Integrating Reasoning about Ordinal Arithmetic into ACL2

Panagiotis Manolios and Daron Vroon

Georgia Institute of Technology, College of Computing
801 Atlantic Drive, Atlanta, Georgia, 30332, USA,
{manolios,vroon}@cc.gatech.edu
http://www.cc.gatech.edu/~{manolios,vroon}

Abstract. Termination poses one of the main challenges for mechanically verifying infinite state systems. In this paper, we develop a powerful and extensible framework based on the ordinals for reasoning about termination in a general purpose programming language. We have incorporated our work into the ACL2 theorem proving system, thereby greatly extending its ability to automatically reason about termination. The resulting technology has been adopted into the newly released ACL2 version 2.8. We discuss the creation of this technology and present two case studies illustrating its effectiveness.

1 Introduction

Termination arguments play a critical role in the design and verification of computing systems. We are interested in providing support for reasoning about the termination of arbitrary programs written in actual programming languages. To that end, we develop a powerful and extensible framework —based on our previous work on ordinal arithmetic [15, 16]— for reasoning about termination in the ACL2 theorem proving system [11, 12].

Our choice of ACL2 for this project was based on two criteria. Since termination is unsolvable, we wanted a system with theorem proving support and in which termination plays a key role. ACL2 meets both of these criteria. It is a powerful theorem proving system which has been applied to several large-scale industrial projects by companies such as AMD, IBM, Motorola, Rockwell Collins, and Union Switch and Signal. Termination is a centerpiece of reasoning in ACL2, as all functions admitted using the definitional principal must be proved to terminate before ACL2 will admit them. This is accomplished by providing a measure function that maps the function parameters into the ordinals and showing that recursive calls decrease according to the measure.

In previous work we developed and verified algorithms for ordinal arithmetic. In this paper, we discuss how we integrated this work with ACL2 version 2.8 [12] to create a powerful, extensible, general framework for reasoning about termination. It is extensible in that new techniques and theorems can be added to ACL2 to enhance its ability to automatically reason about termination, *e.g.*, we proved the well-foundedness of the lexicographic ordering over lists of natural numbers, which enables ACL2 to use measure functions that instead of mapping into the ordinals, map into lists of natural numbers.

The generality of our approach is a byproduct of our focus on providing support for proving arbitrary termination arguments, not on automatically proving termination for a decidable fragment of the termination problem. As an example of this generality, our work has been used to prove Dickson's Lemma [21], which plays a crucial role in proving the termination of Buchberger's algorithm for finding Gröbner bases of polynomial ideals (see Section 6.2).

Our work can also be used to reason about *reactive systems*, nonterminating systems that participate in ongoing interactions with their environments (*e.g.*, networking protocols and operating systems). In this context, termination arguments are used to prove liveness properties, which assert that a desired behavior of the system is not postponed forever. For example, imagine a complicated bus protocol operating on a system with a dynamic topology. Suppose this protocol partitions long messages into packets that are sent according to a priority-based scheme. The property stating that the protocol will never result in deadlock or livelock is a liveness property, which is proved with termination arguments.

While the current literature on termination is vast, most of the related work is focused on various restricted instances of the termination problem. For example, much of the current research on termination is aimed at providing termination proofs for Term Rewriting Systems (TRSs) [2, 1, 8]. Most of the remaining research is focused on developing algorithms and heuristics for the automatic generation of appropriate well-founded measure functions [14, 19, 7, 6, 5]. Since termination is an undecidable problem, this research focuses on solving decidable fragments and is generally presented in terms of toy languages that lack the full functionality of programming languages used in practice. The work we present here, on the other hand, focuses on automating the process of verifying termination arguments and not on guessing measure functions.

In sections 2 and 3, we give an overview of ACL2 and the ordinals. In Section 4 we briefly review our previous work on developing efficient algorithms for ordinal arithmetic. In Section 5, we present the changes we made to ACL2 in integrating our ordinal arithmetic work. Section 6 contains two case studies illustrating the use of our new technology. In Section 7, we discuss some lessons we learned in the course of this project. Finally, we cover the related work in more detail and conclude in Sections 8 and 9.

2 ACL2 Overview

ACL2 stands for "A Computational Logic for Applicative Common Lisp." It comprises a programming language, a first-order mathematical logic based on recursive functions, and a mechanical theorem prover for that logic.

The programming language can best be thought of as an applicative ("side-effect-free" or "pure functional") subset of Common Lisp. We assume basic knowledge of Common Lisp syntax. Because it is a programming language, ACL2 is executable, and execution can reach speeds comparable to programs written in C [22].

The logic of ACL2 is a first-order predicate calculus with equality, recursive function definitions, and mathematical induction. The primitive built-in functions are axiomatized. For example, one axiom is `(car (cons x y)) = x` and another is `x` $\neq$ `nil` $\Rightarrow$ `(if x y z) = y`. After axiomatizing the basic data types, a representation of the ordinals up to ε_0 is introduced along with an ordering relation, "less than," defined recursively on this definition. This forms the basis for the principle of mathematical induction in ACL2. To prove a conjecture by induction one must identify some ordinal-valued measure function. The induction principle then allows one to assume inductive instances of the conjecture being proved, provided the instance has a smaller measure according to the chosen measure function.

The ACL2 theorem prover is an example of the so-called Boyer-Moore school of inductive theorem proving [3, 4]. It is an integrated system of ad hoc proof techniques that include simplification, generalization, induction and many other techniques. Simplification is, however, the key technique and includes the use of evaluation, conditional rewrite rules, definitions (including recursive definitions), propositional calculus, a linear arithmetic decision procedure for the rationals, user-defined equivalence and congruence relations, user-defined and mechanically verified simplifiers, a user-extensible type system, forward chaining, an interactive loop for entering proof commands, and various means to control and monitor these features. See the ACL2 online user's manual for the full details [12].

ACL2 has been applied to a wide range of commercially interesting verification problems. We recommend visiting the ACL2 home page [12] and inspecting the links on Tours, Demo, Books and Papers, and for the most current work, The Workshops and Related Meetings. See especially [10].

3 Ordinal Overview

Ordinals can most easily be thought of as a transfinite extension of the natural numbers (0, 1, 2, ...). The first infinite ordinal is ω, which is the least ordinal that is greater than all the natural numbers. The next ordinal is $\omega + 1$, then $\omega + 2$, and so on until we reach $\omega + \omega$, which is denoted $\omega \cdot 2$. We can continue this process to get $\omega \cdot 2 + 1$, $\omega \cdot 2 + 2, \ldots$, $\omega \cdot 2 + \omega = \omega \cdot 3$, and so on. Eventually, we get to $\omega \cdot \omega$, which is denoted ω^2. Likewise, we can keep on counting to ω^3 and ω^4, and so on. The ordinal $\omega^{\omega^{\omega^{\cdots}}}$ is denoted ε_0 and is the ordinal on which termination reasoning in ACL2 is based.

Not surprisingly, set theorists define the ordinals in terms of sets. Each ordinal is simply the set of all ordinals less than itself. Thus, the ordinal denoted as 0 is the empty set, $\emptyset$. The ordinal corresponding to 1 is the set containing 0, $\{0\} = \{\emptyset\}$. The ordinal denoted by 2 is $\{0, 1\} = \{\emptyset, \{\emptyset\}\}$. The other natural numbers are defined similarly. The ordinal ω is just the set of all natural numbers, $\{0, 1, 2, \ldots\}$. Note that this implies that the "element of operator", $\in$, the proper subset operator, $\subset$, and the "less than" operator, $<$, are all equivalent on the ordinals.

For the purposes of termination, the most interesting property of ordinals is that they are *well-founded*. That is, there is no infinite sequence of ordinals, $(\alpha_1, \alpha_2, \ldots)$, such that $\alpha_i > \alpha_{i+1}$ for all $i > 0$. Thus, for any ordinal α, the pair $\langle \alpha, < \rangle$ is what is known as a *well-founded structure*. In fact, it is a *well-ordered structure*, which means α is well-founded under $<$ and for any ordinals $\beta, \gamma \in \alpha$, either $\beta < \gamma$, $\gamma < \beta$, or $\beta = \gamma$.

Proving termination means showing that a program has no infinite computations. This is generally done by assigning a value to each program state and showing that this value decreases with each step of the program. If these values range over a well-founded structure, then by definition, the values cannot decrease infinitely, which proves that the program terminates. Any well-founded structure can be extended to a well-ordered structure by making the relation total while preserving well-foundedness. The termination argument based on the original well-founded structure then directly transfers to this well-ordered extension. A basic result of set theory is that any well-ordered structure is order-isomorphic to a unique ordinal. In this sense, ordinals are the most general setting for termination arguments. This is why Turing says that for proving termination, "it is natural to give an ordinal number" [18].

3.1 Ordinal Arithmetic

Given a well-ordered structure, $\langle A, <_A \rangle$, we denote the unique ordinal that is isomorphic to this structure as $Ord(A, <_A)$.

Ordinal addition is defined as follows. Given two ordinals, α and β, $\alpha + \beta = Ord(A, <_A)$, where $A = (\{0\} \times \alpha) \cup (\{1\} \times \beta)$, and $<_A$ is the lexicographic ordering on A. Thus addition corresponds to starting with the elements of α and then tacking on the elements of β.

Ordinal multiplication is defined as follows. Given two ordinals, α and β, $\alpha \cdot \beta = Ord(A, <_A)$, where $A = \beta \times \alpha$ and $<_A$ is the standard lexicographic ordering. In other words, we create β copies of α.

Ordinal exponentiation is defined by transfinite recursion. Given an ordinal, $\alpha \neq 0$, $\alpha^0 = 1$, $\alpha^{\beta+1} = \alpha^\beta \cdot \alpha$, and $\alpha^\beta = \bigcup_{\gamma<\beta} \alpha^\gamma$. For the case where $\alpha = 0$, we have $0^0 = 1$, and $0^\beta = 0$ for all ordinals $\beta \neq 0$.

Although the finite ordinals correspond to the natural numbers and therefore enjoy all the algebraic properties we expect, the infinite ordinals behave differently. For example, addition and multiplication are not commutative: $2 + \omega = \omega < \omega + 2$ and $2 \cdot \omega = \omega < \omega \cdot 2$. Also, multiplication only distributes from the left. That is, $\alpha \cdot (\beta + \gamma) = (\alpha \cdot \beta) + (\alpha \cdot \gamma)$, but it is not the case that $(\beta + \gamma) \cdot \alpha = (\beta \cdot \alpha) + (\gamma \cdot \alpha)$. This makes reasoning about ordinal arithmetic more interesting.

4 Algorithms for Ordinal Arithmetic

In previous work we developed algorithms for ordinal arithmetic based on a notation for the ordinals up to ε_0. We developed efficient algorithms on succinct notations that are now used by ACL2 to reason about ordinal expressions in the ground (variable-free) case. We present here a brief overview of this work.

4.1 Ordinal Notations

The basis for the ordinal notation used in ACL2, for versions prior to version 2.8, is the following variant of Cantor's Normal Form Theorem.

Theorem 1. *For every ordinal $\alpha \in \varepsilon_0$, there are unique $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_n > 0$ such that $\alpha > \alpha_1$ and $\alpha = \omega^{\alpha_1} + \cdots + \omega^{\alpha_n} + p$.*

Table 1 Ordinal Arithmetic Complexity Results

Function	Complexity
`(ocmp a b)`	$O(\min(\#\texttt{a}, \#\texttt{b}))$
`(o-p a b)`	$O(\#\texttt{a}(\log \#\texttt{a}))$
`(o+ a b)`	$O(\min(\#\texttt{a}, \lvert\texttt{a}\rvert \cdot \#(\texttt{o-first-expt b})))$
`(o- a b)`	$O(\min(\#\texttt{a}, \#\texttt{b}))$
`(o* a b)`	$O(\lvert(\texttt{o-first-expt a})\rvert\lvert\texttt{b}\rvert + \#(\texttt{o-first-expt a}) + \#\texttt{b})$
`(o^ a b)`	$O((\texttt{natpart b})[\lvert\texttt{a}\rvert\lvert\texttt{b}\rvert + \lvert(\texttt{o-first-expt a})\rvert\lvert\texttt{a}\rvert + \#\texttt{a}]$ $+\#(\texttt{o-first-expt (o-first-expt a)})\lvert\texttt{b}\rvert + \#\texttt{b})$

With this notation, the ACL2 representation of the ordinal $\alpha \in \varepsilon_0$, with normal form $\omega^{\alpha_1} + \cdots + \omega^{\alpha_n} + p$, is:

$$ACL2(\alpha) = (ACL2(\alpha_1)\ ACL2(\alpha_2)\ \ldots\ ACL2(\alpha_n)\ .\ p)$$

For example, $\omega + 2$ is `(1 . 2)` in ACL2 and $\omega^{\omega} + \omega^{\omega} + \omega^2 + 3$ is `((1 . 0) (1 . 0) 2 . 3)` in ACL2.

The basis for our ordinal notation, which is used in the newly released ACL2 version 2.8, is based on the following variant of Cantor's Normal Form Theorem. The idea is to collect terms with the same exponent using the left distributive property of ordinal multiplication over addition.

Theorem 2. (Cantor Normal Form) *For every ordinal $\alpha \in \varepsilon_0$, 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 $\alpha = \omega^{\alpha_1} x_1 + \cdots + \omega^{\alpha_n} x_n + p$.*

With this notation, the ACL2 representation of the ordinal α, with normal form $\omega^{\alpha_1} x_1 + \cdots + \omega^{\alpha_n} x_n + p$, is:

$$CNF(\alpha) = ((CNF(\alpha_1)\ .\ x_1)\ (CNF(\alpha_2)\ .\ x_2)\ \ldots\ (CNF(\alpha_n)\ .\ x_n)\ .\ p)$$

The difference between the notations is conceptually trivial, but important because the notation based on Theorem 2 is exponentially more succinct than the one based on Theorem 1, where the *size* of an ordinal under a given representation is the number of bits needed to denote the ordinal in that representation. To see this, consider $\omega \cdot k$: it requires $O(k)$ bits with the representation in Theorem 1 and $O(\log k)$ bits with the representation in Theorem 2.

4.2 Algorithms for Arithmetic

Despite the fact that ordinals have been studied for over 100 years, and that ordinal notations play a critical role in several fields of mathematics, we could not find a complete set of algorithms for the standard arithmetic operators on ordinal notations. We therefore defined efficient algorithms for ordinal ordering ($<$), addition, subtraction, multiplication, and exponentiation for our ordinal notation. Analysis of the correctness

Fig. 1 Basic Ordinal Functions

```
(defun natp (x)
  (and (integerp x)
       (<= 0 x)))

(defun posp (x)
  (and (integerp x)
       (< 0 x)))

(defun o-finp (x)
  (atom x))

(defmacro o-infp (x)
  `(not (o-finp ,x)))
```

```
(defun o-first-expt (x)
  (if (o-finp x)
      0
    (caar x)))

(defun o-first-coeff (x)
  (if (o-finp x)
      x
    (cdar x)))

(defun o-rst (x) (cdr x))

(defun make-ord (fe fco rst)
  (cons (cons fe fco) rst))
```

and complexity of these algorithms can be found in [15], and the complexity results are summarized in Table 1. Complexity is given in terms of the length (denoted $|\ |$) and size (denoted $\#$) of the arguments. The length of an ordinal is the length of its list representation, and the size is 1 for natural numbers and the sum of the sizes of the ordinal's exponents for infinite ordinals. The complexity of `o^` is given in terms of `natpart`, which returns the natural number at the end of the list representation of an ordinal.

Here we present the ordinal addition algorithm as an example. The basic ordinal functions on which our arithmetic algorithms are based are given in Figure 1. Note that `natp` and `posp` are recognizers for natural numbers and positive integers, respectively. The function `finp` and macro `infp` recognize whether or not an ordinal is finite. Note that `(make-ord a b c)` constructs an ordinal in our representation where `a` is the first exponent, `b` is the first coefficient, and `c` is the rest of the ordinal: `((a . b) . c)`. The functions `o-first-expt`, `o-first-coeff`, and `o-rst` deconstruct an ordinal, returning the first exponent, first coefficient, and rest of an ordinal, respectively.

Given these definitions, binary addition of two ordinals in our notation is defined as follows:

```
(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 (o-finp x) (o-finp y)) (+ x y))
      ((or (o-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))))))
```

where `(ocmp a b)` is a function that returns `lt`, `gt`, or `eq` if `a` is less than, greater than, or equal to `b`, respectively.

The correctness of this algorithm relies heavily on the properties of so-called *additive principal ordinals*, which are ordinals of the form ω^β where β is an ordinal greater than 0. There are two properties of these ordinals that concern us. The first is that they are closed under addition. That is, $\alpha < \omega^\gamma$ and $\beta < \omega^\gamma$ implies that $\alpha + \beta < \omega^\gamma$. The second is the additive principal property, which states that $\alpha < \omega^\beta$ implies that $\alpha + \omega^\beta = \omega^\beta$. Here we give several examples to illustrate ordinal addition. Multiplication and exponentiation are much more complex.

The first is $(\omega 5 + 8) + (\omega^2 3 + \omega 7 + 1)$. By associativity and the closure of additive principal ordinals, we have $(\omega 5+8)+(\omega^2 3+\omega 7+1) = \omega 5+(8+\omega^2 3)+\omega 7+1 = (\omega 5 + \omega^2 3) + \omega 7 + 1 = \omega^2 3 + \omega 7 + 1$. This corresponds to the second case of our algorithm. For the second example, consider $(\omega^2 + \omega 5 + 8) + (\omega^2 3 + \omega 7 + 1)$. This is equal to $\omega^2 + (\omega^2 3 + \omega 7 + 1)$ by our last example. By the left distributive property of multiplication, this is equal to $\omega^2 4 + \omega 7 + 1$, which corresponds to the last case of our algorithm. Finally, consider $(\omega^3 + \omega^2 + \omega 5 + 8) + (\omega^2 3 + \omega 7 + 1)$. By our last example, this is equal to $\omega^3 + \omega^2 4 + \omega 7 + 1$, which is already in normal form. This corresponds to the third case of our algorithm.

5 Changes to ACL2

In this section, we discuss how we integrated our new ordinal arithmetic results with ACL2 to make a powerful, extensible, general tool for reasoning about program termination. We partition this discussion into two sections. In Section 5.1, we give an overview of the *interface changes*, the changes that users of ACL2 will notice. This includes alterations to the core ACL2 logic and our library. In Section 5.2, we discuss the internals of our library, including how we tuned it to maximize its efficiency and effectiveness.

5.1 Interface Changes

The first and most fundamental change we made was to the ACL2 logic itself, which now uses our ordinal representation as its foundation for reasoning about induction, well-foundedness, and termination. This involved adding the helper functions in Figure 1, the ordering function, `o<`, the ordinal recognizer predicate, `o-p`, and the macros `o<=`, `o>`, and `o>=`. Once the new ordinal functions were added, we updated the affected sections of the logic to use our ordinal notation. We did not add our arithmetic functions to the base "ground-zero" ACL2 theory, but included them in a library so as to maintain the simplicity and minimality of the ground-zero theory.

The next change to ACL2 was to improve its ability to reason about arithmetic over the natural numbers and positive integers using the `natp` and `posp` functions. This was crucial for our ordinal arithmetic library, and in order to better integrate these results into ACL2, we created a library, `natp-posp`, based on these results and added it to the arithmetic module, a collection of theorems about arithmetic over the integers, rationals, and complex rationals. The result is an arithmetic module with better support for reasoning about natural numbers and positive integers.

Our library is comprised of several *books*, files of definitions and theorems. Here we review the top-level books that a typical ACL2 user might want to use. The `ordinals` and `ordinals-without-arithmetic` books provide an easy way to access all of our results about ordinal arithmetic. The difference between these books is simply that ordinals-without-arithmetic does not include ACL2's arithmetic module, which is useful for users who use different arithmetic modules.

The lexicographic-ordering book contains a proof of the well-foundedness of lists of natural numbers under the lexicographic ordering, allowing ACL2 to use the lexicographic order on naturals to prove termination, instead of the ordinals. This book is valuable for two reasons. First, it is a good tool for teaching new ACL2 users, as the lexicographic order on the naturals is simpler to explain than the ordinals. This allows new users to more quickly and easily start reasoning about termination. Second, it provides an example for more experienced users of how to use our library to prove that an ordering is well-founded.

The e0-ordinal book is useful for transfering legacy results to the new version of ACL2. It includes the predicate recognizing the old ordinals, e0-ordinalp, the corresponding ordering function, e0-ord-<, and functions for converting ordinals in this notation to and from ordinals in our notation (atoc and ctoa respectively). These functions are proved to be order-isomorphisms and inverses of each other.

We used our ordinal arithmetic library to certify the ACL2 regression suite, which is a collection of hundreds of books that formalize mathematical concepts in ACL2 and provide case studies illustrating how to model and verify large systems such as microprocessors. To deal with books that explicitly mention the old ordinals only in termination proofs, this requires simply using the old ordinal representation, which, given the isomorphism result in the e0-ordinal book, involves one call to set-well-founded-relation. However, some books contain more significant reasoning about the old ordinals and therefore require the full power of the ordinal isomorphism result; an example appears in Section 6.1.

5.2 Internal Engineering of the Books

Creating an efficient and robust library required a considerable amount of effort and in this section we discuss some of the issues.

First, we configured ACL2 to reason about the representation of the ordinals and the basic operations on them in an algebraic fashion. While ACL2 is a typeless language, it is still possible to use algebraic specifications by defining constructors and destructors for the ordinals, proving that they satisfy the appropriate properties, and then disabling the definitions. Since ACL2 is not able to use the definitions of the functions, it is forced to reason using only the algebraic theory. We did this for the functions make-ord, o-first-expt, o-first-coeff, and o-rst (see Figure 1). Besides the obvious advantages of algebraic specifications, this approach is more efficient, as otherwise the rewrite rules for manipulating ordinals are in terms of lists (which is how the ordinals are represented), but these rules interact with ACL2's rules for reasoning about lists, leading to inefficiencies.

Next, we related the most efficient version of our algorithms [15] with a simpler but less efficient version [16] using a new feature of ACL2 called mbe ("must be equal"). This feature allows the user to give two definitions for a single function, which must be proved to be equivalent under some *guard conditions* that characterize the intended domain of application. The logic definition is used by ACL2 during proof attempts. The exec definition is used as the executable version of the function, when the function is applied to the intended domain. This allows us to execute using efficient definitions, but to reason using simpler, cleaner definitions. We used `mbe` for ordinal multiplication and exponentiation.

Finally, we profiled the books. We used proof analysis tools provided by ACL2 to find sources of inefficiency in proof attempts. Theorems proved by the user cause the ACL2 system to behave differently depending on how they are tagged. Thus, as one can imagine, a large collection of theorems such as those in the ordinal library can interact in very subtle and complex ways. This makes finding sources of inefficiency difficult. For example, we originally had the following rule.

```
(defthm fe-o-p
  (implies (o-p a) (o-p (o-first-exp a)))
  :rule-classes ((:forward-chaining)))
```

Once this rule is admitted, ACL2 will add `(o-p (o-first-exp a))` to the *context*, the set of things it knows, when `(op a)` appears in the context. Note that this will not cause an infinite loop since ACL2 has heuristics for applying forward chaining rules that avoid this. Therefore, this seemed like a harmless rule to us. However, when combined with other forward chaining rules triggered by `(o-p (o-first-exp a))`, this rule gave us a significant slowdown in the verification of our books. In order to fix this, we changed the theorem to this.

```
(defthm fe-o-p
  (implies (o-p a) (o-p (o-first-exp a)))
  :rule-classes ((:forward-chaining
                   :trigger-terms ((o-first-exp a)))
                  (:rewrite :backchain-limit-lst (5))))
```

The new trigger term insures that `(o-first-exp a)` is mentioned somewhere in the theorem before the rule is used. This significantly cuts down on the number of times this rule, and the rules that are triggered by it, are used. We also tagged this theorem to be used as as rewrite rule, but only if the hypothesis can be proved in 5 or less steps. Profiling is a crucial part of engineering an effective library of theorems. We therefore carefully profiled our library, and the result was an order of magnitude improvement in performance.

6 Using the New Ordinals : Two Case Studies

In this section we provide two case studies illustrating the use of our ordinal library in ACL2. The first demonstrates how existing libraries making significant use of the ordinals in the old representation can easily be altered to use our new representation. The second case study illustrates how other users have used our ordinal arithmetic library to mechanically prove complex termination arguments.

6.1 Legacy Books: Multiset Case Study

ACL2's multiset ordering library [20] makes significant use of the ordinals. A *multiset* is a set in which items can appear more than once. For example, $\{1, 3, 2, 2, 4\}$ is a multiset over the natural numbers which contains two 2's. Given a set, A, with an order $<$, the *multiset order*, $<_{mul}$, of multisets over A is defined as follows. $N <_{mul} M$ iff

Fig. 2 Original Multiset Results

```
(encapsulate ((mp (x) booleanp)
              (rel (x y) booleanp)
              (fn (x) e0-ordinalp))
  ...

  (defthm rel-well-founded-relation-on-mp
   (and (implies (mp x) (e0-ordinalp (fn x)))
        (implies (and (mp x) (mp y) (rel x y))
                 (e0-ord-< (fn x) (fn y))))
   :rule-classes :well-founded-relation))

  ...

(defthm multiset-extension-of-rel-well-founded
  (and (implies (mp-true-listp x)
                (e0-ordinalp (map-fn-e0-ord x)))
       (implies (and (mp-true-listp x)
                     (mp-true-listp y)
                     (mul-rel x y))
                (e0-ord-< (map-fn-e0-ord x)
                          (map-fn-e0-ord y))))
  :rule-classes :well-founded-relation)
```

there exist multisets X and Y (over A), such that $\emptyset \neq X \subseteq M$, $N = (M - X) \cup Y$, and $\forall y \in Y$, $\exists x \in X$ such that $y <_A x$. If we restrict ourselves to finite sets, then if $<_A$ is well-founded, it can be shown that so is $<_{mul}$. The multiset library provides a macro called `defmul` which, given a well-founded relation over a set and a recognizer for that set, automatically generates the corresponding multiset relation and proves it to be well-founded.

The `defmul` macro depends on results proved in another book, called `multiset`, which provides useful lemmas about multisets, and uses ACL2's `encapsulate` feature to prove in general that a multiset extension of a well-founded relation is well-founded (See Figure 2). The encapsulated code hides the details of the functions from the rest of the book. All that is known outside the encapsulate is that `mp` and `rel` return boolean values, `fn` returns an ordinal in the old representation, and `rel` has been proved to be well-founded on the set recognized by `mp` using the embedding `fn`. Following this encapsulate, there are a number of lemmas about these functions based only on that information, which culminate in the proof of the well-foundedness of the multiset extension of `rel`.

There are two problems in certifying this book using the new version of ACL2. The first is that the original theorem declaring the well-foundedness of `rel` is no longer a proof of well-foundedness. The embedding, `fn` must return an ordinal in the new representation in an order-preserving way. The second problem is that the final theorem about the well-foundedness of the multiset extension must also be altered to use our new ordinals.

The solution is relatively simple, and relies on the results of our `e0-ordinal` book. Using our conversion functions, `ctoa` and `atoc`, we transfered the results of the multiset book to results about the new ordinal notation. First, we altered the encapsulate so that `fn` and the well-foundedness result were in terms of the new ordinals. This simply required replacing `e0-ordinalp` and `e0-ord-<` by `o-p` and `o<`, respectively.

Next, we added the following macro.

```
(defmacro fn0 (x) `(ctoa (fn ,x)))
```

This simply converts the ordinal in the new notation given by `fn` into the corresponding ordinal in the old representation. The theorems involving `fn` were changed to use `fn0` instead. After the final result (which we renamed and retagged as a rewrite rule), we added the following lines of code to convert the results into a valid well-founded-relation argument using the new ordinal notation.

```
(defun map-fn-op (x)
  (atoc (map-fn-e0-ord x)))

(defthm multiset-extension-of-rel-well-founded
  (and (implies (mp-true-listp x) (o-p (map-fn-op x)))
       (implies (and (mp-true-listp x)
                     (mp-true-listp y)
                     (mul-rel x y))
                (o< (map-fn-op x) (map-fn-op y))))
  :rule-classes :well-founded-relation)
```

Finally, we changed the `defmul` macro so that it uses the new theorem and function names. With this approach, we did not have to alter the lemmas about the old ordinals in `multiset`. Doing so would have required essentially modifying the entire book. This “wrapping” method can be used to quickly and easily update old libraries so that they can be certified using the new ordinals.

6.2 New Results: Dickson’s Lemma Case Study

Our library was used by Sustik to give a constructive proof of Dickson’s Lemma [21]. This is a key lemma in the proof of the termination of Buchberger’s algorithm for finding a Gröbner basis of a polynomial ideal, and is therefore an important step toward the larger goal of formalizing results from algebra in ACL2 [17]. Sustik made essential use of the ordinals and our library, as his proof depends heavily on the ordinals and could not have been proved in older versions of ACL2 without essentially building up a theory of ordinal arithmetic similar to our own. Our library was able to automatically discharge all the proof obligations involving the ordinals.

Dickson’s Lemma states that, given an infinite sequence of monomials, m_0, m_1, m_2, ..., there exists $i, j \in \mathbb{N}$ such that $i < j$ and m_i divides m_j. Sustik’s argument involves mapping initial segments of the monomial sequence into the ordinals such that if Dickson’s lemma fails, the ordinal sequence will be decreasing. Thus, the existence of an infinite sequence of monomials such that no monomial divides a later monomial implies the existence of an infinite decreasing sequence of ordinals, which is not possible due to the well-foundedness of the ordinals.

This proof relies heavily on ordinal addition and exponentiation. For example, sets of monomials, which are represented as lists of tuples of natural numbers, are mapped to the ordinals by the following function.

```
(defun tuple-set->ordinal-partial-sum (k S i)
  (cond ((or (not (natp k)) (not (natp i))) 0)
        ((zp k) 0)
        ((equal k 1)
         (tuple-set-min-first S))
        ((<= (tuple-set-max-first S) i)
         (o^ (omega) (o+ (tuple-set->ordinal-partial-sum
                           (1- k) (tuple-set-projection S) 0)
                         1)))
        (T (o+ (o^ (omega)
                   (tuple-set->ordinal-partial-sum
                    (1- k) (tuple-set-filter-projection S i) 0))
                  (tuple-set->ordinal-partial-sum k S (1+ i))))))
```

Key lemmas about this function therefore required sophisticated reasoning about the behavior of ordinal addition and exponentiation. One such lemma is as follows.

```
(defthm map-lemma-3.2
  (implies (and (tuple-setp k A) (natp k) (< 1 k) (natp i))
           (o< (o^ (omega) (tuple-set->ordinal-partial-sum
                              (1- k)
                              (tuple-set-filter-projection A i)
                              0))
               (tuple-set->ordinal-partial-sum k A i))))
```

This and other similar theorems require results about ordinal arithmetic including the following: (1) $\alpha < \beta \Rightarrow \gamma + \alpha < \gamma + \beta$, (2) $\alpha \leq \beta \Rightarrow \alpha + \gamma \leq \beta + \gamma$, (3) $\alpha \leq \beta \wedge \gamma \leq \delta \Rightarrow \alpha+\gamma \leq \beta+\delta$, (4) $\alpha < \beta \Rightarrow \gamma^{\alpha} < \gamma^{\beta}$, and (5) $\alpha \leq \beta \Rightarrow \alpha^{\gamma} \leq \beta^{\gamma}$.

Initially, Sustik used a preliminary version of our library, and he needed 26 additional theorems about ordinal arithmetic for his proof. After streamlining our library, no additional ordinal arithmetic lemmas were required, and the results specific to Dickson's Lemma, such as those above, were discharge twice as quickly. The overall result was a 70.5% speedup in the verification of the Dickson's Lemma library. This is an example of the kind of termination proof that would be quite difficult to fully automate.

7 Lessons Learned

During this project we learned several lessons that we believe will be of benefit to users working on large projects in ACL2 and similar systems. These include lessons about the features and shortcomings of ACL2, as well as lessons about effectively designing and implementing large projects in ACL2. Here, we share some of these lessons.

One invaluable feature of ACL2 is its regression suite. This large collection of books includes the formalization of many mathematical theories and industrial case studies, making it a valuable testbed for new features. Running the regression suite on our altered version of ACL2 stressed our library and helped us maximize its efficiency and effectiveness. Along the way we learned two valuable lessons. The first is that it is important to have a general way of integrating results into the regression suite. In our case, we used the ordinal isomorphism results, as we illustrate in Section 6.1, to transfer results about the old ordinals to the new ordinals; this saved us from having to understand the details of existing books. The second lesson we learned is that the regression suite can reveal patterns in the use of ACL2 that can inspire new improvements. For example, we did not originally plan on integrating our results about `natp` and `posp` with the arithmetic module. However, when working with the regression suite, we found that many libraries contained functions similar to `natp` and `posp`, and this prompted us to create a separate book that we added to the arithmetic module.

Another feature of ACL2 is its extensive documentation [12]. It describes each ACL2 feature and function in detail, and an important part of integrating our work into ACL2 was updating the documentation. This included describing our functions, but, more importantly, it required us to reason at the meta-level, providing a hand-written proof of the well-foundedness of our ordinal notation (which does not appeal to the ordinals), in order to demonstrate the soundness of our new additions to ACL2. Thus, updating the documentation is important both for keeping users up-to-date with the current features of ACL2 and for arguing at a meta-level about the soundness of the ACL2 logic.

As we mentioned earlier, profiling was a crucial step in making our library more efficient. What we found is that this is actually very difficult to do in ACL2. There is a mechanism called `accumulated-persistance` that allows the user to gauge the performance of each individual rule. However, many performance problems come from the interaction among the rules, not from each rule's individual performance. We think that ACL2 users would benefit from a mechanism for analyzing this interaction. For example, one can imagine having a mechanism for reporting the amount of time spent on rules of each class (*e.g.*, forward chaining rules versus rewrite rules). Since rules of one class often trigger other rules of the same class, this could prove to be useful.

Another shortcoming of the ACL2 system is the naming scheme, which it has borrowed from Lisp. Namespace collisions can be avoided in ACL2 by creating new packages. For example, we could have created a package called `ORD`, and defined all our functions in that package (*e.g.*, `ORD::o<`). In fact, this would have been useful for us, since we found functions called `op` (the original name of our predicate function) and `natp` in several libraries in the regression suite. However, referring to one package from another involves either prefixing symbols with package names or importing symbols into the current package (thus causing namespace issues again). It usually takes several iterations to determine which symbols a package should import, but the ACL2 implementation requires restarting ACL2 for every such change. In the end, we found it easier to rename our predicate function to `o-p` and to rename or delete the `natp` functions found in other books. ACL2 users would benefit from a better mechanism for dealing with namespace issues.

Our use of algebraic specifications to deal with `make-ord`, `o-first-expt`, `o-first-coeff`, and `o-rst` sped up our books, but it took several iterations to discover where abstraction should be used. We found that algebraic specifications are often more trouble than they are worth. When in doubt, we recommend starting with little or no abstraction, and adding more based on how functions are being used in proof attempts. If the theorem prover seems to be struggling with the underlying representation, then perhaps abstraction can help.

Finally, we learned the value of recording lessons learned while working on a big project in ACL2. We have noticed through past experience that users (including us) often make the same mistakes repeatedly. They have to rediscover ways to improve their libraries or avoid pitfalls. Having a record of these tips, tricks, and lessons can potentially be a valuable time-saver when working on new projects. They are also valuable for finding difficulties with ACL2 such as the ones we presented here, which can be used to improve the theorem-proving system and may provide insight that will help developers of other theorem proving systems as well.

8 Related Work

There has been a significant amount of work dealing with the problem of termination in recent years. Much of it has focused on the termination of term rewriting systems (TRSs). Current techniques for proving the termination of TRSs can be found in [2]. One such method is the *interpretation method*, which involves mapping terms into a well-founded set and showing that the left-hand-side of rewrite rules map to a bigger value than the corresponding right-hand-side for all rules. Another method involves *simplification orders* which are orders over terms such that terms are always greater than their subterms. This often involves extending a well-founded order on the signature of the TRS to apply to all terms. Popular simplification orders include the *lexicographic path ordering* and *Knuth-Bendix orderings*. One method for proving termination of TRSs using simplification orders is called the *dependency pair method* [1]. Recent work focusing on automating this method has met with some success [8]. These methods, while useful in the context of theorem proving and optimization in compilers, are designed for TRSs rather than programming languages used in practice. They therefore have not been shown to scale to the complexities of actual programming languages.

Another approach to termination is the size-change principal [14]. This method involves using a well-order on function parameters, analyzing recursive calls to label any clearly decreasing parameters. All possible infinite sequences of function calls are then checked to be sure that there are infinite decreases and only finitely many possible increases in the values of arguments to recursive function calls. This is similar but much less sophisticated than ACL2's termination reasoning. For example, there is no explicit description on how to determine if a function parameter "decreases." The examples are based on a simple toy language, and the analysis of arguments of the form `(f x)`, where `f` is a user-defined function, is not considered. Only primitive operations are dealt with. The conditions under which recursive calls are made are not taken into account, *e.g.*, if a recursive call is made in the else branch of an if statement, we know that the test of the if statement is false at that point. This information is often *necessary* for establishing termination.

There are many other methods that can potentially be extended to deal with full programming languages. Podelski and Rybalchenko give a complete method for proving termination of non-nested loops with linear ranking functions [19]. Dams, Gerth, and Grumberg give a heuristic for automatically generating ranking functions [7]. Colón and Sipma give two algorithms for proving termination, one which synthesizes linear ranking functions, but is limited to programs with few variables, and one which is more heuristic in nature and converges faster on the invariants it can discover [6, 5].

What is novel about our approach is that we focus on extendability and generality. The result is a system into which new heuristics and techniques (such as the ones cited above) can be incorporated in order to improve automation. However, when these techniques fail (as they eventually must, since termination is undecidable), the user can interact with the theorem prover to find a proof.

9 Conclusions and Future Work

We have developed a general framework based on the ordinals for proving program termination, which has been incorporated into ACL2 v2.8. The resulting system allows us to prove termination in a general context for arbitrary programs and in a highly automated fashion, as we demonstrated with the case study of Dickson's Lemma. For future work, we plan to add decision procedures and heuristics to our framework to further automate ACL2's ability to reason about termination. We also plan to use ACL2 as a back-end reasoning engine, combined with a front-end system containing static analysis techniques in order to reason about the termination of programs written in imperative languages such as C and Java.

Acknowledgments

We would like to thank the authors and maintainers of ACL2, J Moore and Matt Kaufmann for their help and support of our project. We also wish to thank Mátyás Sustik for providing us with the Dickson's Lemma book, which proved very helpful as a testbed for our library.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. R. S. Boyer and J. S. Moore. Proving theorems about pure lisp functions. *JACM*, 22(1):129–144, 1975.
4. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
5. M. A. Colón and H. B. Sipma. Synthesis of linear ranking functions. In *TACAS01: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81, 2001.
6. M. A. Colón and H. B. Sipma. Practical methods for proving program termination. In *International Conference on Computer Aided Verification, CAV'02*, volume 2404 of *LNCS*, pages 442–454, 2002.

7. D. Dams, R. Gerth, and O. Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advanced Verification*, July 2000. See URL `http://www.cs.utah.edu/wave/`.
8. N. Hirokawa and A. Middledorp. Automating the dependency pair method. In *Automated Deduction – CADE-19*, LNAI, pages 32–46. Springer-Verlag, 2003.
9. A. J. Hu and M. Y. Vardi, editors. *Computer-Aided Verification – CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.
10. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
11. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.
12. M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.-utexas.edu/users/moore/acl2`.
13. M. Kaufmann and J. S. Moore, editors. *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003. See URL `http://www.cs.-utexas.edu/users/moore/acl2/workshop-2003/`.
14. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. *ACM Symposium on Principles of Programming Languages*, 28:81–92, 2001.
15. P. Manolios and D. Vroon. Algorithms for ordinal arithmetic. In F. Baader, editor, *19th International Conference on Automated Deduction – CADE-19*, volume 2741 of *LNAI*, pages 243–257. Springer–Verlag, July/August 2003.
16. P. Manolios and D. Vroon. Ordinal arithmetic in ACL2. In Kaufmann and Moore [13]. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/`.
17. F.-J. Martín-Mateos, J.-A. Alonso, M.-J. Hidalgo, and J.-L. Ruiz-Reina. A formal proof of Dickson's Lemma in ACL2. In M. Y. Vardi and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2003)*, volume 2850 of *LNCS*, pages 49–58. Springer Verlag, 2003.
18. F. Morris and C. Jones. An early program proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, April–June 1984.
19. A. Podelske and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, VMCAI 2004*, volume 2937 of *LNCS*, pages 239–251, 2004.
20. J.-L. Ruiz-Reina, J.-A. Alonso, M.-J. Hidalgo, and F.-J. Martin. Multiset relations: A tool for proving termination. In M. Kaufmann and J. S. Moore, editors, *Proceedings of the ACL2 Workshop 2000*. The University of Texas at Austin, Technical Report TR-00-29, November 2000.
21. M. Sustik. Proof of Dixon's lemma using the ACL2 theorem prover via an explicit ordinal mapping. In Kaufmann and Moore [13]. See URL `http://www.cs.utexas.edu/-users/moore/acl2/workshop-2003/`.
22. M. Wilding, D. Greve, and D. Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3):233–248, May 2001.