44] 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, non-looping 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.

But this paper has highlighted 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 runtime tests that can be shown to be unnecessary once the properties of the newly defined concept have been established. Using the new feature we showed how such runtime tests can be eliminated after the fact.

As another highlighted use of `mbe` we showed 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 showed 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.

Acknowledgements

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 of the corresponding paper for helpful expository suggestions.

References

[1] S. Allen, R. Constable, D. Howe, and W. Aitken. The semantics of reflected proof. In J. Mitchell, editor, *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105. IEEE Computer Society Press, 1990.

[2] K. R. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs*. Springer–Verlag, 1991.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

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

[6] R. S. Boyer and J S. Moore. Proving theorems about pure Lisp functions. *Journal of the ACM*, 22(1):129–144, 1975.

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

[8] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

[9] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, second edition, 1997.

[10] R. S. Boyer and J S. Moore. Single-threaded objects in ACL2. In S.Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages : 4th International Symposium, PADL 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 9–27. Springer-Verlag, 2002. See URL `http://www.cs.utexas.edu/users/moore/-publications/stobj/main.ps.Z`.

[11] B. Brock and W. A. Hunt, Jr. Formal analysis of the Motorola CAP DSP. In Mike Hinchey and Jonathan Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 81–116. Springer-Verlag, 1999.

[12] G. Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 46:481–512, 1895.

[13] G. Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 49:207–246, 1897.

[14] G. Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover Publications, Inc., 1955. Translated by Philip E. B. Jourdain.

[15] A. Church and S. C. Kleene. Formal definitions in the theory of ordinal numbers. *Fundamenta Mathematicae*, 28:11–21, 1937.

[16] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. *Information Processing*, 83:909–914, 1983.

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

[18] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001. See URL `http://www.csl.sri.com/users/rushby/papers/attach.pdf`.

[19] J. Davis. Finite set theory based on fully ordered lists. In M. Kaufmann and J S. Moore, editors, *Proceedings of the 5th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004. See URL `http://-www.cs.utexas.edu/users/moore/acl2/workshop-2004/`.

[20] K. Devlin. *The Joy of Sets: Fundamentals of Contemporary Set Theory*. Springer-Verlag, second edition, 1992.

[21] G. Gentzen. Die widerspruchsfreiheit der reinen zahlentheorie. *Mathematische Annalen*, 112:493–565, 1936. English translation in Szabo, M.E. (ed), The Collected Works of Gerhard Gentzen, 132-213, North Holland, 1969.

[22] M. Gordon, J. Hurd, and K. Slind. Executing the formal semantics of the Accellera property specification language by mechanised theorem proving. In D. Geist, editor, *Proceedings of the 12th International Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, 2003.

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

[24] D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 113–136. Kluwer Academic Press, 2000.

[25] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995. See URL `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.html`.

[26] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1994.

[27] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[28] M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997. See URL: `http://www.cs.utexas.edu/users/moore/-publications/acl2-papers.html#Foundations`.

[29] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[30] M Kaufmann and J S. Moore. ACL2 Home Page, 2006. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.

[31] M. Kaufmann and R. Sumners. Efficient rewriting of data structures in ACL2. In D. Borrione, M. Kaufmann, and J S. Moore, editors, *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications*, 2002. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/`.

[32] K. Kunen. *Set Theory - An Introduction to Independence Proofs*, volume 102 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1980.

[33] H. Liu and J S. Moore. Executable JVM model for analytical reasoning: A study. In *IVME '03: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 15–23. ACM Press, 2003.

[34] P. Manolios and M. Kaufmann. Adding a total order to ACL2. In D. Borrione, M. Kaufmann, and J S. Moore, editors, *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications*, 2002. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/`.

[35] P. Manolios and J S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[36] P. Manolios and D. Vroon. Algorithms for ordinal arithmetic. In Franz Baader, editor, *19th International Conference on Automated Deduction – CADE-19*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 243–257. Springer–Verlag, 2003.

[37] P. Manolios and D. Vroon. Ordinal arithmetic in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J S. Moore, editors, *Proceedings of the 4th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`.

[38] P. Manolios and D. Vroon. Integrating reasoning about ordinal arithmetic into ACL2. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design – FMCAD 2004*, volume 3312 of *Lecture Notes in Computer Science*. Springer–Verlag, 2004.

[39] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, pages 1–37, 2006.

[40] J. Matthews and D. Vroon. Partial clock functions in ACL2. In M. Kaufmann and J S. Moore, editors, *Proceedings of the 5th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/`.

[41] J S. Moore. Inductive assertions and operational semantics. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods – CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 289–303. Springer-Verlag, 2003.

[42] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[43] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5(2):143–169, 1985.

[44] S. Ray. Attaching efficient executability to partial functions in ACL2. In M. Kaufmann and J S. Moore, editors, *Proceedings of the 5th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/`.

[45] S. Ray and R. Sumners. Verification of an in-place quicksort in ACL2. In D. Borrione, M. Kaufmann, and J S. Moore, editors, *Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications*, 2002. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/`.

[46] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[47] J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martín. Mechanical verification of a rule-based unification algorithm in the Boyer-Moore theorem prover. In *AGP'99 Joint Conference on Declarative Programming*, pages 289–304, 1999.

[48] J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martín. Formal correctness of a quadratic unification algorithm. *Journal of Automated Reasoning*, pages 1–26, 2006. Accepted for publication, published Online First.

[49] D. Russinoff, M. Kaufmann, E. Smith, and R. Sumners. Formal verification of floating-point rtl at amd using the acl2 theorem prover. In Nikolai Simonov, editor, *Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, July 2005. See URL `http://sab.sscc.ru/imacs2005/papers/T2-I-94-1021.pdf`.

[50] D. M. Russinoff and A. Flatau. RTL verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232. Kluwer Academic Publishers, 2000.

[51] K. Schütte. *Proof Theory*. Springer–Verlag, 1977. Translation from the German by J. N. Crossley. The book is a completely rewritten version of *Beweistheorie*, Springer–Verlag, 1960.

[52] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.

[53] N. Shankar. Efficiently executing PVS, 1999. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November, 1999.

[54] G. L. Steele, Jr. *Common Lisp The Language*. Digital Press, second edition, 1990.

[55] R. Sumners. Correctness proof of a BDD manager in the context of satisfiability checking. In M. Kaufmann and J S. Moore, editors, *Proceedings of the 2nd International Workshop on the ACL2 Theorem Prover and Its Applications*, 2000. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/`.

[56] M. Sustik. Proof of Dickson's lemma using the ACL2 theorem prover via an explicit ordinal mapping. In W. A. Hunt, Jr., M. Kaufmann, and J S. Moore, editors, *Proceedings of the 4th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`.

[57] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, second edition, 2000.

[58] R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence Journal*, 13(1):133–170, 1980.