AUTOMATICALLY PROVING THE TERMINATION OF FUNCTIONAL PROGRAMS

A Dissertation
Presented to
The Academic Faculty

by

Daron Vroon

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December, 2007

AUTOMATICALLY PROVING THE TERMINATION OF FUNCTIONAL PROGRAMS

Approved by:

Panagiotis Manolios, Advisor
College of Computing
Georgia Institute of Technology

Byron Cook
Microsoft Research, Cambridge Lab

Eric Feron
Department of Aerospace Engineering
Georgia Institute of Technology

Richard Lipton
College of Computing
Georgia Institute of Technology

J Strother Moore
Department of Computer Sciences
The University of Texas at Austin

Professor Santosh Pande
College of Computing
Georgia Institute of Technology

Date Approved: August 7, 2007

To my wife,

Julie,

for her tireless love and support.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Panagiotis Manolios. His guidance in matters of research and life have been invaluable, and I am a better researcher and a richer person for it. I also must thank him for his unwavering support for me from start to finish, even when I declared my intention to follow a different career path. Without it, I would not have finished this dissertation.

I would also like to thank J Moore and Matt Kaufmann, the authors of ACL2, for their collaboration and support for this research. Their tireless efforts to improve ACL2 has made it the award-winning system it is today, and their help was key in the implementation of my own work in ACL2. I also would like to thank Mátyás Sustik for developing his Dickson's Lemma book, which proved very helpful as a testbed for our ordinal arithmetic work.

My family and friends all deserve my gratitude as well, for their emotional support, encouragement, and the occasional distraction.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Establishing the termination of programs is a fundamental problem in the field of software verification. For transformational programs, termination is used to extend partial correctness to total correctness. For reactive systems, termination reasoning is used to establish liveness properties. In the context of theorem proving, termination is used to establish the consistency of definitional axioms and to automate proofs by induction. Of course, termination is an undecidable problem, as Turing himself proved. However, the question remains: how automatic can a general termination analysis be in practice?

In this dissertation, we develop two new general frameworks for reasoning about termination and demonstrate their effectiveness in automating the task of proving termination in the domain of applicative first-order functional languages.

The foundation of the first framework is the development of the first known complete set of algorithms for ordinal arithmetic over an ordinal notation. We provide algorithms for ordinal ordering ($<$), addition, subtraction, multiplication, and exponentiation on the ordinals up to ε_0. We prove correctness and complexity results for each algorithm. We also create a library for automating arithmetic reasoning over ε_0 in the ACL2 theorem proving system. This ordinal library enables new termination proofs that were previously not possible in previous versions of ACL2.

The foundation of the second framework is an algorithm for fully automating termination reasoning with no user assistance. This algorithm uses a combination of theorem proving and static analysis to create a Calling Context Graph (CCG), a novel abstraction that captures the looping behavior of the program. Calling Context Measures (CCMs) are then used to prove that no infinite path through the CCG can be an actual computation of the program. We implement this algorithm in the ACL2, and empirically evaluate its effectiveness on the regression suite, a collection of over 11,000 user-defined functions from a wide variety of applications.

CHAPTER I

INTRODUCTION

In this work we defend the following thesis:

> *A highly automatic, general, interactive, and efficient termination analysis is possible for feature-rich, first-order, purely functional programming languages.*

The problem of proving program termination has one of the longest and richest histories of any in the field of Computer Science. It was introduced as the "Printing Problem" and proven to be uncomputable by Alan Turing in the same 1936 paper in which he introduced Turing Machines and the concept of computability [113]. Despite the difficulty of the problem, Turing recognized its importance, writing that "The checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified" [115, 85].

The difficulty and importance of termination analysis is still recognized today as can be seen in the terminology of the verification community. In the context of the verification of transformational programs (*i.e.*, programs that take a set of inputs, calculate an output, and stop), the community refers to *partial* and *total* correctness. A program for which partial correctness has been shown is guaranteed to give the correct answer *if* it terminates. A program for which total correctness has been shown is partially correct and guaranteed to terminate [1].

Termination is also an important concept in the context of reactive systems (*i.e.*, programs that do not terminate, but engage in ongoing interactions with their environments) such as operating systems and networking protocols. In this context, it is important to show that desired responses to stimuli are not postponed forever. Such a property is known as a *liveness* property.

A third context in which termination plays an important role is that of mechanical theorem proving. Here, termination is used to show that axioms defining new functions

are consistent and conservative extensions of the current logical world. For example, in the ACL2 theorem proving system [57, 56], all functions must be proven to terminate before they will be admitted into the logic. This prevents the admission of function definitions such as $f(x) = \neg f(x)$, which would extend the current theory in an inconsistent way.

In this dissertation, we focus on proving termination in the domain of feature-rich, first-order, purely functional programming languages, of which ACL2's programming language is an example. The challenges presented by this class of languages, are challenges that are faced when reasoning about the termination of any modern programming language, and include the following.

- Reasoning about a rich set of data-types including some that are not well-founded, including complex rationals, rationals, and integers, and those that are non-algebraic, such as lists and trees.

- Complex and general looping behaviors including non-algebraic loops —*e.g.*, loops over data structures such as graphs and trees, looping behavior that is non-linear and non-polynomial —*e.g.*, loops using modular arithmetic, absolute value, and minimum and maximum functions.

- Arbitrarily complex mutual recursion and nested loops.

- Reasoning about the output of user-defined functions.

- Large and complex industrial-scale code bases, such as a nearly complete model of the Java Virtual Machine [66], efficient executable models of hardware from AMD [82, 101, 102, 107], Motorola [17, 18], and Rockwell-Collins [47], a proof checker [79], and a verified model checker for the μ-calculus [67].

The goal of this dissertation, then, is to overcome these challenges by developing novel and general techniques for mechanically reasoning about termination in this domain. We demonstrate these techniques to be highly automatic, general, interactive, and efficient by implementing and empirically evaluating them in ACL2.

1.1 Contributions and Organization of this Dissertation

Here we outline the contributions and structure of this dissertation. Some of the results that appear in this dissertation have appeared in previous conference proceedings and journals [70, 71, 72, 74, 75, 73, 54, 45, 46]. This work is divided into four parts.

In Part I, we present a core semantics for an applicative first-order functional language and describe the "traditional" method for proving the termination of functions written in such a language, based on the concept of *measures*. We also give a brief overview of the ACL2 theorem proving system, in which our work is implemented and empirically evaluated.

In Part II, we present our general framework for reasoning about termination, which is based on a constructive theory of ordinal arithmetic. We give the first known complete set of algorithms for ordinal ordering, addition, subtraction, multiplication, and exponentiation on the ordinals up to ε_0. We develop efficient algorithms and prove correctness and complexity results for each algorithm. We discuss the mechanical verification of the algorithms in ACL2 which we accomplish by checking that they satisfy well-known properties of ordinal arithmetic. We also describe the engineering of a powerful library for reasoning about ordinal arithmetic that allows ACL2 users to verify termination-related results that were previously beyond ACL2's capabilities.

We discuss the challenges of fully integrating this work into ACL2's logic by replacing ACL2's previous ordinal notation with the more succinct notation used in our algorithms. We provide case studies that demonstrate that this new version of the ACL2 logic maintains support for legacy ACL2 code while enabling new termination related results, and discuss the lessons we learned while developing this framework. The result is a powerful framework for mechanically reasoning about the algebraic properties of the ordinals.

In Part III, we present a new termination analysis based on calling context graphs (CCGs) for a fully featured class of modern first-order functional programming languages. These CCGs, constructed through a novel combination of theorem proving and static analysis, give a manageable but surprisingly accurate representation of the programs looping behavior. The termination proof then involves assigning sets of calling context measures (CCMs) over well-founded domains to the calls and showing that for every possible infinite

sequence there is a corresponding sequence of CCMs that is infinitely decreasing.

We present an algorithm based on CCGs and CCMs that can automatically reason about any source of looping behavior in first order purely functional programming languages and which can automatically handle a much larger class of programs than previous approaches. We discuss the implementation of this algorithm, including a hierarchical approach which attempts termination proofs using more lightweight versions of the CCG analysis, and only resorts to the more powerful, but slower, full version of the CCG analysis when these faster analyses fail. We present experimental results that demonstrate the effectiveness of our analysis on a test-bed of over 11,000 functions, representing real user-submitted ACL2 code. Our analysis can prove termination in over 98% of all cases, and is consistently over 20% more effective than ACL2's current analysis on non-trivial recursive function definitions.

Finally, we discuss the integration of our termination analysis into the ACL2 logic, proving that it results in no change to the ACL2 logic, and can be engineered to interface with ACL2's automatic theorem prover.

Part IV of this dissertation is a discussion of future work and conclusions.

PART I

Preliminaries

CHAPTER II

APPLICATIVE FIRST-ORDER FUNCTIONAL LANGUAGES

In this chapter, we describe the semantics of a standard applicative first-order functional language, FL, which we use throughout the dissertation to develop our termination analysis. As a starting point for thinking about the termination of programs written in such a language, we present a well-known general termination proof technique for such languages based on the notion of well-founded *measures*. Finally, we introduce the ACL2 theorem proving system, which has a language similar to FL and is the testbed in which we implement our termination analyses.

2.1 Programming Language

As a programming language, FL can best be thought of as an applicative —side-effect free or purely functional— subset of Lisp. Its semantics are given in Figure 1 on the next page. These semantics are similar to what can be found in standard programming language texts. Readers interested in the correctness proofs in Chapter 8 should take the time to understand these semantics. Other readers may want to skim the remainder of this section initially, returning as needed later.

We are concerned with proving the termination of well-formed function definitions (members of the set *Def*), which are of the form `(defun` f `(`x^f_1 ... $x^f_{ar(f)}$`)` e^f`)`, where $f \in FName$ is a function name, $x^f_1, \ldots, x^f_{ar(f)} \in Var$ are the *formals* of f, and $e^f \in Expr$, called the *body* of f, is an expression whose free variables are a subset of $\{x^f_i \mid 1 \leq i \leq ar(f)\}$. We use this notation throughout the dissertation, denoting the arity of a function definition, f as $ar(f)$, the i^{th} formal of f as x^f_i, and the body of f as e^f.

We leave the universe of values, denoted *Val*, as undefined, but our examples throughout the text use values of integer and list types. Since this is a first order language, functions are not first class data objects, and are not included in *Val*. We use $\perp$ (which is not in *Val*) to denote non-termination, and $Val_\perp = Val \cup \{\perp\}$. An environment maps variables (*i.e.*,

$$
\begin{array}{rcl}
d & \in & \mathit{Def} \\
f & \in & \mathit{FName} \\
x & \in & \mathit{Var} \\
e & \in & \mathit{Expr} \\
v & \in & \mathit{Val} \\
u & \in & \mathit{Val}_\bot = \mathit{Val} \cup \{\bot\} \\
\epsilon & \in & \mathit{Env} = \mathit{Var} \to \mathit{Val} \\
\phi & \in & \mathit{Fun} = \mathit{Val}^* \to \mathit{Val}_\bot \\
\psi & \in & \mathit{TFun} = \mathit{Val}^* \to \mathit{Val} \subseteq \mathit{Fun} \\
h & \in & \mathit{IHist} = \mathit{FName} \to \mathit{Val}^* \to \mathit{Val}_\bot \\
H & \in & \mathit{Hist} = \mathit{FName} \to \mathit{Val}^* \to \mathit{Val} \subseteq \mathit{IHist}
\end{array}
$$

$$
\begin{array}{rcl}
[\![e]\!]^h\,\epsilon & : & \mathit{Expr} \times \mathit{IHist} \times \mathit{Env} \to \mathit{Val}_\bot \\
\mathit{str} & : & \mathit{Fun} \to \mathit{Val}_\bot^* \to \mathit{Val}_\bot \\
\mathcal{D}\,[\![d]\!]\,H & : & \mathit{Def} \times \mathit{Hist} \to \mathit{Fun}^+ \\
\mathit{fix} & : & (\mathit{Fun}^* \to \mathit{Fun}^*) \times \mathit{Fun}^* \to \mathit{Fun}^*
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{str}\ (\phi)\ \langle u_i \rangle_{i=1}^n & = & \begin{cases} \bot & \text{if } \langle \exists i \in [1..n] :: u_i = \bot \rangle \\ \phi \langle u_i \rangle_{i=1}^n & \text{otherwise} \end{cases} \\
\mathit{fix}\ \xi\ \Phi & = & \lim\limits_{j \to \omega} \xi^j\ \Phi
\end{array}
$$

$$
\begin{array}{rcl}
[\![x]\!]^h\,\epsilon = \epsilon.x \quad , \quad [\![v]\!]^h\,\epsilon = v & & \\
[\![\texttt{(}f\ e_1\ \ldots e_n\texttt{)}]\!]^h\,\epsilon & = & \mathit{str}\ (h.f)\ \langle [\![e_i]\!]^h\,\epsilon \rangle_{i=1}^n \\
[\![\texttt{(let ((}x_1\ e_1\texttt{) ... (}x_n\ e_n\texttt{)) }e\texttt{)}]\!]^h\,\epsilon & = & \mathit{str}\ \left(\lambda \langle v_i \rangle_{i=1}^n\ .\ [\![e]\!]^h\,\epsilon[x_i \mapsto v_i]_{i=1}^n\right)\ \langle [\![e_i]\!]^h\,\epsilon \rangle_{i=1}^n \\
[\![\texttt{(if }e_1\ e_2\ e_3\texttt{)}]\!]^h\,\epsilon & = & \mathit{str}\ \left(\lambda(v)\ .\ \begin{cases} [\![e_2]\!]^h\,\epsilon & \text{if } v \neq \texttt{nil}, \\ [\![e_3]\!]^h\,\epsilon & \text{otherwise.} \end{cases}\right)\ \langle [\![e_1]\!]^h\,\epsilon \rangle \\
\mathcal{D}\left[\!\!\left[\begin{array}{l} \texttt{(defun }f_1\ \texttt{(}x_1^{f_1}\ \ldots\ x_{ar(f_1)}^{f_1}\texttt{)}\ e^{f_1}\texttt{)} \\ \ldots \\ \texttt{(defun }f_k\ \texttt{(}x_1^{f_k}\ \ldots\ x_{ar(f_1)}^{f_k}\texttt{)}\ e^{f_k}\texttt{)} \end{array} \right]\!\!\right] H & = & \mathit{fix}\ (\mathit{nextfs})\ \langle \lambda \langle v_j \rangle_{j=1}^{ar(f_i)}\ .\ \bot \rangle_{i=1}^k
\end{array}
$$

$$
\text{where } \mathit{nextfs}\ \langle \phi_i \rangle_{i=1}^k = \langle \lambda \langle v_{i,j} \rangle_{j=1}^{ar(f_i)}\ .\ [\![e_i]\!]^{H[f_j \mapsto \phi_j]_{j=1}^k\,[x_j^i \mapsto v_j^i]_{j=1}^{n_i}} \rangle_{i=1}^k
$$

Figure 1: Partial semantics of a core of the ACL2 programming language.

members of *Var*) to values. The set of environments is denoted as *Env*.

Function definitions in FL denote mathematical functions, which can either be members of the set *Fun* or *TFun*. *Fun* consists of a set of partial functions, which means that for some inputs, functions in *Fun* may return $\bot$, denoting non-termination. *TFun* is the subset of *Fun* consisting of all the total (*i.e.*, terminating) functions. A *history* (*i.e.*, a member of *Hist*) maps function names to total functions (of the appropriate arity) and an *intermediate history* (*i.e.*, a member of *IHist*) maps function names to partial functions (of the appropriate arity).

The termination problem we consider is: given a history, H, and a set of mutually

recursive definitions, d, show that the functions corresponding to the definitions in d are terminating. To do this, we need to refer not only to H, but also to the (possibly partial) functions corresponding to the definitions in d. This is accomplished by using an intermediate history, h, which is just H extended so that it includes the function names appearing in d and their corresponding functions, as given by the semantics in Figure 1 on the preceding page (and described in more detail in the next paragraph). We then attempt to prove that the functions defined in d terminate, which implies that the intermediate history, h, is actually a history. If so, we have a new history. Otherwise, we reject d, revert to H, and report the problem to the user. This allows the user to incrementally define programs, as is common in programming environments for functional languages, such as Lisp. Before the user provides any new definitions, they are presented with a "ground-zero" history, that defines basic operators such as `and`, `or`, `not`, `iff`, `implies`, `+`, `-`, `/`, `*`, and so on.

We use four functions to define the semantics of FL. The function $[\![e]\!]^h\,\epsilon$ defines how to evaluate an expression, e, given an intermediate history, h, and an environment, ϵ. The function *str* corresponds to strict application. As input, it takes a function and a vector of values (possibly including $\perp$, which indicates non-termination). It returns $\perp$ if any of the input values is $\perp$; otherwise, it returns the result of applying the function to the values (which could also be $\perp$). The definitions of the semantic functions for variables, values, function application, `let`s, and `if`s are now straightforward.

Function definitions are handled with $\mathcal{D}\,[\![d]\!]\,H$, which defines what mathematical functions (elements of $Functs$) correspond to a set of function definitions, d, given history H. Its definition depends on the *fix* function, which is used to define the semantics of recursive function definitions using the standard fixpoint approach. The *fix* function takes as input ξ, a function from a vector of functions to a vector of functions, and a sequence of initial functions, Φ, and returns the vector of functions obtained by taking the limit as j approaches infinity of applying ξ to Φ j times. The definition of $\mathcal{D}\,[\![d]\!]\,H$ uses *fix* along with another function, *nextfs*, which "unrolls" the function definitions one time. The initial values given to *fix* are functions that immediately return $\perp$. The application of *fix* to these functions then "unrolls" the bodies of the definitions an unbounded number of times, which results

in a vector of partial functions that corresponds to the semantics of the definitions.

Throughout the rest of this dissertation, unless otherwise specified, we assume a fixed history, H and a set of syntactically correct, mutually-recursive function definitions, d, such that none of the function names in d are the same as those in the domain of H. The intermediate history h is obtained by extending H with the semantics of the function definitions in d.

We provide the following definitions to help in reasoning about FL programs. Throughout the rest of the dissertation, we denote the free variables of an expression, e, as $\mathit{free}(e)$. To talk precisely about subexpressions, we define *positions*, which tell us where within an expression a given subexpression is located. This allows us to avoid confusion when subexpressions at different positions are syntactically the same.

Definition 2.1.1. Given an expression, e, the *set of positions of* e, denoted $Pos(e)$ is defined recursively as follows:

- $Pos(\texttt{(if } e_1\ e_2\ e_3\texttt{)}) = \{\epsilon\} \cup \bigcup_{i=1}^{3}\{ip \mid p \in Pos(e_i)\}$,
- $Pos(\texttt{(let ((}x_1\ e_1\texttt{) ... (}x_n\ e_n\texttt{)) } e_{n+1}\texttt{)}) = \{\epsilon\} \cup \bigcup_{i=1}^{n+1}\{ip \mid p \in Pos(e_i)\}$,
- $Pos(\texttt{(}f\ e_1 \ldots e_n\texttt{)}) = \{\epsilon\} \cup \bigcup_{i=1}^{n}\{ip \mid p \in Pos(e_i)\}$,
- $Pos(v) = Pos(x) = \epsilon$

We use *variable substitutions* in the normal way to replace variables inside an expression with an expression. More formally, we have the following.

Definition 2.1.2. A *variable substitution* is a function $\sigma : V \to \mathit{Expr}$ for some $V \subseteq \mathit{Var}$. The application of σ to e, denoted $e\sigma$, is defined recursively as follows.

- $v\sigma = v$.
- $x\sigma = \begin{cases} \sigma.x & \text{if } x \in Dom(\sigma) \\ x & \text{otherwise} \end{cases}$
- $\texttt{(}f\ e_1\ \ldots\ e_n\texttt{)}\sigma = \texttt{(}f\ e_1\sigma\ \ldots\ e_n\sigma\texttt{)}$.
- $\texttt{(let ((}x_1\ e_1\texttt{) ... (}x_n\ e_n\texttt{)) } e_{n+1}\texttt{)}\sigma = \texttt{(let ((}x_1\ e_1\sigma\texttt{) ... (}x_n\ e_n\sigma\texttt{)) } e_{n+1}\texttt{)}$

- `(if` e_1 e_2 e_3`)` $= $ `(if` $e_1\sigma$ $e_2\sigma$ $e_3\sigma$`)`

The reader may notice that this particular definition of variable substitution is simpler than the standard definition and ignores the possible problem of variable capture. The difference is with `let` expressions, for which the substitution is not applied to the body of the `let`. This may cause problems if the body of the `let` contains free variables that would normally be substituted by σ. We give this particular definition in order to simplify our presentation. To avoid the previously mentioned pitfall, we assume that all of the free variables in the body of the `let` are bound in the `let` expression itself. This is a reasonable assumption, since any expression of the form `(let ((`x_1 e_1`) ... (`x_n e_n`))` e_{n+1}`)` such that $\{x'_1, \ldots x'_m\} = \mathit{free}(e_{n+1} - \{x_1, \ldots, x_n\}$ can be replaced with `(let ((`x_1 e_1`) ... (`x_n e_n`)` $(x'_1\ x'_1)$ `...` $(x'_m\ x'_m)$`))` e_{n+1}`)`, thereby binding every free variable in e_{n+1} in the bindings of the `let` expression.

Next, we define the notion of *let-adjusted subexpressions*, which are subexpressions with all applicable `let` bindings applied statically.

Definition 2.1.3. The let-adjusted subexpression of $e \in \mathit{Expr}$ at $p \in \mathit{Pos}(e)$, denoted $e|_p$, is defined recursively as follows.

- $e|_\epsilon = e$.
- `(if` e_1 e_2 e_3`)`$|_{ip} = e_i|_p$.
- `(let ((`x_1 e_1`) ... (`$x_n e_n$`))` e_{n+1}`)`$|_{ip} = \begin{cases} e_i|_p & \text{if } 1 \leq i \leq n \\ e_i[x_j \mapsto e_j]_{j=1}^n|_p & \text{otherwise} \end{cases}$
- `(`f e_1 `...` e_n`)`$|_{ip} = e_i|_p$

By $\sigma\sigma'$ we mean the substitution

$$\lambda x'. \begin{cases} \sigma'.x' & \text{if } x' \in \mathit{Dom}(\sigma'). \\ \sigma.x' & \text{otherwise.} \end{cases}$$

By $[x \mapsto e]$, we mean the substitution that maps x to e. Finally, by $[x_i \mapsto e_i]_{i=1}^n$, we mean $[x_1 \mapsto e_1][x_2 \mapsto e_2] \ldots [x_n \mapsto e_n]$. We also extend this notation to apply substitutions to sets

of expressions. Thus, $E\sigma = \{e\sigma \mid e \in E\}$. We also define the following special substitution related to function calls.

Definition 2.1.4. The *call substitution* of $e = (f\ \ e_1\ \ e_2\ \ \ldots\ \ e_n)$, denoted σ_e, maps x_i to e_i for all $1 \leq i \leq n$, where $x_1, x_2, \ldots, x_n$ are the parameters of f.

Finally, we define the sets of *governors* and *rulers* guarding a subexpression e' at position p of e. The idea is to collect the conditions of the `if` statements in e containing e', thereby telling us the conditions under which e' is reached when executing e. Our definitions are synonymous with those in [59].

Definition 2.1.5. Given an expression e and $p \in Pos(e)$, the *governors of* $e|_p$ is the set $gov(e, p)$ defined recursively as follows.

- $gov(e, \epsilon) = \{\}$,
- $gov((\texttt{if}\ \ e_1\ \ e_2\ \ e_3), 1q) = gov(e_1, q)$,
- $gov((\texttt{if}\ \ e_1\ \ e_2\ \ e_3), 2q) = \{e_1\} \cup gov(e_2, q)$,
- $gov((\texttt{if}\ \ e_1\ \ e_2\ \ e_3), 3q) = \{(\texttt{not}\ \ e_1)\} \cup gov(e_3, q)$,
- For all other cases, $gov(e, iq) = gov(e|_i, q)$.

Rulers are a subset of the governors that represent only "top-level" `if` statements.

Definition 2.1.6. Given an expression, e, and $p \in Pos(e)$, the *rulers* for p in e are defined recursively as follows:

- $rulers(e, \epsilon) = \{\}$.
- $rulers(e, (n+1)p) = rulers(e|_{n+1}, p)[x_i \mapsto e_i]_{i=1}^{n}$
  when $e = (\texttt{let}\ \ ((x_1\ \ e_1)\ \ \ldots\ \ (x_n\ \ e_n))\ \ e_{n+1})$.
- $rulers((\texttt{if}\ \ e_1\ \ e_2\ \ e_3), 1q) = rulers(e_1, q)$.
- $rulers((\texttt{if}\ \ e_1\ \ e_2\ \ e_3), 2q) = \{e_1\} \cup rulers(e_2, q)$.
- $rulers((\texttt{if}\ \ e_1\ \ e_2\ \ e_3), 3q) = \{(\texttt{not}\ \ e_1)\} \cup rulers(e_3, q)$.

- $rulers(e, p) = \{\}$ for all other e and p.

For example, in the expression `(if p (if q 0 (+ (if r x y) z)) 0)`, the governors of `x` are $\{$`p`, `(not q)`, `r`$\}$, but the rulers are just $\{$`p`, `(not q)`$\}$. Governors therefore give a more accurate picture of what conditions must hold for a subexpression to be executed.

When reasoning about governors and rulers, we want to know when they are all satisfied, so we know when the corresponding subexpression is executed. We therefore define the more general notion of when a set of expressions *holds*.

Definition 2.1.7. We say a set of expressions, E, *holds* for environment ϵ, denoted $\mathcal{H}^h \llbracket E \rrbracket \epsilon$, if $\bigwedge_{e \in E}(\llbracket e \rrbracket^h \epsilon \notin \{\texttt{nil}, \bot\})$.

2.1.1 Proving Termination with Measures

The standard method for proving termination of function definitions in a language such as FL is based on the concept of *well-foundedness*. Informally, a relation, $\prec$ is well-founded over a set, S, if there is no infinite sequence of decreasing values in S: $s_1 \succ s_2 \succ \ldots$. An example of this is the standard $<$ relation over the natural numbers. A more formal definition and discussion of well-foundedness will appear in Section 4.1.

Let d be a set of function definitions that we want to prove terminating, and F be the names of the functions defined in d. Let $m : F \rightarrow Expr$ map each function, f, to an expression over the parameters of f, as defined in d. Such a function is known as a *measure* for d. Using this notion of a measure, termination can be proven by the following theorem.

Theorem 2.1.1. *The functions of d are terminating, if there exists a set S and relation $\prec$ that is well-founded over S such that the following conditions hold.*

- $\langle \forall f \in F, \epsilon \in Env :: \llbracket m(f) \rrbracket^h \epsilon \rangle \in S$.
- *For all $f \in F$, $\epsilon \in Env$, and $p \in Pos(e^f)$ such that $e = e^f|_p$ is a function call of the form $(g\ e_1\ \ldots\ e_{ar(g)})$ to a function, $g \in F$, the following is true: $\mathcal{H}^h \llbracket gov(e^f, p) \rrbracket \epsilon \Rightarrow \llbracket m(f) \rrbracket^h \epsilon \succ \llbracket m(g)\sigma_e \rrbracket^h \epsilon$.*

In other words, we need to show that m maps each $f \in F$ to an expression that always evaluates to S, and that every time a recursive call is made, the measure goes down in value

```
(defun fact (x)
  (if (zp x)
      1
    (* x (fact (-
x 1)))))
```

Figure 2: Simple termination example: `fact`.

by some well-founded relation, $\prec$. Recall that the recursive call is executed exactly when the governors for that call hold. Therefore, we need to show that whenever the governors hold, the measure decreases.

Consider, as an example, the recursive definition of `fact`, a function that returns the factorial of its argument. Here, `(zp x)` is true when `x` is *not* a positive integer. Therefore, the definition says that if `x` is not a positive integer, we return 1, otherwise, we multiply `x` by the result of applying `fact` recursively to the result of subtracting 1 from `x`. Consider the measure that maps f to `(nfix x)`, which returns 0 if `x` is not a positive integer, and the value of `x` otherwise. Note that `(nfix x)` always maps `x` to the set of natural numbers, for which $<$ is well-founded.

The set of governors of the recursive call to `fact` is $\{$`(zp x)`$\}$. So, our second proof obligation is to show that, when $\mathcal{H}^h \llbracket\{$`(zp x)`$\}\rrbracket\, \epsilon$ is true, $\llbracket$`(nfix x)`$\rrbracket^h\, \epsilon > \llbracket$`(nfix (- x 1))`$\rrbracket^h\, \epsilon$. When the governors hold, `x` is a positive integer, and therefore $\llbracket$`(nfix x)`$\rrbracket^h\, \epsilon = \llbracket$`x`$\rrbracket^h\, \epsilon$ and $\llbracket$`(nfix (- x 1))`$\rrbracket^h\, \epsilon = \llbracket$`(- x 1)`$\rrbracket^h\, \epsilon$. But $\llbracket$`(- x 1)`$\rrbracket^h\, \epsilon$ is trivially less than $\llbracket$`x`$\rrbracket^h\, \epsilon$, which completes our proof.

The ordinals play an important part in termination proofs such as this one. This is because any set, S, with a well-founded ordering, $\prec$, can be embedded into the ordinals in an order preserving way. This means that any measure can be expressed in terms of the ordinals. In Part II of this dissertation, we develop a constructive theory of ordinal arithmetic, and use it to create a library for automating proofs involving the ordinals. The goal is to more fully automate measure-based termination proofs. Part III of this dissertation is concerned with developing a new termination analysis in order to fully automate termination proofs. We implement both of these components in the ACL2 theorem proving system, of which we now give an overview.

2.2 ACL2

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." It is the name of a programming language, a first-order mathematical logic based on recursive functions, and a mechanical theorem prover for that logic. In this section, we give a brief overview of the central features to ACL2 that are pertinent to this dissertation. A more thorough description can be found in the following sources [56, 57, 55]. Those readers who are not interested in the ACL2-specific aspects of this dissertation may wish to skip this section and refer to it as necessary later.

2.2.1 Language

ACL2's programming language can be thought of as a superset of FL (presented in Section 2.1), or as an applicative subset of Common Lisp. ACL2 is executable: terms composed entirely of defined functions and constants can be reduced to constants by Lisp calculation. This is important to many applications. For example, ACL2 models of commercial floating-point designs have been executed on millions of test cases to "validate" the models against industrial design simulation tools, before subjecting the ACL2 models to proof [106]. ACL2 models of microprocessors have been executed at 90% of the speed of comparable C simulation models [48].

The ACL2 programming language is a feature-rich and modern one, with many convenient features – *e.g.*, single-threaded objects, which are key to the fast execution mentioned above. However, many of these features are extra-logical. That is, they are implemented in such a way that the core language semantics are unaffected. Therefore, proving the termination of ACL2 functions basically reduces to proving the termination of functions in FL.

2.2.2 Logic

As a mathematical logic, ACL2 may be thought of as first-order predicate calculus with equality, recursive function definitions, and mathematical induction. The primitives of applicative Common Lisp are axiomatized, as are the basic data types, including natural

numbers, integers, rationals, complex rationals, ordered pairs, symbols, characters, and strings. ACL2 includes a representation of the ordinals up to ε_0 and the principle of mathematical induction, in ACL2, is stated as a rule of inference that allows induction up to ε_0. A principle of definition is also provided, by which the user can extend the axioms by the addition of equations defining new function symbols. To admit a new recursive definition, the principle requires the identification of an ordinal measure function and a proof that the arguments to every recursive call decrease according to this measure, as presented in Section 2.1.1. Only terminating recursive definitions can be so admitted under the definitional principle. (However, "partial functions" can be axiomatized; see [68, 69].)

2.3 Theorem Prover

As a theorem prover, ACL2 is an industrial-strength version of the Boyer-Moore theorem prover [13]. Of special note is its "industrial-strength," *e.g.*, it has been used to prove some of the largest and most complicated theorems ever proved about commercially designed digital artifacts [83, 104, 103, 105, 106, 19, 49]. The theorem prover is an integrated system of *ad hoc* proof techniques that include simplification, generalization, induction, and many other techniques. Simplification is the main technique and includes: (1) the use of evaluation (*i.e.*, the explicit computation of constants when, in the course of symbolic manipulation, certain variable-free expressions, like `(expt 2 32)`, arise), (2) conditional rewrite rules (derived from previously proved lemmas), (3) definitions (including recursive definitions), (4) propositional calculus (implemented both by the normalization of if-then-else expressions and the use of BDDs), (5) a linear arithmetic decision procedure for the rationals, (6) user-defined equivalence and congruence relations, (7) user-defined and mechanically verified simplifiers (meta-reasoning), (8) a user-extensible type system, (9) forward chaining, (10) an interactive loop for entering proof commands, and (11) various means to control and monitor these features including heuristics, interactive features, and user-supplied functional programs. See [56, 55] or the documentation, source code and examples at the URL [57] for details.

2.4 The ACL2 Regression Suite

Another key feature of ACL2 with regard to this dissertation is a large and diverse collection of user-submitted libraries known as the regression suite. This suite contains over 100 megabytes of code, including function definitions and theorems for reasoning about those functions, all of which have been verified by ACL2. These libraries include submissions from ACL2's world-wide user base over the course of several decades. The regression suite contains such diverse projects as:

- M5, which is a nearly-complete executable operational model of the Java Virtual Machine [84, 77]. Included in the model are support for 138 bytecode instructions, the creation and initialization of objects in the heap, the manipulation of static and instance fields, static, special, and virtual method invocation, inheritance and method resolution, multi-threading, and synchronization via thread monitors.

- A verified theorem prover based on the Otter prover [79]. A variant of Otter provided the first proof of Robbins Problem [78], which was introduced in the early 1930s and remained open until solved by McCune in the 1970s [50].

- Libraries used to verify the correctness of the floating point operations in AMD's flagship processors at the register-transfer level [82, 101, 102, 107].

- A large collection of algorithms for sorting lists and manipulating data structures. For example, there is a library verifying an in-place quicksort algorithm [95]. There is also a library verifying and analyzing the complexity of an efficient red-black-tree implementation in ACL2 [41].

- A verified model checker for the μ-calculus [67].

- A sophisticated and efficient implementation of a unification algorithm using term dags [99, 100].

- Sophisticated arithmetic libraries for reasoning about such topics as numerical arithmetic [53, 62], matrix algebra [31], Euclidean domains [32].

The regression suite, therefore, is a representative cross section of typical ACL2 usage. It reflects the wide variety of work done using ACL2, including projects of industrial scope. It is therefore a perfect testbed for new changes and additions to the ACL2 system, and are used as such by the developers of ACL2. One of the contributions of this dissertation is that we provide extensive experimental validation of our automatic termination analysis by running it on the ACL2 regression suite.

2.5 Bibliographic Notes

More on ACL2 including tutorials and collections of papers related to ACL2 can be found on the ACL2 website [57]. The book *Computer-Aided Reasoning: An Approach* [56] is another excellent resource for learning how to use ACL2. For more on the applications of ACL2, see *Computer-Aided Reasoning: ACL2 Case Studies* [55].

ACL2 is part of the Boyer-Moore family of theorem provers, for which ACL2's authors recently received the prestigious ACM Software System Award. Previous recipients of this award include Unix, TCP/IP, TeX, and the World-Wide Web.

2.6 Summary

In this chapter, we have introduced the semantics of a core applicative first-order language, which we call FL. We have discussed the traditional measure-based approach to proving the termination of programs in this domain. We have also introduced the ACL2 theorem proving system, in which the work of this dissertation is implemented and validated. We discussed its programming language, logic, theorem prover, and regression suite.

PART II

Ordinal Arithmetic

CHAPTER III

OVERVIEW

Despite the fact that ordinals have been studied and used extensively by various communities for over 100 years, we have not been able to find a comprehensive treatment of arithmetic on ordinal notations. The *ordinal arithmetic problem* for a notational system denoting the ordinals up to some ordinal δ, is as follows: given α and β, expressions in the system denoting ordinals $< \delta$, is γ the expression corresponding to $\alpha \star \beta$, where $\star$ can be any of $+, -, \cdot$, exponentiation? Solving this problem amounts to defining algorithms for ordinal arithmetic on the notation system in question. The practical implications of a solution to the ordinal arithmetic problem is that it allows users of theorem proving systems such as ACL2 to think and reason about ordinals algebraically. Algebraic reasoning is more convenient and powerful than the previously available options, which required users to use the underlying representation of ordinals to both define measure functions and to reason about them.

In this part of the dissertation, we present a solution to the ordinal arithmetic problem for a notational system denoting the ordinals up to ε_0. We describe how we have used this solution to create a powerful library of theorems for reasoning algebraically about these ordinals.

We begin by reviewing the theory of the ordinals, including arithmetic operators and notations for representing countable ordinals in Chapter 4. We present our ordinal arithmetic algorithms in Chapter 5, along with a correctness proof and complexity analysis for each algorithm.

In Chapter 6, describe how we have used these algorithms to build a powerful framework for mechanically reasoning about the theory of ordinal arithmetic. We discuss our implementation of our algorithms in ACL2, using a new representation of the ordinals up to ε_0 that is exponentially more succinct than ACL2's previous notation. We present our work

on mechanically verifying this implementation which we achieved by verifying that they satisfy well-known algebraic properties of ordinal arithmetic. We have altered ACL2 to use our ordinal notation, and show that our changes do not affect the soundness of the ACL2 logic by exhibiting a bijection between our ordinal representation and the previous ACL2 representation, using ACL2 version 2.7. These modifications appear in ACL2 starting in version 2.8, which also includes a library of definitions and theorems that we engineered to significantly automate reasoning involving the ordinals. With our library, users can ignore representational issues and can work with the ordinals in an algebraic setting. We give two case studies demonstrating that our ordinal arithmetic library gives adequate support for legacy code while enabling users to more easily verify new ordinal-related results in ACL2.

CHAPTER IV

PRELIMINARIES: THE ORDINALS

4.1 Set Theoretic Ordinals

We review the theory of ordinals, beginning with a brief overview of orderings.

Definition 4.1.1. A *totally ordered set*, or *toset*, is a pair $\langle S, \prec \rangle$ where S is a set, and $\prec$ is a binary relation on S for which the following properties hold.

- Irreflexivity (*i.e.*, $\neg(a \prec a)$),
- Asymmetry (*i.e.*, if $a \prec b$ then $\neg(b \prec a)$,
- Transitivity (*i.e.*, if $a \prec b$ and $b \prec c$ then $a \prec c$),
- Comparability (*i.e.*, for any $a, b \in S$ such that $a \neq b$, either $a \prec b$ or $b \prec a$.

In some texts, this is referred to as a *strictly totally ordered set*. Next, we define well-foundedness, which is a central concept in reasoning about termination.

Definition 4.1.2. A *well-founded relation*, R, on a set, S, is a binary relation such that there is no infinite sequence $s_1, s_2, \ldots$ of elements of S such that, for all $i \geq 1$, $s_{i+1} R s_i$.

Definition 4.1.3. A *well-ordered set*, or *woset*, is a toset whose order is well-founded.

We now move to *ordinals*, which are a particular class of wosets defined entirely using sets.

Definition 4.1.4. An *ordinal* is a woset, $\langle X, \prec \rangle$, such that $\forall x \in X, x = \{y \in X \mid y \prec x\}$

Notice that all of the elements of an ordinal are also ordinals by the same ordering, $\prec$. Also, it follows from the definition that $\prec \equiv \in \equiv \subset$. That is, $x \prec y$ if and only if $x \in y$ if and only if $x \subset y$. For the rest of this dissertation, we use lower case Greek letters to denote ordinals and $<$ or $\in$ 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 denote the ordinal to which it is isomorphic as $Ord(X,\prec)$. Note that every well-founded relation can be extended in an order-preserving way to a woset. In this way, the theory of the ordinals is the most general setting possible for proving termination.

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

4.2 Ordinal Arithmetic

In this section we define addition, subtraction, multiplication, and exponentiation for the ordinals. After each definition, we list various well-known properties.

Definition 4.2.1. $\alpha + \beta = Ord(A, <_A)$ where $A = (\{0\} \times \alpha) \cup (\{1\} \times \beta)$ and $<_A$ is the lexicographic ordering on A.

Ordinal addition satisfies the following properties.

$$\begin{aligned}
&\alpha + 1 \;=\; \alpha' \\
&(\alpha + \beta) + \gamma \;=\; \alpha + (\beta + \gamma) && \text{(associativity)} \\
&\beta < \gamma \;\Rightarrow\; \alpha + \beta < \alpha + \gamma && \text{(strict right monotonicity)} \\
&\beta < \gamma \;\Rightarrow\; \beta + \alpha \leq \gamma + \alpha && \text{(weak left monotonicity)}
\end{aligned}$$

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

Definition 4.2.2. $\alpha - \beta$ is defined to be 0 if $\alpha \leq \beta$, otherwise, it is the unique ordinal, ξ such that $\beta + \xi = \alpha$.

Definition 4.2.3. $\alpha \cdot_o \beta = Ord(A, <_A)$ where $A = \beta \times \alpha$ and $<_A$ is the lexicographic ordering on A.

Ordinal multiplication satisfies the following properties.

$$
\begin{array}{lll}
0 < n < \omega \;\Rightarrow\; n \cdot \omega = \omega & & \\
(\alpha \cdot \beta) \cdot \gamma \;=\; \alpha \cdot (\beta \cdot \gamma) & & \text{(associativity)} \\
(0 < \alpha \;\wedge\; \beta < \gamma) \;\Rightarrow\; \alpha \cdot \beta < \alpha \cdot \gamma & & \text{(strict right monotonicity)} \\
\beta < \gamma \;\Rightarrow\; \beta \cdot \alpha \leq \gamma \cdot \alpha & & \text{(weak left monotonicity)} \\
\alpha \cdot (\beta + \gamma) \;=\; (\alpha \cdot \beta) + (\alpha \cdot \gamma) & & \text{(left distributivity)}
\end{array}
$$

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

Definition 4.2.4. Given any ordinal, α, exponentiation is defined using transfinite recursion: $\alpha^0 = 1$, $\alpha^{\beta+1} = \alpha^\beta \cdot \alpha$, and for β a limit ordinal, $\alpha^\beta = \bigcup_{0<\xi<\beta} \alpha^\xi$.

Ordinal exponentiation satisfies the following properties, where an additive principal ordinal is an ordinal, β such that $\forall \alpha < \beta$, $\alpha + \beta = \beta$ (such ordinals always have the form ω^γ for some ordinal, $\gamma > 0$).

$$
\begin{array}{lll}
1 < p < \omega \;\Rightarrow\; p^\omega = \omega & & \\
\alpha^\beta \cdot \alpha^\gamma \;=\; \alpha^{\beta+\gamma} & & \\
(\alpha^\beta)^\gamma \;=\; \alpha^{\beta \cdot \gamma} & & \\
\alpha < \omega^\beta \;\Rightarrow\; \alpha + \omega^\beta = \omega^\beta & & \text{(additive principal property)} \\
\alpha, \beta < \omega^\gamma \;\Rightarrow\; \alpha + \beta < \omega^\gamma & & \text{(closure of additive principal ordinals)} \\
1 < \alpha \;\wedge\; \beta < \gamma \;\Rightarrow\; \alpha^\beta < \alpha^\gamma & & \text{(strict right monotonicity)} \\
\beta < \gamma \;\Rightarrow\; \beta^\alpha \leq \gamma^\alpha & & \text{(weak left monotonicity)}
\end{array}
$$

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.

4.3 Representation

Our representation deals with the ordinals less than ε_0. It is based upon the Cantor Normal Form for ordinals, which we now define.

Theorem 4.3.1. *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 $\alpha \in \varepsilon_0$, we have that $\alpha < \omega^\alpha$, since ε_0 is the smallest ε-ordinal. Thus, we can add the restriction that $\alpha > \alpha_1$ for these ordinals. This is essentially the representation of ordinals used in ACL2. However, since $\omega^\alpha \cdot k + \omega^\alpha = \omega^\alpha \cdot (k+1)$ and $n \in \omega$, we can collect like terms and rewrite the normal form as follows.

Corollary 4.3.1. (Cantor Normal Form) *For every ordinal $\alpha \in \varepsilon_0$, there are unique $n, p \in \omega$, $\alpha_1 > \cdots > \alpha_n > 0$, and $x_1, \ldots, x_n \in \omega - \{0\}$ such that $\alpha > \alpha_1$ and $\alpha = \omega^{\alpha_1} x_1 + \cdots + \omega^{\alpha_n} x_n + p$.*

By the *size* of an ordinal under a representation, we mean the number of bits needed to denote the ordinal in that representation.

Lemma 4.3.1. *The ordinal representation in Cor. 4.3.1 is exponentially more succinct than the representation in Thm. 4.3.1.*

Proof. Consider $\omega \cdot k$: it requires $O(k)$ bits with the representation in Thm. 4.3.1 and $O(\log k)$ bits with the representation in Cor. 4.3.1. □

We use nested triples to represent our ordinals. These triples are denoted by square brackets, with commas delimiting the elements in the triple. Thus the triple containing `a`, `b`, and `c` appears as [`a`, `b`, `c`]. CNF(α) denotes our representation of the ordinal α. If $\alpha \in \omega$, then CNF(α) = α. Otherwise, α has a unique decomposition, $\alpha = \sum_{i=1}^{n} \omega^{\alpha_i} x_i + p$. When this is the case,

$$\text{CNF}(\alpha) = [\text{CNF}(\alpha_1), x_1, \text{CNF}(\sum_{i=2}^{n} \omega^{\alpha_i} x_i + p)]$$

We now define several basic functions for manipulating ordinals in our notation. Some of our functions are partial, *i.e.*, they are not specified for all inputs. In such cases, we never use them outside of their intended domain. **finp**(`a`) returns true if `a` is a natural number, and false if it is an infinite ordinal. **triplep**(`x`) returns true if `x` is of the form [`a`, `b`, `c`]. **fe**, **fco**, and **rst** return the first exponent, first coefficient, and rest of an ordinal, respectively.

If $\mathbf{finp}(\mathtt{a})$, $\mathbf{fe}(\mathtt{a}) = 0$, $\mathbf{fco}(\mathtt{a}) = \mathtt{a}$, and **rst** is not used on $\mathtt{a}$. For an infinite ordinal of the form $[\mathtt{a}, \mathtt{b}, \mathtt{c}]$, $\mathbf{fe}([\mathtt{a}, \mathtt{b}, \mathtt{c}]) = \mathtt{a}$, $\mathbf{fco}([\mathtt{a}, \mathtt{b}, \mathtt{c}]) = \mathtt{b}$, and $\mathbf{rst}([\mathtt{a}, \mathtt{b}, \mathtt{c}]) = \mathtt{c}$.

4.4 Correctness and Complexity Concerns

In the following chapters, we define algorithms for ordinal arithmetic and analyze their correctness and complexity. In this section, we provide a high-level overview and explain what exactly is entailed and what assumptions we make.

Taken together, the correctness proofs establish that the structure consisting of the set-theoretic ordinals up to ε_0 with the usual arithmetic operations, is isomorphic to the structure consisting of E_0, the set of expressions corresponding to ordinals in our representation, along with the corresponding arithmetic operations (for which we provide algorithms). The set-theoretic structure is $\langle \varepsilon_0, \mathit{cmp}, +, -, \cdot, \mathit{exp} \rangle$, where *exp* is ordinal exponentiation and *cmp* is a function that orders ordinals: given ordinals α and β, it returns *lt* if $\alpha < \beta$, *gt* if $\alpha > \beta$ and *eq* if $\alpha = \beta$. The other structure is $\langle E_0, \mathbf{cmp_o}, +_o, -_o, \cdot_o, \mathbf{exp_o} \rangle$, where the intended meaning of the functions should be clear. Showing that the two structures are isomorphic involves first exhibiting a bijection between ε_0 and E_0; a trivial consequence of results in the previous section is that CNF is such a bijection. Secondly, the proof requires showing that the corresponding functions are equivalent. To this end, we show that: $\mathit{cmp}(\alpha, \beta) = \mathbf{cmp_o}(\mathrm{CNF}(\alpha), \mathrm{CNF}(\beta))$, $\mathrm{CNF}(\alpha \star \beta) = \mathrm{CNF}(\alpha) \star_o \mathrm{CNF}(\beta)$, where $\star$ ranges over $\{+, -, \cdot\}$, and, lastly, $\mathrm{CNF}(\mathit{exp}(\alpha, \beta)) = \mathbf{exp_o}(\mathrm{CNF}(\alpha), \mathrm{CNF}(\beta))$.

Note that these proofs are not mechanically verified. To do so would require using a theorem prover that can reason both about ACL2 and set theory. But, we implement the algorithms in ACL2 and reason about the implementations. For example, we prove that the implementations terminate and that they satisfy the numerous properties that their set-theoretic counterparts satisfy. We also develop a powerful library for reasoning about the ordinals. The details are in Chapter 6.

For our complexity analysis, we assume that integers require constant space and that integer operations have constant running time. One can later account for the integer operations by using the fastest known algorithms. This approach allows us to focus on the

```
|a|  {the length of a}
   finp(a)   :  0
   true      :  1 + |rst(a)|
```

```
#a  {the size of a}
   finp(a)   :  1
   true      :  #fe(a) + #rst(a)
```

Figure 3: The length and size functions used for complexity analysis.

interesting aspects of our algorithms, namely the aspects pertaining to the ordinal representations. To make explicit that arithmetic operations are being applied to integers, we refer to the usual arithmetic operations on integers as $<_w$, $+_\omega$, $-_\omega$, $\cdot_\omega$, and exp_ω.

The complexity of the ordinal arithmetic algorithms is given in terms of the functions in Figure 3. In the figure, we use a sequence of *condition* : *result* forms to define functions: the conditions should be read from top to bottom until a condition that holds is found and then the corresponding result is returned. Note that the `true` condition always holds. We use this format for definitions throughout this Volume.

4.5 Bibliographic Notes

Proofs of the properties listed in Section 4.2 can be found in texts on set theory [36, 63, 108]. The Cantor Normal Form for ordinals is discussed in [108]. A discussion of the ordinals in ACL2 can be found in [56, 57].

4.6 Summary

In this chapter, we have given an overview of the set-theoretic ordinals. This included a definition of ordinals and definitions of ordinal addition, subtraction, multiplication, and exponentiation. We also presented Cantor's Normal Form for ordinals and shown how to use it to make a succinct notation for the ordinals less than ε_0. Finally, we discussed what we mean by correctness and complexity with regards to the analysis of our ordinal arithmetic algorithms.

CHAPTER V

ORDINAL ARITHMETIC: ALGORITHMS

In this chapter, we define algorithms to compare, recognize, add, subtract, multiply, and raise to ordinal powers the ordinals in ε_0.

5.1 Comparing Ordinals

In this section, we present functions that compare ordinals according to the standard total ordering over the ordinals. These functions are given in Figure 4 on the following page.

In the sequel, the ordinals α and β have the following Cantor normal form decompositions $\alpha = \sum_{i=1}^{n} \omega^{\alpha_i} x_i + p$ and $\beta = \sum_{i=1}^{m} \omega^{\beta_i} y_i + q$; in addition, $\mathtt{a}$, $\mathtt{a}_i$, $\mathtt{b}$, and $\mathtt{b}_j$ denote CNF(α), CNF(α_i), CNF(β), and CNF(β_j), respectively, for all $1 \leq i \leq n$ and $1 \leq j \leq m$.

We start with $\mathbf{cmp_o}$, the comparison function for ordinals corresponding to *cmp*. In Figure 4 on the next page we also define $<_o$, $\leq_o$, and $=_o$. These functions are not needed and not used most of the time in our algorithms, where for efficiency reasons we use $\mathbf{cmp_o}$ most of the time. The definition of $-_o$ (page 30) provides a nice example of where this is useful, as instead of computing both $\mathbf{fe}(\mathtt{a}) <_o \mathbf{fe}(\mathtt{b})$ and $\mathbf{fe}(\mathtt{a}) >_o \mathbf{fe}(\mathtt{b})$, we only compute $\mathbf{cmp_o}(\mathbf{fe}(\mathtt{a}), \mathbf{fe}(\mathtt{b}))$. The reason for including $<_o$, $\leq_o$, and $=_o$ is to make the presentation clearer, and we also use $>_o$ and $\geq_o$ where we find them useful.

Theorem 5.1.1. *For all* $\alpha, \beta \in \varepsilon_0$, $\mathbf{cmp_o}(\mathtt{a}, \mathtt{b}) = cmp(\alpha, \beta)$.

Proof. The proof is by induction on the sizes of $\mathtt{a}$ and $\mathtt{b}$. The base case, where $\mathbf{finp}(\mathtt{a})$ or $\mathbf{finp}(\mathtt{b})$ holds, is straightforward.

For the induction step, we have that $\#\mathtt{a}, \#\mathtt{b} > 1$ and for all γ, δ if $\#\text{CNF}(\gamma) < \#\mathtt{a}$ and $\#\text{CNF}(\delta) < \#\mathtt{b}$, then $\mathbf{cmp_o}(\text{CNF}(\gamma), \text{CNF}(\delta)) = cmp(\gamma, \delta)$. There are 3 cases.

In the first, $\mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1) \neq eq$. If $\mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1) = lt$, then by the induction hypothesis, $\alpha_1 < \beta_1$. Thus $\omega^{\alpha_1} < \omega^{\beta_1}$. Thus, since $\sum_{i=2}^{n} \omega^{\alpha_i} x_i < \omega^{\alpha_1}$, $\alpha < \omega^{\beta_1} \leq \beta$ by the closure of additive principal ordinals under addition. Therefore, $cmp(\alpha,\beta) = lt = \mathbf{cmp_o}(\mathtt{a}, \mathtt{b})$. By a

```
cmp_ω(p,q)   {ordering on naturals}
   p <_ω q   :  lt
   q <_ω p   :  gt
   true      :  eq

cmp_o(a,b)   {ordering on ordinals}
   finp(a)  ∧  finp(b)            :  cmp_ω(a,b)
   finp(a)                        :  lt
   finp(b)                        :  gt
   cmp_o(fe(a),fe(b)) ≠ eq        :  cmp_o(fe(a),fe(b))
   cmp_ω(fco(a),fco(b)) ≠ eq      :  cmp_ω(fco(a),fco(b))
   true                           :  cmp_o(rst(a),rst(b))

a <_o b   {< for ordinals}
   cmp_o(a,b) = lt   :  true
   true              :  false

a ≤_o b   {≤ for ordinals}
   cmp_o(a,b) = gt   :  false
   true              :  true

a =_o b   {= for ordinals}
   cmp_o(a,b) = eq   :  true
   true              :  false
```

Figure 4: The ordinal ordering algorithms.

similar argument, $\mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1) = gt \quad \equiv \quad cmp(\alpha,\beta) = \mathbf{cmp_o}(\mathtt{a},\mathtt{b}) = gt$.

In the next case, we have that $\mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1) = eq \ \wedge \ \mathbf{cmp}_\omega(x_1, y_1) \neq eq$. By induction hypothesis, $\alpha_1 = \beta_1$. Suppose that $\mathbf{cmp}_\omega(x_1, y_1) = lt$. Again, we note that $\sum_{i=2}^{n} \omega^{\alpha_i} x_i < \omega^{\alpha_1}$. Thus $\alpha < \omega^{\alpha_1} x_1 + \omega^{\alpha_1}$, by the strict right monotonicity of ordinal addition. But then we have

$$\omega^{\alpha_1} x_1 + \omega^{\alpha_1} = \omega^{\alpha_1}(x_1 + 1) = \omega^{\beta_1}(x_1 + 1) \leq \omega^{\beta_1} y_1$$

Hence, $\alpha < \beta$, so $\mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1) = lt \ = cmp(\alpha, \beta)$. A similar argument establishes the case where $\mathbf{cmp}_\omega(x_1, y_1) = gt$.

In the final case, we have that $\mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1) = eq \quad \wedge \quad \mathbf{cmp}_\omega(x_1, y_1) = eq$. By the induction hypothesis, this means $\alpha_1 = \beta_1$. If $\mathbf{cmp_o}(\mathbf{rst}(\mathtt{a}), \mathbf{rst}(\mathtt{b})) = eq$, then by the induction hypothesis, $\sum_{i=2}^{n} \omega^{\alpha_i} x_i + p = \sum_{i=2}^{n} \omega^{\beta_i} y_i + q$ and we have $cmp(\alpha, \beta) = eq \ = \mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1)$.

If $\mathbf{cmp_o}(\mathbf{rst}(\mathtt{a}), \mathbf{rst}(\mathtt{b})) = lt$, then by the induction hypothesis, $\sum_{i=2}^{n} \omega^{\alpha_i} x_i + p < \sum_{i=2}^{n} \omega^{\beta_i} y_i + q$; hence we have $\alpha < \beta$. Therefore, $cmp(\alpha, \beta) = lt \ = \mathbf{cmp_o}(\mathtt{a}_1, \mathtt{b}_1)$. A

```
op(a)   {ordinal recognizer}
  finp(a)   :  a ∈ ω
  true      :  triplep(a)
               ∧ fco(a) ∈ ω
               ∧ 0 <ω fco(a)
               ∧ op(fe(a))
               ∧ op(rst(a))
               ∧ fe(rst(a)) <o fe(a)
```

Figure 5: The ordinal recognizer algorithm.

similar argument establishes the case where $\mathbf{cmp_o}(\mathbf{rst}(\mathtt{a}), \mathbf{rst}(\mathtt{b})) = gt$. □

Theorem 5.1.2. $\mathbf{cmp_o}(\mathtt{a}, \mathtt{b})$ *runs in time* $O(\min(\#\mathtt{a}, \#\mathtt{b}))$.

Proof. In the worst case we simultaneously recur down a and b. In more detail, the complexity of this function is bounded by the recurrence relation

$$T(\mathtt{a}, \mathtt{b}) = \begin{cases} c, & \text{if } \mathbf{finp}(\mathtt{a}) \text{ or } \mathbf{finp}(\mathtt{b}) \\ T(\mathtt{a}_1, \mathtt{b}_1) + T(\mathbf{rst}(\mathtt{a}), \mathbf{rst}(\mathtt{b})) + c, & \text{otherwise} \end{cases}$$

for some constant value, c. It now follows by induction on the size of a and b that $T(\mathtt{a}, \mathtt{b}) \leq k \cdot \min(\#\mathtt{a}, \#\mathtt{b}) - t$ for any constants, k, t, such that $t \geq c$ and $k \geq c + t$. □

5.2 Recognizing Ordinals

In this section we present and analyze the function that recognizes ordinals in our notation. The definitions are given in Figure 5.

The definition is a straightforward implementation of the Cantor Normal Form representation presented in Section 4.3. At first glance it seems that the complexity is quadratic as **op** calls $<_o$ at every level of recursion. However, a closer examination reveals the following.

Theorem 5.2.1. $\mathbf{op}(\mathtt{a})$ *runs in time* $O(\#\mathtt{a}(\log \#\mathtt{a}))$.

Proof. The running time is bounded by the (non-linear) recurrence relation

$$T(\mathtt{a}) = \begin{cases} c, & \text{if } \mathbf{finp}(\mathtt{a}) \\ T(\mathtt{a}_1) + T(\mathbf{rst}(\mathtt{a})) + \min(\#\mathtt{a}_1, \#\mathbf{rst}(\mathtt{a})) + c, & \text{otherwise} \end{cases}$$

for some constant, c, by Thm. 5.1.2. We show by induction on $\#\mathtt{a}$, that $T(\mathtt{a}) \leq k(\#\mathtt{a})(\log \#\mathtt{a}) + t$ where k, t are constants such that $t \geq c$ and $k \geq 3t$. In the base case, we have $T(\mathtt{a}) = c \leq t$.

```
a +_o b  {ordinal addition}
   finp(a) ∧ finp(b)   :  a +_ω b
   fe(a) <_o fe(b)     :  b
   fe(a) =_o fe(b)     :  [fe(a), fco(a) +_ω fco(b), rst(b)]
   true                :  [fe(a), fco(a), rst(a) +_o b]
```

Figure 6: The ordinal addition algorithm.

For the induction step, let $x = \min(\#\mathtt{a}_1, \#\mathbf{rst}(\mathtt{a}))$ and $y = \max(\#\mathtt{a}_1, \#\mathbf{rst}(\mathtt{a}))$. Note that $x + y = \#\mathtt{a}$. We have:

$$\begin{aligned}
& T(\mathtt{a}) \\
\{\, \textit{Definition of } T \,\} \quad &= \quad T(\mathtt{a}_1) + T(\mathbf{rst}(\mathtt{a})) + x + c \\
\{\, \textit{Induction Hypothesis} \,\} \quad &\leq \quad kx \log x + t + ky \log y + t + x + c \\
\{\, kx \geq 2t + x \ \textit{as}\ k \geq 3t \,\} \quad &\leq \quad k(x \log x + y \log y + x) + c \\
\{\, \textit{Log} \,\} \quad &= \quad k \log(x^x y^y 2^x) + c \\
\{\, \langle \forall z \in \omega :: 2^z \leq \tbinom{2z}{z} \rangle \,\} \quad &\leq \quad k \log(x^x y^y \tbinom{2x}{x}) + c \\
\{\, x \leq y \,\} \quad &\leq \quad k \log(x^x y^y \tbinom{x+y}{x}) + c \\
\{\, \textit{Binomial Theorem} \,\} \quad &\leq \quad k \log\left((x+y)^{x+y}\right) + c \\
\{\, t \geq c,\ x + y = \#\mathtt{a} \,\} \quad &= \quad k(\#\mathtt{a}) \log(\#\mathtt{a}) + t
\end{aligned}$$

□

5.3 Ordinal Addition

The algorithm for ordinal addition is given in Figure 6. The main idea of the algorithm is to traverse `b` until an exponent is found that is $\leq$ the first exponent of `a`. We now prove the correctness of the algorithm and analyze its complexity.

Theorem 5.3.1. *For all* $\alpha, \beta \in \varepsilon_0$ $\mathrm{CNF}(\alpha + \beta) = \mathtt{a} +_o \mathtt{b}$.

Proof. The proof is by induction on α. The key insight (as it was for the proof of Thm. 5.1.1) is that $\sum_{i=2}^{n} \omega^{\alpha_i} x_i < \omega^{\alpha_1}$.

If $\alpha, \beta \in \omega$, then $\mathrm{CNF}(\alpha + \beta) = \mathtt{a} +_o \mathtt{b}$. Now suppose that $\beta > \omega$ and either $\alpha \in \omega$ or $\alpha_1 < \beta_1$. Then:

$$
\begin{array}{rcl}
 & & \alpha + \beta \\
\{\, \textit{Definition of } \beta\textit{, arithmetic} \,\} & = & \alpha + \omega^{\beta_1}(1 + y_1 - 1) + \sum_{i=2}^{m} \omega^{\beta_i} y_i + q \\
\{\, \textit{Left distributivity} \,\} & = & \alpha + \omega^{\beta_1} + \omega^{\beta_1}(y_1 - 1) + \sum_{i=2}^{m} \omega^{\beta_i} y_i + q \\
\{\, \textit{Additive principal property} \,\} & = & \omega^{\beta_1} + \omega^{\beta_1}(y_1 - 1) + \sum_{i=2}^{m} \omega^{\beta_i} y_i + q \\
\{\, \textit{Definition of } \beta \,\} & = & \beta
\end{array}
$$

Next, suppose that $\alpha, \beta > \omega$ and $\alpha_1 = \beta_1$. Then:

$$
\begin{array}{rcl}
 & & \alpha + \beta \\
\{\, \textit{Definition of } \alpha \,\} & = & \omega^{\alpha_1} x_1 + \sum_{i=2}^{n} \omega^{\alpha_i} x_i + p + \beta \\
\{\, \textit{Additive principal property} \,\} & = & \omega^{\alpha_1} x_1 + \beta \\
\{\, \textit{Definition of } \beta\textit{, distributivity} \,\} & = & \omega^{\alpha_1}(x_1 + y_1) + \sum_{i=2}^{m} \omega^{\beta_i} y_i + q
\end{array}
$$

Note that $\mathrm{CNF}(\omega^{\alpha_1}(x_1 + y_1) + \sum_{i=2}^{m} \omega^{\beta_i} y_i + q) = [\mathtt{a}_1, \mathtt{x}_1 +_o \mathtt{y}_1, \mathbf{rst}(\mathtt{b})]$, which matches the definition of $+_o$.

In the final case, we have that $\mathtt{a}_1 <_o \mathtt{b}_1$ and $\neg\mathbf{finp}(\mathtt{a})$. Now, $\alpha + \beta = \omega^{\alpha_1} x_1 + \delta$, where $\delta = \sum_{i=2}^{n} \omega^{\alpha_i} x_i + p + \beta$. Since $\sum_{i=2}^{n} \omega^{\alpha_i} x_i + p < \omega^{\alpha_1}$ and $\beta < \omega^{\alpha_1}$, $\delta < \omega^{\alpha_1}$. Letting $\sum_{i=1}^{k} \omega^{\delta_i} z_i + r$ be the Cantor normal form decomposition of δ, we see that $\mathrm{CNF}(\alpha + \beta)$ $= \mathrm{CNF}(\omega^{\alpha_1} x_1 + \sum_{i=1}^{k} \omega^{\delta_i} z_i + r)$, which by the induction hypothesis is $[\mathtt{a}_1, \mathtt{x}_1, \mathbf{rst}(\mathtt{a}) +_o \mathtt{b}]$. □

Theorem 5.3.2. $\mathtt{a} +_o \mathtt{b}$ *runs in time* $O(\min(\#\mathtt{a},\ |\mathtt{a}| \cdot \#\mathtt{b}_1))$.

Proof. The running time of $\mathtt{a} +_o \mathtt{b}$ is given by the recurrence relation

$$
T(\mathtt{a}, \mathtt{b}) = \begin{cases} c, & \text{if } \mathbf{finp}(\mathtt{a}) \\ T(\mathbf{rst}(\mathtt{a}), \mathtt{b}) + k_1 \min(\#\mathtt{a}_1, \#\mathtt{b}_1) + c, & \text{otherwise} \end{cases}
$$

for some constants c and k_1, using Thm. 5.1.2. We use induction to show that $T(\mathtt{a}, \mathtt{b}) \leq$ $k \cdot \min(\#\mathtt{a}, |\mathtt{a}| \cdot \#\mathtt{b}_1) + c$, where k is a constant such that $k \geq k_1 + c$. In the base case, $T(\mathtt{a}, \mathtt{b}) = c = k \cdot \min(\#\mathtt{a}, |\mathtt{a}| \cdot \#\mathtt{b}_1) + c$, since $|\mathtt{a}| = 0$. Otherwise, using the induction hypothesis, we have:

$$
\begin{array}{l}
T(\mathtt{a}, \mathtt{b}) \\
= \ \{\, \textit{Definition of } T \,\} \\
T(\mathbf{rst}(\mathtt{a}), \mathtt{b}) + k_1 \min(\#\mathtt{a}_1, \#\mathtt{b}_1) + c \\
\leq \ \{\, \textit{Inductive Hypothesis} \,\}
\end{array}
$$

$\mathtt{a} -_o \mathtt{b}$ {*ordinal subtraction*}

$\mathbf{finp}(\mathtt{a}) \wedge \mathbf{finp}(\mathtt{b}) \wedge \mathtt{a} \leq_\omega \mathtt{b}$	:	$\mathtt{0}$
$\mathbf{finp}(\mathtt{a}) \wedge \mathbf{finp}(\mathtt{b})$	:	$\mathtt{a} -_\omega \mathtt{b}$
$\mathbf{fe}(\mathtt{a}) <_o \mathbf{fe}(\mathtt{b})$	:	$\mathtt{0}$
$\mathbf{fe}(\mathtt{a}) >_o \mathbf{fe}(\mathtt{b})$	:	$\mathtt{a}$
$\mathbf{fco}(\mathtt{a}) <_\omega \mathbf{fco}(\mathtt{b})$	:	$\mathtt{0}$
$\mathbf{fco}(\mathtt{a}) >_\omega \mathbf{fco}(\mathtt{b})$	:	$[\mathbf{fe}(\mathtt{a}), \mathbf{fco}(\mathtt{a}) -_\omega \mathbf{fco}(\mathtt{b}), \mathbf{rst}(\mathtt{a})]$
$\mathtt{true}$	:	$\mathbf{rst}(\mathtt{a}) -_o \mathbf{rst}(\mathtt{b})$

Figure 7: The ordinal subtraction algorithm.

$$
\begin{aligned}
& k \cdot \min(\#\mathbf{rst}(\mathtt{a}), |\mathbf{rst}(\mathtt{a})| \cdot \#\mathtt{b}_1) + c + k_1 \min(\#\mathtt{a}_1, \#\mathtt{b}_1) + c \\
& \leq \ \{\, \textit{Arithmetic, } k \geq k_1 + c \,\} \\
& k \cdot \min(\#\mathtt{a}_1 + \#\mathbf{rst}(\mathtt{a}), |\mathbf{rst}(\mathtt{a})| \cdot \#\mathtt{b}_1 + \#\mathtt{b}_1) + c \\
& = \ \{\, \textit{Definition of } \# \,\} \\
& k \cdot \min(\#\mathtt{a}, |\mathtt{a}| \cdot \#\mathtt{b}_1) + c
\end{aligned}
$$

□

5.4 Subtraction

We now turn our attention to ordinal subtraction; our algorithm is given in Figure 7. Recall that $\alpha - \beta$ is defined to be 0 if $\alpha < \beta$ and otherwise to be the unique ordinal, ξ such that $\beta + \xi = \alpha$. One must be careful to avoid silly mistakes when subtraction involves infinite ordinals, *e.g.*, note that $(\omega + 1) - 1 \neq \omega$.

Theorem 5.4.1. *For all* $\alpha, \beta \in \varepsilon_0$, $\mathrm{CNF}(\alpha - \beta) = \mathtt{a} -_o \mathtt{b}$.

Proof. It is easy to prove, using induction, that if $\alpha < \beta$, then $\mathtt{a} -_o \mathtt{b} = 0$.

When $\alpha \geq \beta$, the proof amounts to showing that $\mathtt{b} +_o (\mathtt{a} -_o \mathtt{b}) = \mathtt{a}$ and $\mathtt{a} -_o \mathtt{b}$ is in proper CNF form, and is by induction on $\#\mathtt{a}$ and $\#\mathtt{b}$. If $\beta = \alpha$ this is trivial, since $\mathtt{b} +_o (\mathtt{a} -_o \mathtt{b}) = \mathtt{a}$. We now focus on the case where $\beta < \alpha$.

If $\mathbf{finp}(\mathtt{a})$ and $\mathbf{finp}(\mathtt{b})$, then $\mathtt{b} +_o (\mathtt{a} -_o \mathtt{b}) = \mathtt{a}$. If $\mathtt{b}_1 < \mathtt{a}_1$, then $\mathtt{b} +_o (\mathtt{a} -_o \mathtt{b}) = \mathtt{b} +_o \mathtt{a} = \mathtt{a}$. If $\mathtt{b}_1 = \mathtt{a}_1$ and $y_1 < x_1$, we have:

```
a *_o b  {ordinal multiplication}
   a = 0   ∨  b = 0     :  0
   finp(a)  ∧  finp(b)  :  a ·_ω b
   finp(b)              :  [fe(a), fco(a) ·_ω b, rst(a)]
   true                 :  [fe(a) +_o fe(b), fco(b), a *_o rst(b)]
```

Figure 8: A first attempt at ordinal multiplication.

$$
\begin{aligned}
& & \mathtt{b} +_o (\mathtt{a} -_o \mathtt{b}) \\
\{\, \textit{Definition of } -_o \,\} & = & \mathtt{b} +_o [\mathtt{a}_1, x_1 - y_1, \mathbf{rst}(\mathtt{a})] \\
\{\, \textit{Definition of } +_o \,\} & = & [\mathtt{a}_1, x_1 - y_1 + y_1, \mathbf{rst}(\mathtt{a})] \\
\{\, \textit{Arithmetic} \,\} & = & [\mathtt{a}_1, x_1, \mathbf{rst}(\mathtt{a})] \\
\{\, \textit{Definition of } \mathtt{a} \,\} & = & \mathtt{a}
\end{aligned}
$$

Also, note that $\mathbf{op}(\mathtt{a} -_o \mathtt{b})$ since $\mathbf{op}(\mathtt{a})$ and $x_1 > y_1$; hence, $x_1 - y_1 > 0$.

Finally, suppose $\mathtt{b}_1 = \mathtt{a}_1$ and $y_1 = x_1$; then $\mathbf{rst}(\mathtt{b}) < \mathbf{rst}(\mathtt{a})$. We now have:

$$
\begin{aligned}
& & \mathtt{b} +_o (\mathtt{a} -_o \mathtt{b}) \\
\{\, \textit{Definition of } -_o, +_o \,\} & = & [\mathtt{b}_1, y_1, \mathbf{rst}(\mathtt{b}) +_o (\mathbf{rst}(\mathtt{a}) -_o \mathbf{rst}(\mathtt{b}))] \\
\{\, \textit{Ind. hypothesis, } x_1 = y_1,\ \mathtt{a}_1 = \mathtt{b}_1 \,\} & = & \mathtt{a}
\end{aligned}
$$

□

Theorem 5.4.2. $\mathtt{a} -_o \mathtt{b}$ *runs in time* $O(\min(\#\mathtt{a}, \#\mathtt{b}))$.

Proof. The recursion relation for the complexity of this function is

$$
T(\mathtt{a}, \mathtt{b}) = \begin{cases} c, & \text{if } \mathbf{finp}(\mathtt{a}) \text{ or } \mathbf{finp}(\mathtt{b}) \\ k_1 \cdot \min(\#\mathtt{a}_1, \#\mathtt{b}_1) + T(\mathbf{rst}(\mathtt{a}), \mathbf{rst}(\mathtt{b})) + c, & \text{otherwise} \end{cases}
$$

for some constants, k_1, c. The proof that $T(\mathtt{a}, \mathtt{b}) \leq k \cdot \min(\#\mathtt{a}, \#\mathtt{b})$ is almost identical to that of Thm. 5.1.2. □

5.5 Ordinal Multiplication

A first attempt at defining multiplication is given in Figure 8. Later in this section we derive a more efficient (and more complicated) algorithm, but its correctness depends on the correctness of $*_o$, which we now consider.

Lemma 5.5.1. *For all* $\alpha, \beta \in \varepsilon_0$, $x, y \in \omega$ *such that* $\beta, x > 0$, $\omega^\alpha x \cdot \omega^\beta y = \omega^{\alpha+\beta} y$.

Proof. $\omega^\alpha x \cdot \omega^\beta y \;=\; \omega^\alpha(x \cdot \omega^\beta)y \;=\; \omega^\alpha \cdot \omega^\beta y \;=\; \omega^{\alpha+\beta} y$ □

Theorem 5.5.1. *For all* $\alpha, \beta \in \varepsilon_0$, $\mathrm{CNF}(\alpha \cdot \beta) = \mathtt{a} *_o \mathtt{b}$.

Proof. The proof is by (transfinite) induction on β. The case where $\alpha \in \omega$ and $\beta \in \omega$ is straightforward. The remainder of the proof consists of two cases: $\beta < \omega$ and $\beta \geq \omega$.

If $\beta < \omega$, then $\beta > 0$ and $\alpha > \omega$. The base case, where $\beta = 1$ is straightforward. For the induction step we have:

$$\begin{aligned}
& & \mathrm{CNF}(\alpha \cdot \beta) \\
\{\, \textit{Subtraction, distributivity} \,\} & = & \mathrm{CNF}(\alpha + (\alpha \cdot (\beta - 1))) \\
\{\, \textit{Thm. 5.3.1, induction hypothesis} \,\} & = & \mathtt{a} +_o (\mathtt{a} *_o \mathrm{CNF}(\beta - 1)) \\
\{\, \textit{Definition of } *_o,\ \beta < \omega \,\} & = & \mathtt{a} +_o [\mathtt{a}_1, x_1 \cdot_\omega (\beta -_\omega 1), \mathbf{rst}(\mathtt{a})] \\
\{\, \textit{Definition of } +_o \,\} & = & [\mathtt{a}_1, x_1 + (x_1 \cdot_\omega (\beta - 1)), \mathbf{rst}(\mathtt{a})] \\
\{\, \textit{Distributivity of } \cdot_\omega \,\} & = & [\mathtt{a}_1, x_1 \cdot_\omega \beta, \mathbf{rst}(\mathtt{a})] \\
\{\, \textit{Definition of } *_o \,\} & = & \mathtt{a} *_o \mathtt{b}
\end{aligned}$$

For the final case we have that $\beta \geq \omega$ and $\alpha > 0$. First, we note that $\alpha \cdot \omega^{\beta_1} y_1 = \omega^{\alpha_1 + \beta_1} y_1$:

$$\begin{aligned}
& & \alpha \cdot \omega^{\beta_1} y_1 \\
\{\, \textit{Weak left monotonicity of multiplication} \,\} & \leq & \omega^{\alpha_1}(x_1 + 1) \cdot \omega^{\beta_1} y_1 \\
\{\, \textit{Lem. 5.5.1} \,\} & = & \omega^{\alpha_1 + \beta_1} y_1 \\
\{\, \textit{Lem. 5.5.1} \,\} & = & \omega^{\alpha_1} x_1 \cdot \omega^{\beta_1} y_1 \\
\{\, \textit{Weak left monotonicity of multiplication} \,\} & \leq & \alpha \cdot \omega^{\beta_1} y_1
\end{aligned}$$

Finally, we have:

$$\begin{aligned}
& & \mathrm{CNF}(\alpha \cdot \beta) \\
\{\, \textit{Distributivity} \,\} & = & \mathrm{CNF}(\omega^{\alpha_1 + \beta_1} y_1 + (\alpha \cdot \textstyle\sum_{i=2}^{m} \omega^{\beta_i} y_i)) \\
\{\, \textit{Thm. 5.3.1} \,\} & = & \mathrm{CNF}(\omega^{\alpha_1 + \beta_1} y_1) +_o \mathrm{CNF}(\alpha \cdot \textstyle\sum_{i=2}^{m} \omega^{\beta_i} y_i) \\
\{\, \textit{Def. of } \mathrm{CNF}\textit{, ind. hyp.} \,\} & = & [\mathrm{CNF}(\alpha_1 + \beta_1), y_1, 0] +_o \mathtt{a} *_o \mathrm{CNF}(\textstyle\sum_{i=2}^{m} \omega^{\beta_i} y_i) \\
\{\, \textit{Def. of } \mathbf{rst}\textit{, Thm. 5.3.1} \,\} & = & [\mathtt{a}_1 +_o \mathtt{b}_1, y_1, 0] +_o \mathtt{a} *_o \mathbf{rst}(\mathtt{b}) \\
\{\, \mathbf{fe}(\mathtt{a} *_o \mathbf{rst}(\mathtt{b})) <_o \mathtt{a}_1 + \mathtt{b}_1 \,\} & = & [\mathtt{a}_1 +_o \mathtt{b}_1, y_1, \mathtt{a} *_o \mathbf{rst}(\mathtt{b})] \\
\{\, \textit{Definition of } *_o \,\} & = & \mathtt{a} *_o \mathtt{b}
\end{aligned}$$

□

```
dropn(a,n)  {n is a natural number}
   n = 0   :  a
   true    :  dropn(rst(a),n-1)

c1(a,b)  {finds the index of the first exponent of a that is ≤ b_1}
   fe(a) >_o fe(b)   :  1 +_ω c1(rst(a),b)
   true              :  0

c2(a,b,n)  {skips over the first n elements of a and then calls c1}
   true   :  n + c1(dropn(a,n),b)

padd(a, b, n)  {skips over the first n elements of a and adds the rest
to b}
   n = 0   :  a +_o b
   true    :  [fe(a), fco(a), padd(rst(a),b,n - 1)]

pmult(a,b,n)  {pseudo-multiplication}
   a = 0  ∨  b = 0      :  0
   finp(a)  ∧  finp(b)  :  a ·_ω b
   finp(b)              :  [fe(a), fco(a) ·_ω b, rst(a)]
   true                 :  [padd(fe(a),fe(b),m),
                            fco(b),
                            pmult(a,rst(b),m)]
                                 where m = c2(fe(a),fe(b),n)

a ·_o b  {quicker ordinal multiplication}
   true   :  pmult(a,b,0)
```

Figure 9: An efficient algorithm for ordinal multiplication.

The problem with this definition is its running time. Note that this algorithm walks down $\mathtt{b}$, adding $\mathtt{a}_1$ to each exponent of $\mathtt{b}$. This is equivalent to adding some ordinal, $\mathtt{c}$ to a decreasing sequence of ordinals $(\mathtt{d}_1, \mathtt{d}_2, \ldots, \mathtt{d}_n)$. Using the addition algorithm, we find that for each $\mathtt{d}_i$, $\mathbf{fe}(\mathtt{d}_i)$ is compared to each exponent of $\mathtt{c}$ until the first exponent of $\mathtt{c}$ such that $\mathbf{fe}(\mathtt{d}_i)$ is $\geq$ this exponent is found. But since the $\mathtt{d}_i$'s are decreasing, we know that $\mathbf{fe}(\mathtt{d}_i) \geq \mathbf{fe}(\mathtt{d}_{i+1})$. Therefore, if the jth exponent of $\mathtt{c}$ is $> \mathbf{fe}(\mathtt{d}_i)$, we know that it is $> \mathbf{fe}(\mathtt{d}_{i+1})$. This means that simply adding each element of the decreasing sequence to $\mathtt{c}$ is inefficient. If we can keep track of how many exponents of $\mathtt{c}$ we went through before adding $\mathtt{d}_i$, we can just skip over those when we add $\mathtt{d}_{i+1}$. These observations lead to the definitions in Figure 9, which provide a quicker way to compute multiplication.

Lemma 5.5.2. $\mathtt{d} <_o \mathtt{b} \quad \Rightarrow \quad \mathbf{c1}(\mathtt{a}, \mathtt{b}) \leq \mathbf{c1}(\mathtt{a}, \mathtt{d})$.

Proof. The proof is by induction on $\mathbf{c1}(\mathtt{a},\mathtt{b})$. If $\mathbf{c1}(\mathtt{a}, \mathtt{b}) = 0$, then this is trivially true since

c1 always returns a natural number. In the induction step, we have that $\mathbf{c1}(\mathtt{a},\mathtt{b}) > 0$ and $\mathtt{d} <_o \mathtt{b}$. Thus, $\mathbf{fe}(\mathtt{d}) \leq_o \mathtt{b}_1 <_o \mathtt{a}_1$ by the definitions of $<_o$ and **c1**. Finally, by the induction hypothesis, $\mathbf{c1}(\mathtt{a},\mathtt{b}) = 1 + \mathbf{c1}(\mathbf{rst}(\mathtt{a}),\mathtt{b}) \leq 1 + \mathbf{c1}(\mathbf{rst}(\mathtt{a}),\mathtt{d}) = \mathbf{c1}(\mathtt{a},\mathtt{d})$. □

Lemma 5.5.3. $n \leq \mathbf{c1}(\mathtt{a},\mathtt{b}) \quad \Rightarrow \quad \mathbf{c1}(\mathtt{a},\mathtt{b}) = \mathbf{c2}(\mathtt{a},\mathtt{b},n)$.

Lemma 5.5.4. $\mathbf{padd}(\mathtt{a},\mathtt{b},\mathbf{c1}(\mathtt{a},\mathtt{b})) = \mathtt{a} +_o \mathtt{b}$.

Proof. The proof is by induction on $\mathbf{c1}(\mathtt{a},\mathtt{b})$. If $\mathbf{c1}(\mathtt{a},\mathtt{b}) = 0$, $\mathtt{b}_1 \leq_o \mathtt{a}_1$, so the lemma is clearly true. In the induction step, $\mathbf{c1}(\mathtt{a},\mathtt{b}) > 0$, so $\mathtt{b}_1 <_o \mathtt{a}_1$ by the definition of **c1**. Hence, $\mathbf{c1}(\mathbf{rst}(\mathtt{a}),\mathtt{b}) < \mathbf{c1}(\mathtt{a},\mathtt{b})$. By the induction hypothesis, we know that $\mathbf{padd}(\mathtt{d},\mathtt{b},\mathbf{c1}(\mathtt{d},\mathtt{b})) = \mathtt{d} +_o \mathtt{b}$ for all $\mathtt{d} <_o \mathtt{a}$. Thus

$$
\begin{array}{rcl}
 & & \mathtt{a} +_o \mathtt{b} \\
\{\ \textit{Definition of } +_o\ \} & = & [\mathtt{a}_1, \mathtt{x}_1, \mathbf{rst}(\mathtt{a}) +_o \mathtt{b}] \\
\{\ \textit{Induction Hypothesis}\ \} & = & [\mathtt{a}_1, \mathtt{x}_1, \mathbf{padd}(\mathbf{rst}(\mathtt{a}), \mathtt{b}, \mathbf{c1}(\mathbf{rst}(\mathtt{a}), \mathtt{b}))] \\
\{\ \textit{Definition of } \mathbf{c1}\ \} & = & [\mathtt{a}_1, \mathtt{x}_1, \mathbf{padd}(\mathbf{rst}(\mathtt{a}), \mathtt{b}, \mathbf{c1}(\mathtt{a},\mathtt{b}) - 1)] \\
\{\ \textit{Definition of } \mathbf{padd}\ \} & = & \mathbf{padd}(\mathtt{a},\mathtt{b},\mathbf{c1}(\mathtt{a},\mathtt{b}))
\end{array}
$$

□

Theorem 5.5.2. $n \leq \mathbf{c1}(\mathtt{a}_1,\mathtt{b}_1) \quad \Rightarrow \quad \mathbf{pmult}(\mathtt{a},\mathtt{b},n) = \mathtt{a} *_o \mathtt{b}$.

Proof. The proof is by induction on $|\mathtt{b}|$. If $\mathbf{finp}(\mathtt{b})$, then this is clearly true. For the induction step, we know $\neg(\mathbf{finp}(\mathtt{b}))$. The induction hypothesis tells us that for all $\mathtt{d}$ such that $|\mathtt{d}| < |\mathtt{b}|, n \leq \mathbf{c1}(\mathbf{fe}(\mathtt{d}),\mathtt{b}_1) \Rightarrow \mathbf{pmult}(\mathtt{d},\mathtt{b},n) = \mathtt{d} *_o \mathtt{b}$. Suppose $n \leq \mathbf{c1}(\mathtt{a}_1,\mathtt{b}_1)$. Then if we let $m = \mathbf{c2}(\mathtt{a}_1,\mathtt{b}_1,n)$, we have that $m = \mathbf{c1}(\mathtt{a}_1,\mathtt{b}_1)$ by Lem. 5.5.3. Therefore, we also know that $m \leq \mathbf{c1}(\mathtt{a}_1,\mathbf{fe}(\mathbf{rst}(\mathtt{b})))$ by Lem. 5.5.2. Thus, by the Induction Hypothesis we have the following.

$$
\begin{array}{rcl}
 & & \mathbf{pmult}(\mathtt{a},\mathtt{b},n) \\
\{\ \textit{Definition of } \mathbf{pmult}\ \} & = & [\mathbf{padd}(\mathtt{a}_1,\mathtt{b}_1,m), \mathtt{y}_1, \mathbf{pmult}(\mathtt{a},\mathbf{rst}(\mathtt{b}),m)] \\
\{\ \textit{Lem. 5.5.4}\ \} & = & [\mathtt{a}_1 +_o \mathtt{b}_1, \mathtt{y}_1, \mathbf{pmult}(\mathtt{a},\mathbf{rst}(\mathtt{b}),m)] \\
\{\ \textit{Induction hypothesis}\ \} & = & [\mathtt{a}_1 +_o \mathtt{b}_1, \mathtt{y}_1, \mathtt{a} *_o \mathbf{rst}(\mathtt{b})] \\
\{\ \textit{Definition of } *_o\ \} & = & \mathtt{a} *_o \mathtt{b}
\end{array}
$$

□

Corollary 5.5.1. *For all* $\alpha, \beta \in \varepsilon_0$, $\mathrm{CNF}(\alpha \cdot \beta) = \mathtt{a} \cdot_o \mathtt{b}$.

Proof. Follows directly from Theorems 5.5.1 on page 32 and 5.5.2 on the preceding page. □

We now turn our attention to complexity issues. For the next lemmas and theorem, let $\mathtt{a}_1 = [\mathtt{d}_1, z_1, [\mathtt{d}_2, z_2, \ldots [\mathtt{d}_k, z_k, \mathtt{r}] \ldots]]$.

Lemma 5.5.5. $\mathbf{c1}(\mathtt{a}, \mathtt{b})$ *takes time* $O(\sum_{i=1}^{\mathbf{c1}(\mathtt{a},\mathtt{b})+1} \min(\#\mathtt{a}_i, \#\mathtt{b}_1))$.

Proof. In the worst case, we traverse $\mathtt{a}$, comparing $\mathtt{a}_i$ with $\mathtt{b}_1$. By Thm. 5.1.2, this takes $\mathrm{O}(\sum_{i=1}^{\mathbf{c1}(\mathtt{a},\mathtt{b})+1} \min(\#\mathtt{a}_i, \#\mathtt{b}_1))$ time. □

Lemma 5.5.6. $\mathbf{c2}(\mathtt{a}, \mathtt{b}, s)$ *takes time* $O(s + \sum_{i=s+1}^{\mathbf{c1}(\mathtt{a},\mathtt{b})+1} \min(\#\mathtt{a}_i, \#\mathtt{b}_1))$.

Lemma 5.5.7. $\mathbf{padd}(\mathtt{a}, \mathtt{b}, s)$ *runs in time* $O(\min(\#\mathbf{fe}(\mathbf{dropn}(\mathtt{a}, s)), \#\mathtt{b}_1) + s)$ *when* $s \geq \mathbf{c1}(\mathtt{a}, \mathtt{b})$.

Proof. Note that $\mathbf{fe}(\mathbf{dropn}(\mathtt{a}, s)) \leq \mathtt{b}_1$ since the exponents of $\mathtt{a}$ are decreasing and $s \geq \mathbf{c1}(\mathtt{a},\mathtt{b})$. Hence, $\mathbf{dropn}(\mathtt{a}, s) +_o \mathtt{b}$ requires one comparison and creates an answer in constant time. Therefore, by Thm. 5.1.2, **padd** takes $O(\min(\#\mathbf{fe}(\mathbf{dropn}(\mathtt{a}, s)), \#\mathtt{b}_1) + s)$ time. □

Theorem 5.5.3. $\mathbf{pmult}(\mathtt{a}, \mathtt{b}, s)$ *runs in time* $O(|\mathtt{a}_1||\mathtt{b}| + \#\mathbf{dropn}(\mathtt{a}_1, s) + \#\mathtt{b})$ *if* $s \leq \mathbf{c1}(\mathtt{a}_1, \mathtt{b}_1)$.

Proof. Let $m = \mathbf{c2}(\mathtt{a}_1, \mathtt{b}_1, s)$; then $m = \mathbf{c1}(\mathtt{a}_1, \mathtt{b}_1)$ by Lem. 5.5.3. Thus, using Lemmas 5.5.6 and 5.5.7, we can construct the following recurrence relation to bound the running time of **pmult**:

$$T(\mathtt{a}, \mathtt{b}, s) = \begin{cases} d, & \text{if } \mathbf{finp}(\mathtt{b}) \quad \vee \quad \mathtt{a} = 0 \\ T(\mathtt{a}, \mathbf{rst}(\mathtt{b}), m) + k_1(s + \sum_{i=s+1}^{m+1} \min(\#\mathtt{d}_i, \#\mathbf{fe}(\mathtt{b}_1))) & \\ \qquad + k_2(\min(\#\mathbf{fe}(\mathbf{dropn}(\mathtt{a}_1, m)), \#\mathtt{b}_1) + m) + d, & \text{otherwise} \end{cases}$$

for some constants k_1, k_2, and d. We use induction on $|\mathtt{b}|$ to show that $T(\mathtt{a}, \mathtt{b}, s) \leq k \cdot (|\mathtt{a}_1||\mathtt{b}| + \#\mathbf{dropn}(\mathtt{a}_1, s) + \#\mathtt{b})$ where $k \geq k_1 + k_2 + d$. This is true in the base case, because $\#\mathtt{b} = 1$ and $k \geq d$. For the induction step, we first note the following.

```
expt_o(a,b)  {ordinal exponentiation}
   b = 0                    :  1
   a = 0                    :  0
   finp(b)                  :  a ·_o expt_o(a,b −_ω 1)
   finp(a)  ∧  fe(b) = 1    :  [fco(b), 1, 0] ·_o exp_ω(a,rst(b))
   finp(a)                  :  [[fe(b) −_o 1, fco(b), 0], 1, 0] ·_o expt_o(a,rst(b))
   true                     :  [fe(a) ·_o [fe(b), fco(b), 0], 1, 0] ·_o expt_o(a,rst(b))
```

Figure 10: A first attempt at ordinal exponentiation.

$$
\begin{aligned}
& k_1[s + \textstyle\sum_{i=s+1}^{m+1} \min(\#\mathtt{d}_i, \#\mathbf{fe}(\mathtt{b}_1))] + k_2[\min(\#\mathbf{fe}(\mathbf{dropn}(\mathtt{a}, m)), \#\mathbf{fe}(\mathtt{b}_1)) + m] + d \\
& \leq \ \{\, \textit{Arithmetic, } m \geq s \,\} \\
& k_1(m + \textstyle\sum_{i=s+1}^{m} \#\mathtt{d}_i + \#\mathbf{fe}(\mathtt{b}_1)) + (k_2 + d)(\#\mathbf{fe}(\mathtt{b}_1) + m) \\
& \leq \ \{\, k \geq k_1 + k_2 + d \,\} \\
& k(m + \textstyle\sum_{i=s+1}^{m} \#\mathtt{d}_i + \#\mathbf{fe}(\mathtt{b}_1))
\end{aligned}
$$

Combining this with the recurrence relation and using the induction hypothesis, we have:

$$
\begin{aligned}
& T(\mathtt{a}, \mathtt{b}, s) \\
& \leq \ \{\, \textit{Definition of } T\textit{, earlier reasoning} \,\} \\
& T(\mathtt{a}, \mathbf{rst}(\mathtt{b}), m) + k(m + \textstyle\sum_{i=s+1}^{m} \#\mathtt{d}_i + \#\mathbf{fe}(\mathtt{b}_1)) \\
& \leq \ \{\, \textit{Induction Hypothesis} \,\} \\
& k(|\mathtt{a}_1||\mathbf{rst}(\mathtt{b})| + \#\mathbf{dropn}(\mathtt{a}_1, m) + \#\mathbf{rst}(\mathtt{b}) + m + \textstyle\sum_{i=s+1}^{m} \#\mathtt{d}_i + \#\mathbf{fe}(\mathtt{b}_1)) + d \\
& \leq \ \{\, \textit{Arithmetic, } m \leq |\mathtt{a}_1| \,\} \\
& k(|\mathtt{a}_1||\mathtt{b}| + \#\mathbf{dropn}(\mathtt{a}_1, m) + \textstyle\sum_{i=s+1}^{m} \#\mathtt{d}_i + \#\mathbf{rst}(\mathtt{b}) + \#\mathbf{fe}(\mathtt{b}_1)) + d \\
& = \ \{\, \textit{Definitions of } \#\textit{, } \mathbf{dropn} \,\} \\
& k(|\mathtt{a}_1||\mathtt{b}| + \#\mathbf{dropn}(\mathtt{a}_1, s) + \#\mathbf{rst}(\mathtt{b}) + \#\mathbf{fe}(\mathtt{b}_1)) + d \\
& < \ \{\, \#\mathbf{fe}(\mathtt{b}_1) < \#\mathtt{b}_1 \,\} \\
& k(|\mathtt{a}_1||\mathtt{b}| + \#\mathbf{dropn}(\mathtt{a}_1, s) + \#\mathtt{b}) + d
\end{aligned}
$$

□

Corollary 5.5.2. $\mathtt{a} \cdot_o \mathtt{b}$ *runs in time* $O(|\mathtt{a}_1||\mathtt{b}| + \#\mathtt{a}_1 + \#\mathtt{b})$.

5.6 Ordinal Exponentiation

As with ordinal multiplication, we begin with a relatively simple but inefficient definition for ordinal exponentiation. This algorithm is given in Figure 10 on the previous page. Before proving the correctness of this algorithm, we present the following two lemmas.

Lemma 5.6.1. *$\forall k, x \in \omega, \alpha \in \varepsilon_0$ such that $\alpha > 0$, $x > 0$, and $k > 1$, $k^{\omega^\alpha x} = \omega^{\omega^{\alpha-1} x}$*

Proof. $k^{\omega^\alpha x} = k^{\omega^{1+\alpha-1} x} = k^{\omega \cdot \omega^{\alpha-1} x} = (k^\omega)^{\omega^{\alpha-1} x} = \omega^{\omega^{\alpha-1} x}$ □

Lemma 5.6.2. *For all $\alpha, \xi, z \in \varepsilon_0$ such that $a \geq \omega$ and $\xi > 0$, $\alpha^{\omega^\xi z} = \omega^{\alpha_1 \omega^\xi z}$.*

Proof. $\alpha^{\omega^\xi z} \leq (\omega^{\alpha_1+1})^{\omega^\xi z} = \omega^{(\alpha_1+1)\omega^\xi z} = \omega^{\alpha_1 \omega^\xi z} = (\omega^{\alpha_1})^{\omega^\xi z} \leq \alpha^{\omega^\xi z}$ □

Using these lemmas, we prove the main correctness result.

Theorem 5.6.1. *For all $\alpha, \beta \in \varepsilon_0$,* $\mathbf{expt_o}(\mathtt{a}, \mathtt{b}) = \mathrm{CNF}(\alpha^\beta)$

Proof. The proof is by transfinite induction on β. For the base case, if $\beta = 0$, the proof is straightforward. The same is true when $\alpha = 0$, so assume that $\alpha > 0$. If β is finite, we have:

$$\begin{aligned}
& & \mathrm{CNF}(\alpha^\beta) \\
\{\, \textit{Property of exponentiation} \,\} & = & \mathrm{CNF}(\alpha \cdot \alpha^{\beta-1}) \\
\{\, \textit{Cor. 5.5.1} \,\} & = & \mathrm{CNF}(\alpha) \cdot_o \mathrm{CNF}(\alpha^{\beta-1}) \\
\{\, \textit{Induction Hypothesis} \,\} & = & \mathtt{a} \cdot_o \mathbf{expt_o}(\mathtt{a}, \mathtt{b} -_\omega 1) \\
\{\, \textit{Definition of } \mathbf{expt_o} \,\} & = & \mathbf{expt_o}(\mathtt{a}, \mathtt{b})
\end{aligned}$$

Next, suppose that $\beta = \omega^1 y + q$ and $\alpha < \omega$. In this case,

$$\begin{aligned}
& & \mathrm{CNF}(\alpha^\beta) \\
\{\, \textit{Current hypotheses} \,\} & = & \mathrm{CNF}(\alpha^{\omega^1 y + q}) \\
\{\, \textit{Lem. 5.6.1} \,\} & = & \mathrm{CNF}(\omega^y \cdot \alpha^q) \\
\{\, \textit{Cor. 5.5.1} \,\} & = & [\mathbf{fco}(\mathtt{b}), 1, 0] \cdot_o \mathbf{exp}_\omega(\mathtt{a}, \mathbf{rst}(\mathtt{b})) \\
\{\, \textit{Definition of } \mathbf{expt_o} \,\} & = & \mathbf{expt_o}(\mathtt{a}, \mathtt{b})
\end{aligned}$$

Now suppose that $\beta = \sum_{i=1}^m \omega^{\beta_i} y_i + q$, $\beta_1 > 1$, and $\alpha < \omega$. Then

$$
\begin{aligned}
& & \mathrm{CNF}(\alpha^{\beta}) \\
\{\, \textit{Current hypotheses} \,\} &= & \mathrm{CNF}(\alpha^{\sum_{i=1}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Property of exponentiation} \,\} &= & \mathrm{CNF}(\alpha^{\omega^{\beta_1} k_1} \cdot \alpha^{\sum_{i=2}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Lem. 5.6.1} \,\} &= & \mathrm{CNF}(\omega^{\omega^{\beta_1 - 1} k_1} \cdot \alpha^{\sum_{i=2}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Cor. 5.5.1} \,\} &= & [[\mathbf{fe}(b) -_o 1, \mathbf{fco}(b), 0], 1, 0] \cdot_o \mathrm{CNF}(\alpha^{\sum_{i=2}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Induction Hypothesis} \,\} &= & [[\mathbf{fe}(b) -_o 1, \mathbf{fco}(b), 0], 1, 0] \cdot_o \mathbf{expt_o}(a, \mathbf{rst}(b)) \\
\{\, \textit{Definition of } \mathbf{expt_o} \,\} &= & \mathbf{expt_o}(\mathtt{a}, \mathtt{b})
\end{aligned}
$$

Finally, consider the case where $\beta, \alpha > \omega$. In this case, we have the following.

$$
\begin{aligned}
& & \mathrm{CNF}(\alpha^{\beta}) \\
\{\, \textit{Current hypotheses} \,\} &= & \mathrm{CNF}(\alpha^{\sum_{i=1}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Property of exponentiation} \,\} &= & \mathrm{CNF}(\alpha^{\omega^{\beta_1} k_1} \cdot \alpha^{\sum_{i=2}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Lem. 5.6.2} \,\} &= & \mathrm{CNF}(\omega^{\alpha_1 \cdot \omega^{\beta_1} k_1} \cdot \alpha^{\sum_{i=2}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Cor. 5.5.1} \,\} &= & [\mathbf{fe}(a) \cdot_o [\mathbf{fe}(b), \mathbf{fco}(b), 0], 1, 0] \cdot_o \mathrm{CNF}(\alpha^{\sum_{i=2}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Induction Hypothesis} \,\} &= & [\mathbf{fe}(a) \cdot_o [\mathbf{fe}(b), \mathbf{fco}(b), 0], 1, 0] \cdot_o \mathbf{expt_o}(a, \mathbf{rst}(b)) \\
\{\, \textit{Definition of } \mathbf{expt_o} \,\} &= & \mathbf{expt_o}(\mathtt{a}, \mathtt{b})
\end{aligned}
$$

□

The efficient definition for ordinal exponentiation is more complex than that for ordinal multiplication, and in an effort to increase clarity, we define exponentiation ($\mathbf{exp_o}$) using four helper functions: $\mathbf{exp_1}$, $\mathbf{exp_2}$, $\mathbf{exp_3}$, and $\mathbf{exp_4}$. We introduce them one at a time, proving the correctness and complexity of each before moving on to the next. The correctness and complexity of $\mathbf{exp_o}$ come at the end and follow directly from the results proved for the helper functions. Before reading further, the reader may want to try a few examples; a particularly revealing class of examples is $(\omega + 1)^{\omega^\omega + k}$, where k ranges over the naturals.

The first helper function, $\mathbf{exp_1}$, is defined in Figure 11 on the following page, and it is used to raise a positive integer to an infinite ordinal power. We proceed by proving that it is correct and analyzing its complexity.

Lemma 5.6.3. *$\forall k \in \omega, \alpha \in \varepsilon_0$ such that $\alpha > 0$ and $k > 1$, $k^\alpha = (\omega^{\sum_{i=1}^{n} \omega^{\alpha_i - 1} x_i}) k^p$*

```
exp_1(k,a)  {raising a positive integer to an infinite ordinal power}
   fe(a) =_o 1    :  [fco(a), exp_ω(k,rst(a)), 0]
   finp(rst(a))   :  [[fe(a) −_o 1, fco(a), 0], exp_ω(k,rst(a)), 0]
   true           :  [[fe(a) −_o 1, 1, fe(c)], fco(c), 0]
                       where c = exp_1(k,rst(a))
```

Figure 11: Ordinal exponentiation: raising a positive integer to an infinite power.

Proof. Recall that the Cantor normal form decomposition of α is $\sum_{i=1}^{n} \omega^{\alpha_i} x_i + p$. The proof follows from Lem. 5.6.1. □

Theorem 5.6.2. *For all $k \in \omega, \alpha \in \varepsilon_0$ such that $\alpha \geq \omega$ and $k > 1$,* $\mathrm{CNF}(k^\alpha) = \mathbf{exp_1}(k, \mathtt{a})$.

Proof. The proof is by induction on α. If $\alpha_1 = 1$, then $\alpha = \omega \cdot x_1 + p$ for some $x_1, p \in \omega$. Thus, we have:

$$
\begin{aligned}
& & \mathrm{CNF}(k^\alpha) \\
\{\textit{ Definition of } \alpha \} & = & \mathrm{CNF}(k^{\omega \cdot x_1 + p}) \\
\{\textit{ Property of exponentiation} \} & = & \mathrm{CNF}(k^{\omega \cdot x_1} \cdot k^p) \\
\{\textit{ Lem. 5.6.1} \} & = & \mathrm{CNF}(\omega^{x_1} \cdot k^p) \\
\{\textit{ Cor. 5.5.1} \} & = & \mathrm{CNF}(\omega^{x_1}) \cdot_o k^p \\
\{\textit{ Definitions of } \mathrm{CNF}, \mathbf{exp}_\omega \} & = & [x_1, 1, 0] \cdot_o \mathbf{exp}_\omega(k, p) \\
\{\textit{ Definition of } \cdot_o \} & = & [x_1, \mathbf{exp}_\omega(k, p), 0] \\
\{\textit{ Definition of } \mathbf{exp_1} \} & = & \mathbf{exp_1}(k, \mathtt{a})
\end{aligned}
$$

Likewise, if $\alpha_1 > 1$ and $\mathbf{finp}(\mathbf{rst}(\mathtt{a}))$, then $\alpha = \omega^{\alpha_1} x_1 + p$ for some $\alpha_1 \in \varepsilon_0$, $x_1, p \in \omega$. We now have:

$$
\begin{aligned}
& & \mathrm{CNF}(k^\alpha) \\
\{\textit{ Definition of } \alpha \} & = & \mathrm{CNF}(k^{\omega^{\alpha_1} x_1 + p}) \\
\{\textit{ Property of exponentiation} \} & = & \mathrm{CNF}(k^{\omega^{\alpha_1} x_1} \cdot k^p) \\
\{\textit{ Lem. 5.6.1} \} & = & \mathrm{CNF}(\omega^{\omega^{\alpha_1 - 1} x_1} \cdot k^p) \\
\{\textit{ Cor. 5.5.1} \} & = & \mathrm{CNF}(\omega^{\omega^{\alpha_1 - 1} x_1}) \cdot_o k^p \\
\{\textit{ Definitions of } \mathrm{CNF}, \mathbf{exp}_\omega \} & = & [[\mathtt{a}_1 -_o 1, x_1, 0], 1, 0] \cdot_o \mathbf{exp}_\omega(k, p) \\
\{\textit{ Definition of } \cdot_o \} & = & [[\mathtt{a}_1 -_o 1, x_1, 0], \mathbf{exp}_\omega(k, p), 0] \\
\{\textit{ Definition of } \mathbf{exp_1} \} & = & \mathbf{exp_1}(k, \mathtt{a})
\end{aligned}
$$

```
exp_2(a,k)  {raising a limit ordinal to a positive integer}
   true   :  [fe(a) ·_o (k - 1), 1, 0] ·_o a

natpart(a)  {returns the natural part of an ordinal}
   finp(a)  :  a
   true     :  natpart(rst(a))

limitp(a)  {returns true if a represents a limit ordinal}
   true   :  op(a)  ∧  ¬finp(a)  ∧  natpart(a) = 0

limitpart(a)  {returns the greatest ordinal, b, such that limitp(b) and
b <_o a}
   finp(a)  :  0
   true     :  [fe(a), fco(a), limitpart(rst(a))]
```

Figure 12: Ordinal exponentiation: raising a limit ordinal to a positive integer.

In the final case $\neg\mathbf{finp}(\mathbf{rst}(\mathtt{a}))$ holds and by the induction hypothesis $\forall \xi < \alpha$, CNF(k^{ξ}) = $\mathbf{exp_1}(k,$ CNF$(\xi))$. Now, letting $\mathtt{c} = \mathbf{exp_1}(k, \mathbf{rst}(\mathtt{a}))$, we have:

$$
\begin{aligned}
& & & \mathrm{CNF}(k^{\alpha}) \\
& \{\ \textit{Lem. 5.6.3}\ \} & = & \mathrm{CNF}((\omega^{\sum_{i=1}^{m} \omega^{\alpha_i - 1} x_i}) k^{p}) \\
& \{\ \textit{Ordinal arithmetic}\ \} & = & \mathrm{CNF}(\omega^{\omega^{\alpha_1 - 1} x_1} (\omega^{\sum_{i=2}^{m} w^{\alpha_i - 1} x_i}) k^{p}) \\
& \{\ \textit{Lem. 5.6.3, } \cdot_o\ \} & = & \mathrm{CNF}(\omega^{\omega^{\alpha_1 - 1} x_1}) \cdot_o \mathrm{CNF}(k^{\sum_{i=2}^{m} w^{\alpha_i} x_i + p}) \\
& \{\ \textit{Ind. hyp.,}\ \mathrm{CNF}\ \} & = & [[\mathtt{a}_1 -_o 1, \mathtt{x}_1, 0], 1, 0] \cdot_o \mathbf{exp_1}(k, \mathbf{rst}(\mathtt{a})) \\
& \{\ \mathtt{c} = [\mathbf{fe}(\mathtt{c}), \mathbf{fco}(\mathtt{c}), 0]\ \} & = & [[\mathtt{a}_1 -_o 1, \mathtt{x}_1, 0], 1, 0] \cdot_o [\mathbf{fe}(\mathtt{c}), \mathbf{fco}(\mathtt{c}), 0] \\
& \{\ \textit{Definition of } \cdot_o\ \} & = & [[\mathtt{a}_1 -_o 1, \mathtt{x}_1, 0] +_o \mathbf{fe}(\mathtt{c}), \mathbf{fco}(\mathtt{c}), 0] \\
& \{\ \mathtt{a}_1 -_o 1 > \mathtt{a}_2 -_o 1\ \} & = & [[\mathtt{a}_1 -_o 1, \mathtt{x}_1, \mathbf{fe}(\mathtt{c})], \mathbf{fco}(\mathtt{c}), 0] \\
& \{\ \textit{Definition of } \mathbf{exp_1}\ \} & = & \mathbf{exp_1}(k, \mathtt{a})
\end{aligned}
$$

□

Theorem 5.6.3. $\mathbf{exp_1}$ *runs in time* $O(|\mathtt{a}|)$.

Proof. Note that by Thm. 5.4.2, computing $\mathtt{a}_1 -_o 1$ takes constant time. The proof is now straightforward. □

We now consider the second helper function, $\mathbf{exp_2}$, which is shown in Figure 12 and is used to raise a limit ordinal to a positive integer.

Lemma 5.6.4. *For all* $\mathtt{a}, \mathtt{b}$ *such that* $\mathbf{op}(\mathtt{a})$, $\mathbf{op}(\mathtt{b})$, $\mathbf{natpart}(\mathtt{b}) = 0$ *and* $\neg\mathbf{finp}(\mathtt{a})$, $\mathtt{a} \cdot_o \mathtt{b} = [\mathtt{a}_1, 1, 0] \cdot_o \mathtt{b}$.

Proof. The proof is by induction on $|\mathtt{b}|$. If $\mathbf{finp}(\mathtt{b})$, then $\mathtt{b} = 0$ and $\mathtt{a} \cdot_o \mathtt{b} = 0 = [\mathtt{a}_1, 1, 0] \cdot_o \mathtt{b}$. For the induction step we have:

$$\begin{aligned}
& & \mathtt{a} \cdot_o \mathtt{b} \\
\{\textit{ Definition of } \cdot_o \} & = & [\mathtt{a}_1 + \mathtt{b}_1, \mathtt{y}_1, \mathtt{a} \cdot_o \mathbf{rst}(\mathtt{b})] \\
\{\textit{ Induction Hypothesis} \} & = & [\mathtt{a}_1 + \mathtt{b}_1, \mathtt{y}_1, [\mathtt{a}_1, 1, 0] \cdot_o \mathbf{rst}(\mathtt{b})] \\
\{\textit{ Definition of } \cdot_o \} & = & [\mathtt{a}_1, 1, 0] \cdot_o \mathtt{b}
\end{aligned}$$

□

Theorem 5.6.4. *For all* $\alpha \in \varepsilon_0, k \in \omega$ *such that* $\alpha \geq \omega$, $\mathbf{limitp}(\mathtt{a})$, *and* $k > 1$, $\mathrm{CNF}(\alpha^k) = \mathbf{exp_2}(\mathtt{a}, k)$.

Proof. The proof is by induction on k. If $k = 2$, then $\mathrm{CNF}(\alpha^k) = \mathrm{CNF}(\alpha^2) = \mathrm{CNF}(\alpha \cdot \alpha) = \mathtt{a} \cdot_o \mathtt{a}$. Applying Lem. 5.6.4, we get $[\mathtt{a}_1, 1, 0] \cdot_o \mathtt{a} = \mathbf{exp_2}(\mathtt{a}, k)$. For the induction step we have:

$$\begin{aligned}
& & \mathrm{CNF}(\alpha^k) \\
\{\textit{ Ordinal arithmetic, Cor. 5.5.1} \} & = & \mathtt{a} \cdot_o \mathrm{CNF}(\alpha^{k-1}) \\
\{\textit{ Induction hypothesis} \} & = & \mathtt{a} \cdot_o \mathbf{exp_2}(\mathtt{a}, k-1) \\
\{\textit{ Definition of } \mathbf{exp_2} \} & = & \mathtt{a} \cdot_o [\mathtt{a}_1 \cdot_o (k-2), 1, 0] \cdot_o \mathtt{a} \\
\{\textit{ Definition of } \cdot_o \} & = & [\mathtt{a}_1 +_o (\mathtt{a}_1 \cdot_o (k-2)), 1, 0] \cdot_o \mathtt{a} \\
\{\textit{ Distr., Thm. 5.3.1, Cor. 5.5.1} \} & = & [\mathtt{a}_1 \cdot_o (k-1), 1, 0] \cdot_o \mathtt{a} \\
\{\textit{ Definition of } \mathbf{exp_2} \} & = & \mathbf{exp_2}(\mathtt{a}, k)
\end{aligned}$$

□

Theorem 5.6.5. $\mathbf{exp_2}(\mathtt{a}, k)$ *runs in time* $O(|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a})$

Proof. Note that $\mathtt{a}_1 \cdot_o (k-1)$ takes constant time, since $k - 1$ is of size 1. Also, note that $\#(\mathtt{a}_1 \cdot_o (k-1)) = \#\mathtt{a}_1$ and $|\mathtt{a}_1 \cdot_o (k-1)| = |\mathtt{a}_1|$. So, by Cor. 5.5.2, we have that the running time is $O(|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a})$. □

```
exp3h(a,p,n,k)  {helper function for exp3}
   k = 1   :  (a ·o p) +o p
   true    :  padd(exp2(a,k) ·o p, exp3h(a,p,n,k-1), n)

exp3(a,k)   {raising an infinite ordinal to a positive integer
power}
   k = 1       :  a
   limitp(a)   :  exp2(a,k)
   true        :  padd(exp2(c,k),
                       exp3h(c,natpart(a),n,k-1),
                       n)
                  where c = limitpart(a) and n = |a|
```

Figure 13: Ordinal exponentiation: raising an infinite ordinal to a positive integer power.

The third helper function, $\mathbf{exp_3}$, is defined in Figure 13. It is used to raise an infinite ordinal to a positive integer power. The complexity analysis of $\mathbf{exp_3}$ will reveal that the running time depends on the positive integer power, *i.e.*, it is exponential in the number of bits needed to represent the integer. As a result, the complexity of our algorithm for exponentiation is exponential. We will show at the end of this section that we cannot do much better.

Lemma 5.6.5. *For all* $\mathtt{a}$ *such that* $\mathbf{op}(\mathtt{a})$ *and* $\alpha \geq \omega$, $\mathbf{exp_3h}(\mathtt{c}, p, |\mathtt{a}|, k) = (\sum_{i=0}^{k-1} \mathbf{exp_2}(\mathtt{c}, k-i) \cdot_o p) +_o p$ *where* $\mathtt{c} = \mathbf{limitpart}(\mathtt{a})$ *and* $p = \mathbf{natpart}(\mathtt{a})$ *(the summation is with respect to* $+_o$*).*

Proof. Let $\mathtt{c} = \mathbf{limitpart}(\mathtt{a}) = [\mathtt{a}_1, x_1, [\mathtt{a}_2, x_2, \ldots [\mathtt{a}_n, x_n, 0] \ldots]]$. The proof is by induction on k. Clearly, by the definition of $\cdot_o$, the lemma holds when $k = 1$. For the induction step we have:

$$
\begin{aligned}
&\mathbf{exp_3h}(\mathtt{c}, p, n, k) \\
&= \{\, \textit{Definition of } \mathbf{exp_3h} \,\} \\
&\mathbf{padd}(\mathbf{exp_2}(\mathtt{c}, k) \cdot_o p, \mathbf{exp_3h}(\mathtt{a}, p, n, k-1), n) \\
&= \{\, \textit{Induction hypothesis, arithmetic} \,\} \\
&\mathbf{padd}(\mathbf{exp_2}(\mathtt{c}, k) \cdot_o p, (\textstyle\sum_{i=1}^{k-1} \mathbf{exp_2}(\mathtt{c}, k-i) \cdot_o p) +_o p, n) \\
&= \{\, \textit{See immediately below} \,\} \\
&(\textstyle\sum_{i=0}^{k-1} \mathbf{exp_2}(\mathtt{c}, k-i) \cdot_o p) +_o p
\end{aligned}
$$

We justify the last step of the above proof by noting that:

$$\mathbf{exp_2}(\mathtt{c}, k) \cdot_o p = [\mathtt{a}_1 \cdot_o x +_o \mathtt{a}_1,\ x_1 \cdot p,\ [\mathtt{a}_1 \cdot_o x +_o \mathtt{a}_2,\ x_2,\ \ldots [\mathtt{a}_1 \cdot_o x +_o \mathtt{a}_n,\ x_n,\ 0] \ldots]]$$

where $x = k - 1$. That is, by the definition of $\mathbf{exp_3h}$, we have that $\mathbf{fe}(\mathbf{exp_3h}(\mathtt{a}, p, n, k-1))$ $= (\mathtt{a}_1 \cdot_o (k-1))$; thus, every exponent in $\mathbf{exp_2}(\mathtt{c}, k) \cdot_o p$ is greater than the first exponent of $\mathbf{exp_3h}(\mathtt{a}, p, n, k-1)$. □

Theorem 5.6.6. *For all $\alpha \in \varepsilon_0$, $k \in \omega$ such that $\alpha \geq \omega$ and $k > 0$,* $\mathrm{CNF}(\alpha^k) = \mathbf{exp_3}(\mathtt{a}, k)$.

Proof. Recall that $\alpha = \sum_{i=1}^{n} \omega^{\alpha_i} x_i + p$ and let $\delta = \sum_{i=1}^{n} \omega^{\alpha_i} x_i$. Note that $\mathrm{CNF}(\delta) =$ $\mathbf{limitpart}(\mathtt{a})$. The case where $k = 1$ or $p = 0$ follows from Thm. 5.6.4. Otherwise, with the aid of Lem. 5.6.5, we can show that $\mathbf{exp_3}(\mathtt{a}, k) = \mathrm{CNF}(\delta^k + (\sum_{j=1}^{k-1} \delta^{k-j})p + p)$ and what remains is to show that $\alpha^k = \delta^k + (\sum_{j=1}^{k-1} \delta^{k-j})p + p$ for all $k > 0$. We do so by induction on k; note that the base case has already been addressed. For the induction step we have the following, where $\gamma = \sum_{j=1}^{k-2} \delta^{k-1-j}$ and $\xi = \delta^{k-1} + \gamma p + p$.

$$\begin{aligned}
& & \alpha^k \\
\{\ \textit{Exponentiation}\ \} & = & \alpha^{k-1} \cdot \alpha \\
\{\ \textit{Induction hypothesis, def. of } \delta\ \} & = & \xi(\delta + p) \\
\{\ \textit{Distributivity}\ \} & = & \xi\delta + \xi p \\
\{\ \textit{Lem. 5.6.4, def. } \xi\ \} & = & \delta^{k-1}\delta + (\delta^{k-1} + \gamma p + p)p \\
\{\ \delta^{k-1} > \gamma p\textit{, additive principal property}\ \} & = & \delta^k + \delta^{k-1}p + \gamma p + p \\
\{\ \textit{Distributivity, def } \gamma\ \} & = & \delta^k + (\textstyle\sum_{j=1}^{k-1} \delta^{k-j})p + p
\end{aligned}$$

□

We now analyze the complexity of $\mathbf{exp_3}$.

Lemma 5.6.6. $\mathbf{exp_3h}(\mathtt{a}, p, n, k)$ *runs in time* $O(k(|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a}))$ *when* $\neg\mathbf{finp}(\mathtt{a})$, $\mathbf{limitp}(\mathtt{a})$, $p \in \omega, n = |\mathtt{a}|$, *and* $0 < k < \omega$.

Proof. Note that for any $\mathtt{c}$, the following hold: $\mathtt{c} \cdot_o p$ takes $O(1)$ time (by the definition of **pmult**); $\mathtt{c} +_o p$ takes $O(|\mathtt{c}|)$ time (by Thm. 5.3.2); $|\mathbf{exp_2}(\mathtt{c}, k) \cdot_o p| = |\mathtt{c}|$ (see Thm. 5.6.5); and $\#(\mathbf{exp_2}(\mathtt{c}, k) \cdot_o p) = \#\mathtt{c}$ (again, see Thm. 5.6.5). Now, $\mathbf{exp_3h}$ gets called $O(k)$ times and each call requires $O(|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a} + |\mathtt{a}|)$ time. Therefore, by Thm. 5.6.5 and the above observations, the total time is $O(k(|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a}))$. □

```
exp4(a,b)  {raising an infinite ordinal to an infinite power}
   true   :  [fe(a) ·o limitpart(b), 1, 0] ·o exp3(a,natpart(b))
expo(a,b)  {ordinal exponentiation (raises a to the b power)}
   b = 0   ∨  a = 1     :  1
   a = 0                :  0
   finp(a)  ∧  finp(b)  :  expω(a,b)
   finp(a)              :  exp1(a,b)
   finp(b)              :  exp3(a,b)
   true                 :  exp4(a,b)
```

Figure 14: The ordinal exponentiation algorithm.

Theorem 5.6.7. $\mathbf{exp_3}(\mathtt{a}, k)$ *runs in time* $O(k \cdot (|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a}))$.

Proof. This is a straightforward consequence of Lem. 5.6.6 and Thm. 5.6.5. □

The fourth and final helper function, $\mathbf{exp_4}$, and $\mathbf{exp_o}$ are defined in Figure 14. We use $\mathbf{exp_4}$ to raise an infinite ordinal to an infinite power. The exponentiation function, $\mathbf{exp_o}$, is now simple to define, as all that is required is to invoke the appropriate helper function. We start by showing the correctness of $\mathbf{exp_4}$, after which the correctness of $\mathbf{exp_o}$ is immediate. We end the section by analyzing the complexity of $\mathbf{exp_4}$ and $\mathbf{exp_o}$, and we show that even though the complexity of $\mathbf{exp_o}$ is exponential, it is of the same order as the size of the resulting ordinal.

Theorem 5.6.8. *For all* $\alpha, \beta \in \varepsilon_0$*, such that* $\alpha, \beta \geq \omega$, $\mathrm{CNF}(\alpha^\beta) = \mathbf{exp_4}(\mathtt{a}, \mathtt{b})$.

Proof. We note the following sequence of equalities.

$$
\begin{aligned}
& & \mathrm{CNF}(\alpha^\beta) \\
\{\, \textit{Def. of } \beta \,\} &= & \mathrm{CNF}(\alpha^{\sum_{i=1}^{m} \omega^{\beta_i} y_i + q}) \\
\{\, \textit{Ordinal arithmetic} \,\} &= & \mathrm{CNF}(\textstyle\prod_{i=1}^{m} \alpha^{\omega^{\beta_i} y_i} \cdot \alpha^q) \\
\{\, \textit{Lem. 5.6.2} \,\} &= & \mathrm{CNF}(\textstyle\prod_{i=1}^{m} \omega^{\alpha_1 \cdot \omega^{\beta_i} y_i} \cdot \alpha^q) \\
\{\, \textit{Property of exponentiation} \,\} &= & \mathrm{CNF}(\omega^{\alpha_1 \cdot \sum_{i=1}^{m} \omega^{\beta_i} y_i} \cdot \alpha^q) \\
\{\, \textit{Thm. 5.6.6, Cor. 5.5.1} \,\} &= & [\mathtt{a}_1 \cdot_o \mathbf{limitpart}(\mathtt{b}), 1, 0] \cdot_o \mathbf{exp_2}(\mathtt{a}, q) \\
\{\, \textit{Definition } \mathbf{exp_4} \,\} &= & \mathbf{exp_4}(\mathtt{a}, \mathtt{b})
\end{aligned}
$$

□

Theorem 5.6.9. *For all* $\alpha, \beta \in \varepsilon_0, \mathrm{CNF}(\alpha^\beta) = \mathbf{exp_o}(\mathtt{a}, \mathtt{b})$.

Proof. The proof follows from Theorems 5.6.2 on page 39, 5.6.6 on page 43, and 5.6.8 on the preceding page. □

Lemma 5.6.7. $\#(\mathtt{a} +_o \mathtt{b}) \leq \#\mathtt{a} + \#\mathtt{b}$

Proof. The proof is by induction on $\#\mathtt{a}$. □

Lemma 5.6.8. $\mathbf{limitp}(\mathtt{b}) \quad \Rightarrow \quad |\mathtt{a} \cdot_o \mathtt{b}| = |\mathtt{b}|$

Proof. The proof is by induction on $|\mathtt{b}|$. □

Lemma 5.6.9. $\mathbf{limitp}(\mathtt{b}) \quad \Rightarrow \quad \#(\mathtt{a} \cdot_o \mathtt{b}) \leq \#\mathtt{a}_1|\mathtt{b}| + \#\mathtt{b}$

Proof. The proof is by induction on the size of $\mathtt{b}$. □

Theorem 5.6.10. $\mathbf{exp_4}(\mathtt{a}, \mathtt{b})$ *runs in time* $O(\mathbf{natpart}(\mathtt{b})[|\mathtt{a}||\mathtt{b}|+|\mathtt{a}_1||\mathtt{a}|+\#\mathtt{a}]+\#\mathbf{fe}(\mathtt{a}_1)|\mathtt{b}|+\#\mathtt{b})$.

Proof. There are 3 operations that $\mathbf{exp_4}$ calls that take more than constant time. The first is $\mathbf{exp_3}(\mathtt{a}, \mathbf{natpart}(\mathtt{b}))$, which we showed runs in time $O(\mathbf{natpart}(\mathtt{b}) \cdot (|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a}))$. The second is $\mathtt{a}_1 \cdot_o \mathbf{limitpart}(\mathtt{b})$, which takes time $O(|\mathbf{fe}(\mathtt{a}_1)||\mathtt{b}| + \#\mathbf{fe}(\mathtt{a}_1) + \#\mathtt{b})$. The final operation is $[\mathtt{a}_1 \cdot_o \mathbf{limitpart}(\mathtt{b}), 1, 0] \cdot_o \mathbf{exp_3}(\mathtt{a}, \mathbf{natpart}(\mathtt{b}))$. If we let $\mathtt{c} = [\mathtt{a}_1 \cdot_o \mathbf{limitpart}(\mathtt{b}), 1, 0]$ and $\mathtt{d} = \mathbf{exp_3}(\mathtt{a}, \mathbf{natpart}(\mathtt{b}))$, we obtain a time bound of $O(|\mathbf{fe}(\mathtt{c})||\mathtt{d}| + \#\mathbf{fe}(\mathtt{c}) + \#\mathtt{d})$. By Lemmas 5.6.8 and 5.6.9, among others, we have that $|\mathbf{fe}(\mathtt{c})| = |\mathtt{b}|$, $\#\mathbf{fe}(\mathtt{c}) = \#\mathbf{fe}(\mathtt{a}_1)|\mathtt{b}| + \#\mathtt{b}$, $|\mathtt{d}| = |\mathtt{a}| \cdot \mathbf{natpart}(\mathtt{b})$, and $\#\mathtt{d} = \#\mathtt{a} \cdot \mathbf{natpart}(\mathtt{b})$.

Hence, the complexity of this operation is $O(|\mathtt{b}|(|\mathtt{a}|\mathbf{natpart}(\mathtt{b})) + \#\mathbf{fe}(\mathtt{a}_1)|\mathtt{b}| + \#\mathtt{b} + \#\mathtt{a} \cdot \mathbf{natpart}(\mathtt{b}))$, which gives an overall complexity for the algorithm of

$$\begin{aligned} O(\ & \mathbf{natpart}(\mathtt{b}) \cdot (|\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a}) \\ & + |\mathbf{fe}(\mathtt{a}_1)||\mathtt{b}| + \#\mathbf{fe}(\mathtt{a}_1) + \#\mathtt{b} + |\mathtt{b}|(|\mathtt{a}|\mathbf{natpart}(\mathtt{b})) \\ & + \#(\mathbf{fe}(\mathtt{a}_1))|\mathtt{b}| + \#\mathtt{b} + \#\mathtt{a} \cdot \mathbf{natpart}(\mathtt{b})) \end{aligned}$$

By gathering like terms and noting that $\#(\mathbf{fe}(\mathtt{a}_1))|\mathtt{b}| > |\mathbf{fe}(\mathtt{a}_1)||\mathtt{b}|$, we obtain a time bound of $O(\mathbf{natpart}(\mathtt{b})[|\mathtt{a}||\mathtt{b}| + |\mathtt{a}_1||\mathtt{a}| + \#\mathtt{a}] + \#\mathbf{fe}(\mathtt{a}_1)|\mathtt{b}| + \#\mathtt{b})$. □

Theorem 5.6.11. $\mathbf{exp_o}(\mathtt{a}, \mathtt{b})$ *runs in time* $O(\mathbf{natpart}(\mathtt{b})[|\mathtt{a}||\mathtt{b}|+|\mathtt{a}_1||\mathtt{a}|+\#\mathtt{a}]+\#\mathbf{fe}(\mathtt{a}_1)|\mathtt{b}|+\#\mathtt{b})$.

Proof. This follows directly from Theorems 5.6.3 on page 40, 5.6.7 on page 44, and 5.6.10 on the previous page. □

An obvious question is whether we can improve the exponential running time of $\mathbf{exp_o}$. Given our representation of the ordinals, the answer is no, as the following class of examples shows. Fix a to be [1, 1, 1], which corresponds to the ordinal $\omega + 1$ and let $\mathtt{b}_k$ be [[1, 1, 1], 1, k], which corresponds to the ordinal $\omega^\omega + k$. For this infinite class of examples, $\#\mathbf{exp_o}(\mathtt{a}, \mathtt{b}_k)$ is exactly equal to the complexity of $\mathbf{exp_o}$. That is, simply constructing the ordinal corresponding to $\mathbf{exp_o}(\mathtt{a}, \mathtt{b}_k)$ takes as long as this function takes to run in the worst case. Therefore, this algorithm is as efficient as can be expected.

5.7 Bibliographic Notes

The ordinal numbers were introduced by Cantor over 100 years ago and are at the core of modern set theory [21, 22, 23]. They are an extension of the natural numbers into the transfinite and are an important tool in logic, *e.g.*, after Gentzen's proof of the consistency of Peano arithmetic using the ordinal number ε_0 [42], proof theorists routinely use ordinals and ordinal notations to establish the consistency of logical theories [108, 112]. To obtain constructive proofs, constructive ordinals notations are employed. The general theory of ordinal notations was initiated by Church and Kleene [24] and is recounted in Chapter 11 of Roger's book on computability [96].

An early use of the ordinals for proving program termination is due to Alan M. Turing, who in 1949 wrote the following [115, 85].

> The checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops. To the pure mathematician it is natural to give an ordinal number. In this problem the ordinal might be

Table 5: Ordinal Arithmetic Complexity Results

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

$(n - r)\omega^2 + (r - s)\omega + k$.[1]

Partial solutions to the ordinal arithmetic problem appear in various books and papers [108, 35, 40, 80, 109, 112], *e.g.*, it is easy to find a definition of $<$ for various ordinal notations, but we have not found any statement of the problem nor any comprehensive solution in previous work. One notable exception is the dissertation work of John Doner [39, 38]. Doner and Tarski (his adviser) study hierarchies of ordinal arithmetic operations. They give a transfinite recursive definition for binary operations O_γ for any ordinal γ. The operation O_0 corresponds to addition, O_1 corresponds to multiplication, O_2 corresponds to exponentiation (for the most part), and so on. Using this hierarchy of operations, Doner and Tarski define a generalization of the Cantor normal form. However, Doner and Tarski stop short of defining an ordinal notation. They give pseudo-algorithms for O_1, O_2, and O_3, but it is not immediately clear how to apply these to an ordinal notation to obtain algorithms. This was not within the scope of their work, as they were studying operations on the set-theoretic ordinals, not on ordinal notations.

5.8 Summary

In this chapter, we presented algorithms for comparing (ordering) ordinals, recognizing ordinals in our notation, as well as performing ordinal addition, subtraction, multiplication, and exponentiation. For multiplication and exponentiation, we presented two separate

[1] Readers familiar with the ordinals may suspect that Turing's measure function is not quite right, as $(i)\omega^j = \omega^j$ when i and j are positive integers. This seems to be purely a notational issue, *e.g.*, Turing uses the same convention in a paper on logics based on ordinals [114]. Using modern conventions, the measure function is written $\omega^2(n - r) + \omega(r - s) + k$.

algorithms each, one which is relatively simple but inefficient, and one that is efficient but more complicated. We proved the correctness of all the algorithms and analyzed their complexity. The complexity results are summarized in Table 5 on the preceding page.

CHAPTER VI

ORDINAL ARITHMETIC: MECHANIZATION

In this chapter, we utilize the results of Chapter 5 to create a framework for mechanically reasoning about the ordinals algebraically. We do this by implementing our ordinal notation and algorithms in the ACL2 theorem proving system, mechanically verifying them by proving that the algorithms share well-known properties with the set-theoretic functions they implement, and engineering a powerful library of theorems in ACL2 for automatically reasoning about the ordinals. We describe these steps in Sections 6.1, 6.2, and 6.3, respectively.

In addition, we have permanently altered ACL2's logic, replacing its original framework for reasoning about the ordinals with our own, as described in Section 6.4. The result is that ACL2 has become a more powerful tool for reasoning about the ordinals, and therefore about the termination of functions. This is demonstrated in Section 6.5, in which we empirically evaluate the effectiveness of our framework for reasoning about the ordinals by examining two case studies from the ACL2 regression suite.

Finally, we reflect on lessons we learned developing and integrating our new framework into ACL2 in Section 6.6, which we hope will be useful to others who may be interested in integrating new reasoning frameworks into ACL2.

6.1 Definitions

We begin by defining our ordinal notation in Figure 15 on the next page. The `finp` function corresponds to `finp`, and recognizes when an ordinal is finite. In our ACL2 notation, ordinals are infinite if and only if they are lists. Therefore, `finp` simply returns whether its argument is an atom. The `infp` macro simply expands to the negation of `finp`, and therefore recognizes if an ordinal is infinite. Infinite ordinals are constructed using the `make-ord` function. The ordinal it creates can be thought of as a linked list whose nodes contain exponent-coefficient pairs, and whose last element is a natural number (the **natpart**of the

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

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

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

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

(defun o-first-coeff (x)
  (cond ((o-finp x) x)
        (t (cdar x))))

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

Figure 15: Ordinal Constructors and Destructors

ordinal). So, $\omega^3 2 + 1$ is represented as `((3 . 2) . 1)`, $\omega^2 3 + \omega 4 + 5$ is `((2 . 3) (1 . 4) . 5)`, and $\omega^{\omega^2 3} + \omega^4 5 + 6$ is `((((2 . 3) . 0) . 1) (4 . 5) . 6)`. The remaining three functions are the ordinal destructors; `o-first-expt`, `o-first-coeff`, and `o-rst` correspond to **fe**, **fco**, and **rst**, respectively.

Using these functions to perform basic manipulations of infinite ordinals, we define the ordinal arithmetic functions in ACL2. Here we present the definition of the ordinal multiplication function, which appears in Figure 16 on the following page. Not all the functions used in this figure are defined there. These include `dropn`, which, given a natural number, n, returns the last $m - n$ elements of a list of length m. Also not defined here is `o+`, which is a macro that applies the binary ordinal addition function to any number of arguments.

For the most part, these definitions should look familiar from Chapter 5. However, there are some ACL2-specific features of interest being used in these examples. The first is the declaration of guards. A guard does not affect the logical definition of the function, but tells ACL2 the intended domain of the function, which allows ACL2 to use smarter compilation methods to ensure faster execution times. Throughout the ordinal function definitions, we use guards for this purpose.

Another way we use guards is in conjunction with ACL2's `defexec` feature, which allows users to attach an executable to a logical definition. When using `defexec`, the user is obligated to prove that, when the guard conditions on the function parameters are satisfied, ACL2 will use the executable definition to execute the functions, rather than the logical

```
(defun count1 (x y)
  (declare (xargs :guard (and (o-p x) (o-p y))))
  (cond ((o-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)
  (declare (xargs :guard (and (o-p x) (o-p y) (natp n))))
  (+ n (count1 (dropn n x) y)))

(defun padd (x y n)
  (declare (xargs :guard (and (o-p x) (o-p y) (natp n) (<= n (count1 x y)))))
  (if (or (o-finp x) (zp n))
      (o+ x y)
    (make-ord (o-first-expt x) (o-first-coeff x) (padd (o-rst x) y (1- n)))))

(defun pmult (x y n)
  (declare (xargs :guard (and (o-p x) (o-p y) (natp n)
                              (<= n (count1 (o-first-expt x)
                                            (o-first-expt y))))))
  (let* ((fe-x (o-first-expt x))
         (fe-y (o-first-expt y))
         (fco-x (o-first-coeff x))
         (fco-y (o-first-coeff y))
         (m (count2 fe-x fe-y n)))
    (cond ((or (equal x 0) (equal y 0)) 0)
          ((and (o-finp x) (o-finp y)) (* x y))
          ((o-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))))))

(defexec ob* (x y)
  (declare (xargs :guard (and (o-p x) (o-p y))))
  (mbe :logic (let ((fe-x (o-first-expt x)) (fe-y (o-first-expt y))
                    (fco-x (o-first-coeff x)) (fco-y (o-first-coeff y)))
                (cond ((or (equal x 0) (equal y 0)) 0)
                      ((and (o-finp x) (o-finp y)) (* x y))
                      ((o-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)))

(defmacro o* (&rest rst)
  (cond ((null rst) 1)
        ((null (cdr rst))
         (car rst))
        (t (xxxjoin 'ob* rst))))
```

Figure 16: ACL2 definitions of ordinal multiplication.

definition [45]. This allows the user to have different definitions in the logical and executable worlds that are guaranteed to be equivalent. Note that we use this feature when defining the binary ordinal multiplication function, `ob*`. Here, we attach the efficient algorithm for multiplication, corresponding to $\cdot_o$ to the simpler, but less efficient logical definition, corresponding to $*_o$. The result is a multiplication function that is both efficient and relatively easy to reason about. We similarly define `o^`, using its simpler and more efficient definitions.

Another feature of note is the macro, `o*`, defined at the end of the figure. This macro calls the `xxxjoin` function, which, when given a binary function symbol and a list, returns the expression applying the function to the list of arguments in a right-associated manner. So, for example, `(o* a b c d)`, becomes `(ob* a (ob* b (ob* c d)))`. This allows us to create polyadic versions of our binary functions. To improve readability, ACL2 can be instructed to print `ob*` in terms of `o*` with the command `(add-binop o* ob*)`. Thus, users are under the illusion that they are reasoning about polyadic functions, while all reasoning is really with respect to the binary functions. This helps to simplify the interface between theorem prover and user.

6.2 Mechanical Verification

The mechanical verification of the ordinal arithmetic algorithm implementations involved proving three different classes of theorems, beyond the guard conjectures and termination proofs mentioned in Section 6.1. The first class deals with the algebraic properties of the operations. We proved that each function has the same algebraic properties as its corresponding set-theoretic operation. Almost all of the following well-known properties appear as theorems in our library on ordinal representations. The four properties marked with a "†" do not appear in the library, but the proofs are simple consequences the theorems we do prove.

The ordering relation on ordinals satisfies the following properties.

$$\neg(\alpha < \alpha) \qquad \text{(irreflexivity)}$$

$$\beta < \alpha \Rightarrow \neg(\alpha < \beta) \quad \wedge \quad \neg(\alpha = \beta)$$

$$\alpha < \beta \quad \wedge \quad \beta < \gamma \Rightarrow \alpha < \gamma \qquad \text{(transitivity)}$$

$$\neg(\alpha < \beta) \quad \wedge \quad \neg(\alpha = \beta) \Rightarrow \beta < \alpha \qquad \text{(totality)}$$

Ordinal addition and subtraction satisfy the following properties.

$$\alpha + 0 = \alpha$$

$$0 + \alpha = \alpha$$

$$\alpha < \alpha + 1$$

† $\quad \alpha + 1 \;=\; \alpha'$

$$\alpha < \beta \quad \equiv \quad \alpha + 1 \leq \beta$$

$$\alpha < \beta + 1 \quad \equiv \quad \alpha \leq \beta$$

$$\alpha \leq \beta + \alpha$$

$$\alpha \leq \alpha + \beta$$

$$(\alpha + \beta) + \gamma \;=\; \alpha + (\beta + \gamma) \qquad \text{(associativity)}$$

$$(\beta < \gamma) \;\Rightarrow\; \alpha + \beta < \alpha + \gamma \qquad \text{(strict right monotonicity)}$$

$$(\beta \leq \gamma) \;\Rightarrow\; \beta + \alpha \leq \gamma + \alpha \qquad \text{(weak left monotonicity)}$$

$$(\alpha < \omega^\beta) \;\Rightarrow\; \alpha + \omega^\beta = \omega^\beta \qquad \text{(additive principal property)}$$

† $\quad (\alpha, \beta < \omega^\gamma) \;\Rightarrow\; \alpha + \beta < \omega^\gamma \qquad \text{(closure of additive principal ordinals)}$

$$\alpha - \alpha = 0$$

$$\alpha - \beta \leq \alpha$$

$$\alpha \leq \beta \Rightarrow \alpha + (\beta - \alpha) = \beta$$

$$\alpha + \gamma = \beta \Rightarrow \beta - \alpha = \gamma$$

$$\alpha + \beta = \alpha + \gamma \quad \equiv \quad \beta = \gamma$$

Ordinal multiplication satisfies the following properties.

$$\alpha 0 = 0$$

$$0\alpha = 0$$

$$\alpha 1 = \alpha$$

$$1\alpha = \alpha$$

$$\dagger \quad n \in \omega \wedge n > 0 \;\Rightarrow\; n \cdot \omega = \omega$$

$$(\alpha \cdot \beta) \cdot \gamma \;=\; \alpha \cdot (\beta \cdot \gamma) \qquad \text{(associativity)}$$

$$(\beta < \gamma) \;\Rightarrow\; \alpha \cdot \beta < \alpha \cdot \gamma \qquad \text{(strict right monotonicity)}$$

$$(\beta \leq \gamma) \;\Rightarrow\; \alpha \cdot \beta \leq \alpha \cdot \gamma \qquad \text{(weak right monotonicity)}$$

$$(\beta \leq \gamma) \;\Rightarrow\; \beta \cdot \alpha \leq \gamma \cdot \alpha \qquad \text{(weak left monotonicity)}$$

$$\alpha \cdot (\beta + \gamma) \;=\; (\alpha \cdot \beta) + (\alpha \cdot \gamma) \qquad \text{(left distributivity)}$$

Ordinal exponentiation has the following properties.

$$\alpha^0 = 1$$

$$\alpha^1 = \alpha$$

$$0 < \alpha \Rightarrow 0^\alpha = 0$$

$$1^\alpha = 1$$

$$\alpha^\beta \cdot \alpha^\gamma \;=\; \alpha^{\beta+\gamma}$$

$$(\alpha^\beta)^\gamma \;=\; \alpha^{\beta \cdot \gamma}$$

$$(\beta < \gamma) \;\Rightarrow\; \alpha^\beta < \alpha^\gamma \qquad \text{(strict right monotonicity)}$$

$$(\beta \leq \gamma) \;\Rightarrow\; \beta^\alpha \leq \gamma^\alpha \qquad \text{(weak left monotonicity)}$$

$$\dagger \quad (p \in \omega) \;\Rightarrow\; p^\omega = \omega$$

Limit ordinals satisfy the following properties

$$lim.\beta \Rightarrow \alpha < \beta \equiv \alpha + 1 < \beta$$

$$lim.\beta \quad \wedge \quad \alpha < \omega^\beta \Rightarrow \langle \exists \gamma :: \alpha < \gamma \quad \wedge \quad \gamma < \omega^\beta \rangle$$

In addition to these theorems, we created counterexamples to the following conjectures

in ACL2.

$$\alpha + \beta = \beta + \alpha \qquad \text{fails when } \alpha = 1,\ \beta = \omega$$

$$\beta < \gamma \Rightarrow \beta + \alpha < \gamma + \alpha \qquad \text{fails when } \alpha = \omega,\ \beta = 1, \text{ and } \gamma = 2$$

$$(\alpha + \beta) - \gamma = \alpha + (\beta - \gamma) \qquad \text{fails when } \alpha = \omega + 1,\ \beta = 1, \text{ and } \gamma = 2$$

$$\alpha\beta = \beta\alpha \qquad \text{fails when } \alpha = 2,\ \beta = \omega$$

$$(\beta + \gamma)\alpha = \beta\alpha + \gamma\alpha \qquad \text{fails when } \alpha = \omega,\ \beta = 1, \text{ and } \gamma = 1$$

$$\beta < \gamma \Rightarrow \beta\alpha < \gamma\alpha \qquad \text{fails when } \alpha = \omega,\ \beta = 1, \text{ and } \gamma = 2$$

$$(\alpha\beta)^\gamma = \alpha^\gamma\beta^\gamma \qquad \text{fails when } \alpha = 2,\ \beta = 2, \text{ and } \gamma = \omega$$

$$(\beta < \gamma) \Rightarrow \beta^\alpha < \gamma^\alpha \qquad \text{fails when } \alpha = \omega,\ \beta = 2, \text{ and } \gamma = 3$$

The second class of theorems are about the notation and involve helper functions, such as `make-ord`, `o-first-expt`, `o-first-coeff`, and `o-rst`, and how they interact with the algebraic functions. An example of this is the following theorem.

```
(defthm o+-fe-1
  (implies (o< (o-first-expt a)
               (o-first-expt b))
           (equal (o-first-expt (o+ a b))
                  (o-first-expt b))))
```

The third class of theorems demonstrates an isomorphism between the ordinals in our notation and those in ACL2's original notation. More precisely, let O_c be the set of objects that correspond to ordinals in our representation, and let O_a be the set of objects representing the original ACL2 ordinals. The purpose of this class of theorems is to show that O_c is isomorphic to O_a. To this end, we created two functions: `ctoa`, which maps O_c to O_a, and `atoc`, which O_a to O_c. To show that our representation is isomorphic to the ordinals in ACL2, we show the following.

First, `ctoa` is well defined. That is, `x` $\in O_c \Rightarrow$ `(ctoa x)` $\in O_a$. Translating this to ACL2 gives us:

```
(implies (o-p x)
         (e0-ordinalp (ctoa x)))
```

We also proved the equivalent theorem for `atoc`:

```
(implies (e0-ordinalp x)
         (o-p (ctoa x)))
```

Second, we show that `ctoa` is surjective. By definition, this means $\langle \forall \mathtt{x} \in O_a :: \langle \exists \mathtt{y} \in O_c :: \mathtt{(ctoa\ y)} = \mathtt{x} \rangle \rangle$. We prove this by showing that `atoc` is the inverse of `ctoa`. Thus, given $\mathtt{x} \in O_a$, we have that `(ctoa (atoc x))` = `x`. In ACL2, this becomes:

```
(implies (e0-ordinalp x)
         (equal (ctoa (atoc x))
                x))
```

We also proved the equivalent theorem for `atoc`:

```
(implies (o-p x)
         (equal (atoc (ctoa x))
                x))
```

Third, we show that `ctoa` is injective, *i.e.*, $\langle \forall \mathtt{x}, \mathtt{y} \in O_c :: \mathtt{(ctoa\ x)} = \mathtt{(ctoa\ y)} \Rightarrow \mathtt{x} = \mathtt{y} \rangle$. In ACL2:

```
(implies (and (o-p x)
              (o-p y))
         (equal (equal (ctoa x) (ctoa y))
                (equal x y)))
```

We also proved the equivalent theorem for `atoc`:

```
(implies (and (e0-ordinalp x)
              (e0-ordinalp y))
         (equal (equal (atoc x) (atoc y))
                (equal x y)))
```

Finally, we show that `ctoa` is homomorphic with respect to `o<` and `e0-ord-<`. That is, $\mathtt{x}, \mathtt{y} \in O_c$ such that `(o< x y)` $\Rightarrow$ `(e0-ord-< (ctoa x) (ctoa y)`. In ACL2, this is:

```
(implies (and (o-p x)
              (o-p y))
         (equal (e0-ord-< (ctoa x)
                          (ctoa y))
                (o< x y)))
```

and equivalently for `atoc`

```
(implies (and (e0-ordinalp x)
              (e0-ordinalp y))
         (equal (o< (atoc x)
                    (atoc y))
                (e0-ord-< x y)))
```

Note that these theorems deal with ordinal notations and the implementations in ACL2 of the ordinal arithmetic algorithms. That is, we do not mechanically establish any connection with the set-theoretic definitions on which our algorithms are based, as our goal was not to formalize set-theory in ACL2. Instead, we focused on using our results about arithmetic on ordinal notations to extend ACL2's ability to reason about termination. However, many of the "paper and pencil" proofs from Chapter 5 turned out to be quite useful, as they provided the key insights required to complete the ACL2 proofs.

6.3 Library Design

Enabling ACL2 to effectively and automatically reason about the ordinals and termination requires more than proving the correctness of the implementations, the topic of the previous section. It requires carefully constructing a library that makes effective and efficient use of the various types of mechanisms that ACL2 provides to control the way in which theorems are used. In this section, we discuss a few important considerations that went into engineering a useful library.

6.3.1 Rule Classes

The first consideration involves a concept in ACL2 known as "rule classes." When ACL2 proves a theorem, it gets entered into a database so that it can be used in subsequent proof attempts. By default, theorems are entered as rewrite rules. Rewrite rules can be conditional and are triggered when a goal contains an expression matching the left hand size of the rule's consequent. When this happens, ACL2 attempts to establish the antecedents of the rule via backchaining, and if successful, it rewrites the expression, using the right hand side of the rewrite rule. For example, consider the following rule.

```
(defthm |~(a=0) /\ b>1  <=>  a < ab|
  (implies (and (o-p a)
                (o-p b))
           (equal (o< a (o* a b))
                  (and (not (equal a 0))
                       (not (equal b 0))
                       (not (equal b 1))))))
```

After proving this theorem, ACL2 enters it into the database of rules as a rewrite rule. Subsequently, when ACL2 sees an expression of the form `(o< e1 (o* e1 e2))`, where `e1` and `e2` are arbitrary ACL2 expressions, it will try to determine if `e1` and `e2` are `o-p`s. If so, ACL2 will rewrite `(o< e1 (o* e1 e2))` to `(and (not (equal e1 0)) (not (equal e2 0)) (not (equal e2 1)))`. Notice that although `(o< e1 (o* e1 e2))` is smaller in size than `(and (not (equal e1 0)) (not (equal e2 0)) (not (equal e2 1)))`, it contains `o<` and `o*`, which are relatively complex functions. It is important to orient rewrite rules so that they reduce expressions containing complex functions into expressions containing simpler functions. It is also important to take into account how much effort will be expended trying to discharge the hypotheses, and rules should be written in a way that forces expressions into "canonical" forms.

While rewrite rules are the most widely used rule class, there are other types of rules, *e.g.*, forward chaining rules are triggered when all of the antecedents are known to be true. When this happens, the consequent is added to the "context," the collection of known facts. As one can imagine, a large collection of theorems with varying rule classes such as those in the ordinal library can interact in 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 `(o-p 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 context 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 —*i.e.*, using proof analysis tools provided by ACL2 to find sources of inefficiency in proof attempts— 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.

6.3.2 Choosing Theorems to Export

It is also important to distinguish between the theorems that one wants to export versus the intermediate lemmas that are used to prove such theorems. For example, to prove the left distributive property of multiplication over addition, we had to prove several lemmas which correspond to special cases of the theorem. The distributive property theorem should be exported (made visible when the library is loaded into ACL2), but the supporting lemmas should not. This is accomplished with ACL2's `local` form. Sometimes a lemma can also cause problems within a library and in this case, one can use the `encapsulate` form, which provides a way of hiding `local` theorems from the rest of the library (and much more).

6.3.3 Functions versus Macros

Another concern is deciding when to use macros and when to use functions. In ACL2 macros are simply syntactic sugar and are expanded away before theorem proving begins. Thus, ACL2 does not reason about macros. In addition to using macros to define polyadic versions of our binary functions (see Section 6.1), we also use them simplify the class of theorems needed to reason about the ordinals. For example, we made `o<=` a macro such that `(o<= a b)` expands to `(not (o< b a))`. This greatly simplified our library, because we did not need to develop rewrite rules to reason about expressions involving `o<=`. The

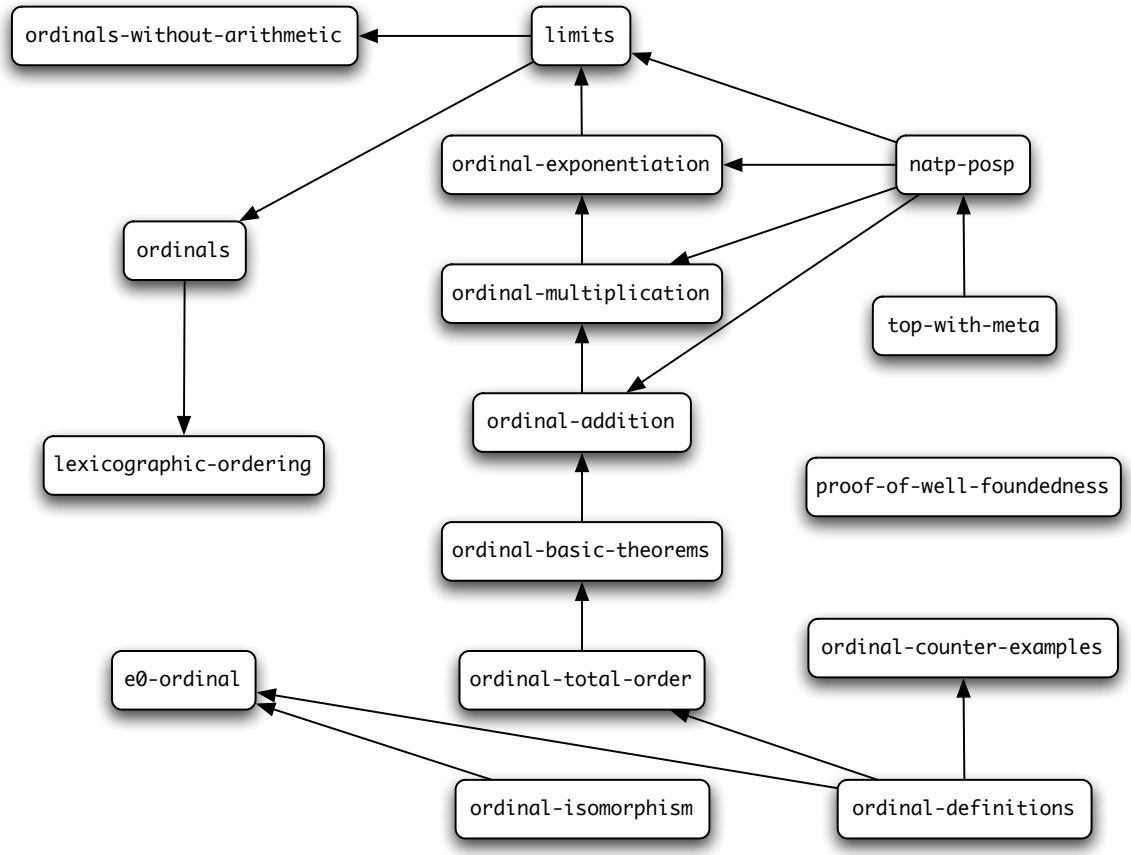


Figure 17: The Ordinal Library.

problem with this approach is that the output generated by ACL2 is with respect to `o<`, so we used the macro aliases table in ACL2 to force it to print `(o<= a b)` instead of `(not (o< b a))`. This leads to improved readability.

6.3.4 Library Structure

The final consideration we address here is the structure of the library. The library is divided into files of ACL2 theorems and definitions, called "books." Dividing the theorems properly between the books adds logical coherence and modularity to the library. This maximizes efficiency through code reuse and makes the books easier for users to understand. The structure of this library is illustrated in Figure 17, where the rectangles represent books, and the arrows represent the dependencies between books. For example, the arrow from `ordinal-isomorphism` to `e0-ordinal` indicates that the results of `e0-ordinal` rely on the

Table 6: The Ordinal Library.

Book	Description
`top-with-meta`	A link to the arithmetic books
`natp-posp`	Theorems about natp and posp
`ordinal-definitions`	The function definitions
`ordinal-total-order`	Theorems about the behavior of `o<`
`ordinal-basic-thms`	Basic theorems about the helper functions
`ordinal-addition`	Theorems about `o+` and `o-`
`ordinal-multiplication`	Theorems about `o*`
`ordinal-exponentiation`	Theorems about `o^`
`ordinal-isomorphism`	Proof of isomorphism of our ordinals & ACL2's
`e0-ordinal`	Exports major results of `ordinal-isomorphism`
`limits`	Theorems about limit ordinals
`ordinal-counter-examples`	Counter-examples, *e.g.*, commutativity
`ordinals-without-arithmetic`	Exports ordinal thms without integer arithmetic
`ordinals`	Exports ordinal thms with integer arithmetic
`proof-of-well-foundedness`	Part of proof of well-foundedness of our ordinals
`lexicographic-ordering`	Proves well-foundedness of a lexicographic order

results of `ordinal-isomorphism`. A short description of the contents of the books can be found in Table 6. The total size of the books is 181K and they consist of about 5,455 lines of definitions, theorems, and comments.

To use the ordinal library, the user loads either the `ordinals-without-arithmetic` or the `ordinals` book, depending on whether she wishes to include results about integer arithmetic. The integer arithmetic book, `top-with-meta`, was essential for the creation of the library, but can sometimes interfere with other books.

6.4 Integration with ACL2

After creating the ordinal arithmetic library, we decided to modify ACL2, replacing its ordinal representation by our own, so that it could take full advantage of our work. The modifications included updating the documentation and modifying the ACL2 sources and consisted of about 1,750 lines of code and documentation. We submitted the changes to Kaufmann and Moore, the authors of ACL2, and they have incorporated the changes into the ACL2 version 2.8 and all subsequent releases [57].

It is worth noting that our changes do not affect the soundness of the ACL2 logic. In

the `ordinal-isomorphism` book of our library, we exhibit a bijection between our ordinal representation and the previous ACL2 representation (see Corollary 4.3.1 on page 22). This proof was carried out in ACL2 version 2.7, thus guaranteeing that soundness is not affected.

We now discuss some of the issues we confronted in modifying ACL2. First, the ordinals are needed in ACL2's *ground-zero theory*, the initial theory encountered when starting an ACL2 session. Proving theorems, defining functions, including books, etc. all result in extensions to the ground-zero theory, and we wanted to keep it as clean and simple as possible. Therefore, we did not want to add our entire library of definitions and theorems to the ground-zero theory. Instead, we included only the basic constructors and destructors (`make-ord`, `o-first-expt`, `o-first-coeff`, `o-rst`), the functions necessary for `o-p` and `o<` (`natp`, `posp`, `infp`, `finp`, `o<`, `o-p`), and a few macros based on `o<` (`o>`, `o<=`, `o>=`). The arithmetic functions and theorems remain in the library.

Next, by examining ACL2's regression suite, we discovered that many books defined their own versions of `natp` or `posp`, and decided that ACL2 could benefit from additional reasoning about these functions. We therefore moved our `natp-posp` book from the ordinal library to ACL2's existing arithmetic library. This is 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.

After replacing the old ordinals with our new representation, we had to deal with legacy issues, including backward compatibility for the books included with ACL2, as many of these books referenced the old ordinal representation. The key to fixing these references was the theorems proved in the `ordinal-isomorphism` and `e0-ordinal` books. The main result in the books is a proof that there exists a bijection between the new and old ordinal representations. This result allowed us to switch the well-founded relation used by ACL2 to the version 2.7 relation (for the admission of the troublesome books only). This fixed most of the problems. However, some books used the old ordinals to prove more than just termination. Again, by using the bijection proof, we were able to transfer results about the old ordinals to the new ordinals, which resolved the remaining problems. We discuss this technique in more detail in Section 6.5.1.

6.5 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.

6.5.1 Legacy Books: Multiset Case Study

ACL2's multiset ordering library [98] 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 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 18 on the following page). 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

```
(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)
```

Figure 18: Original Multiset Results

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 transferred 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 re-tagged 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.

6.5.2 New Results: Dickson's Lemma Case Study

Our library was used by Sustik to give a constructive proof of Dickson's Lemma [110]. 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 [76]. 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, $m_0, m_1, m_2, \ldots$, 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 no such i and j exist, 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.

6.6 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 6.5.1, 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 [57]. 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 difficult to do in ACL2. There is a mechanism called

`accumulated-persistence` 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 ACL2users 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.

6.7 Bibliographic Notes

The automated reasoning community has studied the problem of formalizing the ordinals (as opposed to ordinal notations), focusing on proving known results. Dennis and Smaill

studied higher-order heuristic extensions of rippling. They used ordinal arithmetic as a case study, which was implemented in *λClam*, a higher order proof planning system for induction. They were able to successfully plan standard undergraduate textbook problems using their system [33]. Paulson and Grabczewski have mechanized a good deal of set theory, including the proof that for any infinite cardinal, κ, we have $\kappa \otimes \kappa = \kappa$, most of the first chapter of Kunen's excellent book on set theory [63], and the equivalence of eight forms of the well-ordering theorem [92]. More recently, Paulson has mechanized the proof of the relative consistency of the axiom of choice and has proved the reflection theorem [91, 90]. Paulson and Grabczewski's efforts required reasoning about the ordinals and were carried out with the Isabelle/ZF system [87, 89]. A version of the reflection theorem was also proved by Bancerek, using Mizar [4]. Another line of work is by Belinfante, who has used Otter to prove elementary theorems of ordinal number theory [5, 6, 7]. There is much more work that can be mentioned, but we end by listing some of the theorem proving systems for which there exists support for the ordinals: Nqthm [13], ACL2 [56], Coq [11], PVS [86], HOL [44], Isabelle [88], and Mizar [97].

For a complete discussion of the `defexec` feature and its applications (including this one), see [45], and for more details, see [46]. This and other features of ACL2 mentioned in this chapter can be found on the ACL2 homepage [57]. The details of the multiset ordering books described in Section 6.5.1 can be found in [98].

Dickson's lemma was introduced in [37], while Buchberger's algorithm for finding Gröbner bases first appeared in [20]. More detail on Sustik's constructive proof is given in [110].

6.8 Summary

In this chapter, we described the creation of a framework for mechanically reasoning about the ordinals algebraically. We discussed the implementation and mechanical verification of our ordinal notation and arithmetic algorithms (see Chapter 5) in the ACL2 theorem proving system. We also gave an overview of the issues involved in the creation of a well-engineered library for reasoning about ordinal arithmetic in ACL2, and the modification of ACL2 to use our ordinal representation by default. We explored two case studies that

demonstrate the effectiveness of our ordinal reasoning framework, and considered some lessons we learned in the process of designing and building this framework.

Finally, we presented lessons we 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.

PART III

Automation of Termination Proofs

CHAPTER VII

OVERVIEW

In this part of the dissertation, we present a new algorithm for automatically proving the termination of programs written in first-order purely-functional programming languages. This algorithm is general, as it can be used to reason about any and all looping behaviors allowed by programs written in this class of languages. It does so by using a novel combination of theorem proving and static analysis to abstract the program into a manageable but surprisingly accurate representation, which we call a calling context graph (CCG). By annotating the CCG with expressions we call calling context measures (CCMs), and using the theorem prover to reason about the values of these CCMs, we can show that the programs represented by the CCG are terminating on all inputs. We describe this algorithm in Chapter 8.

In addition, we describe an implementation of the CCG algorithm that is based on a hierarchical model, in which lightweight versions of the CCG algorithm are used to prove the termination of simpler programs, while the slower full CCG algorithm is employed only when the simpler methods fail. The details of this implementation are given in Chapter 9.

Chapter 10 presents the results of running our algorithm on the entire ACL2 regression suite. The results demonstrate that our analysis can automatically prove termination for over 96% of terminating functional definitions, including over 79.90% of the most difficult 1.5% of all function definitions, even when using only the ACL2's ground-zero theory and definitional axioms. Under the same circumstances, ACL2 can only prove termination of 59.93% of the difficult problems, demonstrating the significant improvement of our analysis over ACL2's.

Finally, we discuss the issues involved in integrating our CCG-based termination analysis into the ACL2 logic. We begin by proving that the formal notion of *CCG admissibility*,

which involves proving termination using CCGs in ACL2, implies ACL2's current termination condition, known as *measure admissibility*. This proves that the integration of the CCG algorithm into ACL2 would be a conservative extension of ACL2's current logic. We also discuss the practical challenges of integrating the CCG analysis with ACL2's theorem prover. This is done in Chapter 11.

CHAPTER VIII

TERMINATION ANALYSIS ALGORITHM

In this chapter, we present our automatic termination analysis algorithm. We begin by giving an example.

8.1 An Example

Consider the code in Figure 19. It is taken verbatim from the M5 model of the Java Virtual Machine, written in ACL2 [84, 77]. These two particular functions create a multi-dimensional array, and add it to the heap. It calls code to create a new JVM state, create new heaps, and bind addresses to values in the heap. It relies on over 800 lines of previous code in the JVM model. The two functions defined in the figure, called `mma2` and `mma` are mutually recursive. `mma2` calls itself and `mma`, and `mma` calls `mma2`. We walk through our CCG analysis to give a high level overview of how it works before giving the details.

First, notice that there is a significant amount of code that does not factor into the looping behavior. For example, in `mma`, there is 7 lines of code that comprise the "base case" of the function. That is, if the `if` test, `(<= (len c) 1)` is true, there are no recursive calls. Likewise, the last 7 lines of the function take the value returned by the recursive call to `mma2`, and construct the return values from it. It turns out that this is also not relevant to the termination argument. This is because all looping behavior in applicative first-order functional languages is modeled using recursion. So all we need to know is that the functions in question do not recur infinitely.

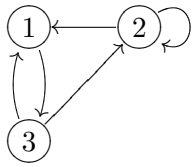
To determine this, we need three pieces of information from each recursive callsite. The first is the function in which the call occurs. The second is the governors under which the call is made (see Section 2.1). The third is the call itself. The triple containing this information is known as a *precise calling context*. The precise contexts for `mma2` and `mma` appear in Figure 20(a) on page 75. For the purposes of presentation, we abreviate the final two arguments to the second context as e and e'.

```
(mutual-recursion

  ; mma2 :: num, counts, s, ac --> [refs]
  (defun mma2 (c1 c2 s ac)
    (declare (xargs :measure (cons (len (cons c1 c2))
                                   (natural-sum (cons c1 c2)))))
    (if (zp c1)
        (mv (heap s) ac)
      (mv-let (new-addr new-heap)
              (mma c2 s)
              (mma2 (- c1 1)
                    c2
                    (make-state (thread-table s)
                                new-heap
                                (class-table s))
                    (cons (list 'REF new-addr) ac)))))

  ; mma :: [c], s --> addr, new-heap
  (defun mma (c s)
    (declare (xargs :measure (cons (+ 1 (len c))
                                   (natural-sum c))))
    (if (<= (len c) 1)

        ; "Base case"  Handles initializing the final dimension
        (mv (len (heap s))
            (bind (len (heap s))
                  (makearray 'T_REF
                             (car c)
                             (init-array 'T_REF (car c))
                             (class-table s))
                  (heap s)))

      ; "Recursive Case"
      (mv-let (heap-prime lst-of-refs)
              (mma2 (car c)
                    (cdr c)
                    s
                    nil)
              (let* ((obj (makearray 'T_REF
                                     (car c)
                                     lst-of-refs
                                     (class-table s)))
                     (new-addr (len heap-prime))
                     (new-heap (bind new-addr obj heap-prime)))
                (mv new-addr new-heap)))))
)
```

Figure 19: Example from an ACL2 model of the Java Virtual Machine. The code in this example creates a multi-dimensional array and returns a new heap with containing the new arrays as well as a reference to the top level array.

1. ⟨`mma2`, {`(not (zp c1))`}, `(mma c2 s)`⟩
2. ⟨`mma2`, {`(not (zp c1))`}, `(mma2 (- c1 1) c2` e e'`)`⟩
3. ⟨`mma`, {`(< 1 (len c))`}, `(mma2 (car c) (cdr c) s nil)`⟩

(a) Precise calling contexts.



(b) CCG.

Figure 20: Precise Calling Contexts and CCG for `mma` and `mma2`.

The first context tells us that the corresponding callsite occurs in the body of `mma2`, under the condition that `(not (zp c1))` is true, and is the expression `(mma c2 s)`. The `zp` function returns true if `c1` is *not* a positive integer, so since the negation is true, `c1` is a positive integer when this call is made. The second context also occurs in the body of `mma2` under the same conditions. It corresponds to a call to `mma2`. The third context corresponds to a call in `mma` under the condition that the length of `c` is more than 1. The call, `(mma2 (car c) (cdr c) s nil)`, splits `c` in two. By the condition, we know that `c` is a list with more than 1 element. The function `car` returns the first element in the list, and `cdr` returns the list resulting in removing the first element.

It turns out that the information contained in these contexts is enough to prove termination (See Theorem 8.3.1 on page 86). Notice that we have effectively eliminated 40 lines of code from our termination analysis, allowing us to focus on those parts of the code that directly relate to the termination analysis.

The next step in our analysis is to build the *Calling Context Graph (CCG)*, which is a labeled directed graph that overapproximates the recursive behavior of the functions. The vertices of the graph are the calling contexts, and an edge from one context to the next signifies that if control reaches the callsite corresponding to the first context, it may reach the context corresponding to the second callsite in the next recursive step. The CCG for our example is given in Figure 20(b).

Consider context 1. It is a call to `mma`, so it will not be able to immediately lead to the execution of contexts 1 or 2, because these occur in the body of `mma2`. However, it may lead to an execution of context 3. To be sure, consider the conditions of contexts 1 and 3. Context 1 is executed when `c1` is a positive integer. The call passes `c2` as parameter `c` of

1. ⟨`mma2`, {`(not (zp c1))`, `(< 1 (len c2))`}, `(mma2 (car c2) (cdr c2) s nil)`⟩
2. ⟨`mma2`, {`(not (zp c1))`}, `(mma2 (- c1 1) c2` e e'`)`⟩

Figure 21: Absorbed Calling Contexts and CCG for `mma` and `mma2`.

`mma` and `s` as the parameter `s`. So, context 3 will be executed in the next recursive step if `c2` is a list whose length is greater than 1. Since the conditions of context 1 do not tell us anything about `c2`, it is entirely possible that `c2` is a list whose length is greater than 1. Therefore, it may be possible to execute the third context immediately after the first, so we add an edge from context 1 to context 3. The reasoning for the other edges is similar.

The key property of CCGs is that any potential infinite sequence of recursive calls is a path through the CCG. Therefore if we can rule out any infinite path through the CCG as an actual execution of our functions, we will have proven termination. Before discussing how this is done, we first demonstrate how the CCG can be manipulated to further simplify the termination analysis.

Notice that in our example, context 1 can only lead to context 3. Also, context 3 can only be reached from context 1. Because of this, we can consider this to be 1 step instead of 2. We do this by using *absorption* (See Section 8.5). In this case, we absorb context 1 into the graph by merging it with context 3, combining their conditions and substituting `c2` for `c`, in the call of context 3, since the call of context 2 passes `c2` for parameter `c`. The resulting contexts and CCG are given in Figure 21.

Notice that by merging contexts 1 and 3, we have further simplified the termination analysis. Now there are only two contexts, and no mention of the function `mma` at all. We have reduced a mutually recursive pair of functions to a termination argument involving just one function. Now we discuss how we prove that no infinite path through the CCG corresponds to an actual execution of the system.

The key to proving this property is the concept of *Calling Context Measures (CCMs)* (see Definition 8.4.1 on page 89), which are simply expressions that map the function parameters of the parent function of a context into some set with a well-founded ordering.

Figure 22: CCMF for `mma` and `mma2`.

The goal, then, is to show that along every infinite path of the CCG, some CCM will decrease infinitely.

In our example, consider the value of `(len c2)`, which is the length of `c2`. In the recursive call of `mma2` for context 2, we pass `(cdr c2)` for parameter `c2`. Since `c2` is a list of length greater than 1, we know that the length of the `cdr` of `c2` will be 1 less than the length of `c2`. Since the length is a natural number, we know that it cannot decrease infinitely. Note also that in context 2, `c2` is passed to parameter `c2`, so the value remains the same.

Consider also the value of `c1` at context 2. We know that if we reach the callsite, `c1` is a positive integer. That means that `(- c1 1)`, the result of decrementing `c1` by 1 is less than `c1`. Since `c1` and `(- c1 1)` are both natural numbers, we know that `c1` cannot decrease forever. Note that for context 1, we pass `(car c2)` to parameter `c1`. Since there is no information about how `(car c2)` compares to `c1`, we do not know how these two values relate.

Putting all this information together we get local information on how the values of `(len c2)` and `c1` change across each recursive call. We gather this information in *Calling Context Measure Functions (CCMFs)* (see Definition 8.4.3 on page 89), which we represent graphically in Figure 22.

Using this information, we do an analysis of the infinite paths of of the CCG. If every infinite path results in an infinite decrease in some CCM, then it cannot be an actual execution of our functions, because the CCMs are compared using a well-founded relation, which has no infinite decreasing sequences.

For our example, consider first any path that visits context 1 infinitely. For such a path, the length of `c2` is never increasing, since it stays the same across the recursive call in context 2. Also, it is infinitely decreasing, since we visit context 1 infinitely often, and

context 1 causes a decrease in the length of `c2`. Therefore, no such path can be an actual computation.

Next consider any path where context 1 is not visited infinitely often. In this case, we will stop visiting context 1 after some finite prefix of the path in question. After this point, we will visit only context 2, which causes the value of `c1` to decrease every time. Therefore, no such path can be an actual computation of our functions.

We have therefore proven the termination of `mma` and `mma2`. It is worth noting that the authors of these functions did prove them terminating in ACL2 using ACL2's measure-based method. The measure they used required an infinite ordinal. For `mma`, the measure was $\omega^{\texttt{(+ 1 (len c))}} + \texttt{(natural-sum c)}$, and the measure for `mma2` was $\omega^{\texttt{(+ 1 (len (cons c1 c2))}} + \texttt{(natural-sum (cons c1 c2))}$, where `cons` adds `c1` onto the front of list `c2`, and `natural-sum` is a function defined by the authors of `mma` and `mma2` specifically for the purpose of proving the termination of these functions. It returns the sum of all the natural number elements of its argument.

There are two things to notice here. First, our analysis can prove these functions terminating with no user assistance, unlike ACL2's measure-based termination analysis, which requires the users to provide a measure in this case. Second, note that our analysis did not require infinite ordinals or the `natural-sum` function to prove termination. It did so with only `c1` and `(len c2)`. One of the strengths if this analysis, then, is that it breaks down the termination problem into simpler components that can be solved with simpler measures than the ones required to prove termination using ACL2's current analysis.

In the remainder of this chapter, we present our termination analysis based on CCGs and CCMs in more detail.

8.2 Classifying Non-Termination

We give useful lemmas for reasoning about the core semantics of FL given in Section 2.1, culminating in a theorem that classifies non-termination in applicative first-order functional languages such as FL. We begin with a lemma that demonstrates that syntactic substitution

and semantic environments are synonymous when the expressions in question are terminating.

Lemma 8.2.1. *Let $e \in Expr$, $n \in \omega$, $\langle e_i \rangle_{i=1}^n \in Expr^n$, $\langle x_i \rangle_{i=1}^n \in Var^n$, $\epsilon \in Env$ map every $x \in ((free(e) - \{x_i \mid 1 \leq i \leq n\}) \cup \bigcup_{i=1}^n free(e_i))$ to a value, and $h \in IHist$ map every function called in e and all the e_i to a function of the appropriate arity. Then $\langle \forall 1 \leq i \leq n :: [\![e_i]\!]^h \epsilon \neq \bot \rangle \Rightarrow [\![e]\!]^h \epsilon[x_i \mapsto [\![e_i]\!]^h \epsilon]_{i=1}^n = [\![e[x_i \mapsto e_i]_{i=1}^n]\!]^h \epsilon$.*

Proof. Let $v_i = [\![e_i]\!]^h \epsilon$ for all $1 \leq i \leq n$. The proof is by induction on the syntactic size of e.

$e \in Val$:

Then $[\![e]\!]^h \epsilon[x_i \mapsto v_i]_{i=1}^n = e = [\![e]\!]^h \epsilon = [\![e[x_i \mapsto e_i]_{i=1}^n]\!]^h \epsilon$

$e \in Var - \{x_i\}_{i=1}^n$:

Then $[\![e]\!]^h \epsilon[x_i \mapsto v_i]_{i=1}^n = \epsilon[x_i \mapsto v_i]_{i=1}^n(e) = \epsilon(e) = [\![e]\!]^h \epsilon = [\![e[x_i \mapsto e_i]_{i=1}^n]\!]^h \epsilon$.

$e = x_i$ **for some** $1 \leq i \leq n$:

Then $[\![e]\!]^h \epsilon[x_i \mapsto v_i]_{i=1}^n = \epsilon[x_i \mapsto v_i]_{i=1}^n(e) = v_i = [\![e_i]\!]^h \epsilon = [\![e[x_i \mapsto e_i]_{i=1}^n]\!]^h \epsilon$.

$e =$ `(let ((`y_1 e'_1`) ... (`y_m e'_m`)) `e'`)` :

By the induction hypothesis, $\left[\!\left[e'_j\right]\!\right]^h \epsilon[x_i \mapsto v_i]_{i=1}^n = \left[\!\left[e'_j[x_i \mapsto e_i]_{i=1}^n\right]\!\right]^h \epsilon$ for all $1 \leq j \leq m$. Let u_j denote this value for each j. If any $u_i = \bot$ for any $1 \leq i \leq m$, then $[\![e]\!]^h \epsilon[x_i \mapsto v_i]_{i=1}^n = [\![e[x_i \mapsto e_i]_{i=1}^n]\!]^h \epsilon = \bot$ by definition. Otherwise we have

$$
\begin{aligned}
& & [\![e]\!]^h \epsilon[x_i \mapsto v_i]_{i=1}^n \\
& \{ \textit{Semantics of } \texttt{let} \} & = [\![e']\!]^h \epsilon[x_i \mapsto v_i]_{i=1}^n [y_i \mapsto u_j]_{j=i}^m \\
& \{ \textit{All } x \in \textit{free}(e') \textit{ bound in } \texttt{let} \} & = [\![e']\!]^h \epsilon[y_i \mapsto u_j]_{j=i}^m \\
& \{ \textit{Semantics of } \texttt{let}\textit{, Def. of substitution} \} & = [\![e[x_i \mapsto e_i]_{i=1}^n]\!]^h \epsilon
\end{aligned}
$$

$e =$ `(if` e'_1 e'_2 e'_3`)` : By the induction hypothesis, we have that $\left[\!\left[e'_j\right]\!\right]^h \epsilon[x_i \mapsto v_i]_{i=1}^n = \left[\!\left[e'_j[x_i \mapsto e_i]_{i=1}^n\right]\!\right]^h \epsilon$ for $1 \leq j \leq 3$. The result then follows from the semantics of `if`.

$e =$ `(`g e'_1 ... e'_m`)` By the induction hypothesis, $\left[\!\left[e'_j\right]\!\right]^h \epsilon[x_i \mapsto v_i]_{i=1}^n = \left[\!\left[e'_j[x_i \mapsto e_i]_{i=1}^n\right]\!\right]^h \epsilon$ for $1 \leq j \leq m$. The result then follows from the semantics of function calls.

□

The following corollary is directly applicable to `let` expressions, and says that the syntactic let substitutions correspond in some way to the semantic execution of `let` expressions.

Corollary 8.2.1. *Let* $e =$ `(let ((`y_1 e_1`) ... (`y_n e_n`)) `e_{n+1}`)` $\in$ *Expr*, $\epsilon \in$ *Env map every* $x \in$ *free*(e) *to a value, and* $h \in$ *IHist map every function called in* e *to a function of the appropriate arity. Then* $\langle \forall 1 \leq i \leq n :: [\![e_i]\!]^h \epsilon \neq \bot \rangle \Rightarrow [\![e]\!]^h \epsilon = [\![e|_{n+1}]\!]^h \epsilon$.

Proof. The results follow directly from Lem. 8.2.1 and the semantics of `let`. □

Another direct result of Lem. 8.2.1 is the following, which we will use later when reasoning about function calls.

Corollary 8.2.2. *Let* $e \in$ *Expr*, $n \in \omega$, $\langle e_i \rangle_{i=1}^n \in$ *Expr*^{n}, $\{x_i \mid 1 \leq i \leq n\} \supseteq$ *free*(e), $\epsilon \in$ *Env, and* $h \in$ *IHist map every function called in* e *and all the* e_i *to a function of the appropriate arity. Then* $\langle \forall 1 \leq i \leq n :: v_i = [\![e_i]\!]^h \epsilon \neq \bot \rangle \Rightarrow [\![e]\!]^h [x_i \mapsto v_i]_{i=1}^n = [\![e[x_i \mapsto e_i]_{i=1}^n]\!]^h \epsilon$.

Proof. Follows directly from Lem. 8.2.1. □

For the next lemma and theorem, we need the concept of *let-adjusted depth* of an expression e. Intuitively, this is the depth of the expression resulting from applying all of the let substitutions suggested by e. More formally,

Definition 8.2.1. The *let-adjusted depth* of an expression, e is defined recursively as follows:

- $lad(x) = lad(v) = 0$
- $lad(e) = 1 + \max\{lad(e|_i) \mid \langle \exists q :: iq \in Pos(e) \rangle\}$.

The first lemma gives a sufficient condition for non-termination.

Lemma 8.2.2. *Let* $h \in$ *IHist*, $\epsilon \in$ *Env, and* $e \in$ *Expr. If there exists* $q \in Pos(e)$ *such that* $\mathcal{H}^h [\![gov(e,q)]\!] \epsilon$ *and* $[\![e|_q]\!]^h \epsilon = \bot$, *then* $[\![e]\!]^h \epsilon = \bot$.

Proof. Suppose that there exists $q \in Pos(e)$ such that $\mathcal{H}^h [\![gov(e,q)]\!] \epsilon$ and $[\![e|_q]\!]^h \epsilon = \bot$. We refer to this as the original hypothesis. We prove that $[\![e]\!]^h \epsilon = \bot$ by induction on q.

For the base case, if $q = \epsilon$, the theorem holds trivially.

For the induction step, suppose that $q = ip \in Pos(e)$. Consider the following cases.

$e =$ `(`$g\ e_1\ \ldots\ e_m$`)` : By definition, $e_i|_p = e|_{ip}$ and $gov(e_i, p) = gov(e, ip)$. Therefore, by the original hypothesis, $\mathcal{H}^h [\![gov(e_i,p)]\!] \epsilon$ and $[\![e_i|_p]\!]^h \epsilon = \bot$. By the induction hypothesis, this means that $[\![e_i]\!]^h \epsilon = \bot$. Therefore, $[\![e]\!]^h \epsilon = \bot$ by the semantics of function calls.

$e =$ `(let ((`$x_1\ e_1$`) ... (`$x_m e_m$`)) `e_{m+1}`)` : Then consider the following two cases:

- $i \neq m+1$: Then by definition, $e|_{ip} = e_i|_p$ and $gov(e, ip) = gov(e_i, p)$. Therefore, by the original hypothesis, $\mathcal{H}^h [\![gov(e_i,p)]\!] \epsilon$ and $[\![e_i|_p]\!]^h \epsilon = \bot$. By the induction hypothesis, this means that $[\![e_i]\!]^h \epsilon = \bot$. Therefore, $[\![e]\!]^h \epsilon = \bot$ by the semantics of `let`.
- $i = m+1$: By definition, $e|_{(m+1)p} = e_{m+1}|_p$. Also, by definition, $gov(e, (m+1)p) = gov(e_{m+1}, p)$. Therefore, $\mathcal{H}^h [\![gov(e_{m+1},p)]\!] \epsilon$ and $[\![e_{m+1}|_p]\!]^h \epsilon = \bot$. By the induction hypothesis, this means that $[\![e_{m+1}]\!]^h \epsilon = \bot$. By Lem. 8.2.1, $[\![e_{m+1}]\!]^h \epsilon[x_j \mapsto [\![e_j]\!]^h \epsilon]_{j=1}^m = \bot$. Therefore, $[\![e]\!]^h \epsilon = \bot$ by the semantics of `let`.

$e =$ `(if `$e_1\ e_2\ e_3$`)` : Let $q = ip$. Consider the following three cases:

- $i = 1$: Then by definition, $e|_{ip} = e_i|_p$ and $gov(e, ip) = gov(e_i, p)$. Therefore, by the original hypothesis, $\mathcal{H}^h [\![gov(e_i,p)]\!] \epsilon$ and $[\![e_i|_p]\!]^h \epsilon = \bot$. By the induction hypothesis, this means that $[\![e_i]\!]^h \epsilon = \bot$. Therefore, $[\![e]\!]^h \epsilon = \bot$ by the semantics of `if`.
- $i = 2$: Then by definition, $e|_{ip} = e_i|_p$ and $gov(e, ip) = \{e_1\} \cup gov(e_i, p)$. Since $gov(e_i, p) \subseteq gov(e, ip)$, $\mathcal{H}^h [\![gov(e,p)]\!] \epsilon \Rightarrow \mathcal{H}^h [\![gov(e_i,p)]\!] \epsilon$. Therefore, by the original hypothesis, $\mathcal{H}^h [\![gov(e_i,p)]\!] \epsilon$ and $[\![e_i|_p]\!]^h \epsilon = \bot$. By the induction hypothesis, this means that $[\![e_i]\!]^h \epsilon = \bot$. Therefore, $[\![e]\!]^h \epsilon = \bot$ by the semantics of `if`.

$i = 3$: Then by definition, $e|_{ip} = e_i|_p$ and $gov(e, ip) = \{\texttt{(not } e_1\texttt{)}\} \cup gov(e_i, p)$. Since $gov(e_i, p) \subseteq gov(e, ip)$, $\mathcal{H}^h [\![gov(e, p)]\!] \epsilon \Rightarrow \mathcal{H}^h [\![gov(e_i, p)]\!] \epsilon$. Therefore, $\mathcal{H}^h [\![gov(e_i, p)]\!] \epsilon$ and $[\![e_i|_p]\!]^h \epsilon = \perp$. By the induction hypothesis, this means that $[\![e_i]\!]^h \epsilon = \perp$. Therefore, $[\![e]\!]^h \epsilon = \perp$ by the semantics of `if`.

□

The following theorem strengthens the previous lemma into a necessary and sufficient condition for termination in FL. It basically says that the execution of an expression with a given history and environment is non-terminating exactly when the execution of that expression reaches a sub-expression that is a non-terminating function call.

Theorem 8.2.1. *Let h be an intermediate history and ϵ be an environment. Then for any expression e, $[\![e]\!]^h \epsilon = \perp$ if and only if there is a position, $q \in Pos(e)$, such that $e|_q = \texttt{(}f\ e'_1\ \ldots\ e'_n\texttt{)}$, $\mathcal{H}^h [\![gov(e, q)]\!] \epsilon$, $[\![e'_i]\!]^h \epsilon \neq \perp$ for all $1 \leq i \leq n$, and $[\![e|_q]\!]^h \epsilon = \perp$.*

Proof. We prove the two directions separately.

($\Leftarrow$): This direction follows directly from Lemma 8.2.2 on page 80.

($\Rightarrow$): Suppose that $[\![e]\!]^h \epsilon = \perp$. We refer to this as the original hypothesis. We prove that there exists $q \in Pos(e)$ as described in the theorem. The proof is by induction on the $lad(e)$. Consider the following cases.

$e = \texttt{(}g\ e_1\ \ldots\ e_m\texttt{)}$: If $\exists 1 \leq j \leq m$ such that $[\![e_j]\!]^h \epsilon = \perp$, then by the induction hypothesis, there exists $q \in Pos(e_j)$ such that $e_j|_q = \texttt{(}f\ e'_1\ \ldots\ e'_n\texttt{)}$, $\mathcal{H}^h [\![gov(e_j, q)]\!] \epsilon$, $[\![e'_i]\!]^h \epsilon \neq \perp$ for all $1 \leq i \leq n$, and $[\![e_j|_q]\!]^h \epsilon = \perp$. By definition, $e_j|_q = e|_{jq}$ and $gov(e, jq) = gov(e_j, q)$. Therefore, $e|_{jq} = \texttt{(}f\ e'_1\ \ldots\ e'_n\texttt{)}$, $\mathcal{H}^h [\![gov(e, jq)]\!] \epsilon$, $[\![e'_i]\!]^h \epsilon \neq \perp$ for all $1 \leq i \leq n$, and $[\![e|_{jq}]\!]^h \epsilon = \perp$.

Otherwise the theorem trivially holds for $\epsilon \in Pos(e)$.

$e = \texttt{(let ((}x_1\ e_1\texttt{)}\ \ldots\ \texttt{(}x_m e_m\texttt{))}\ e_{m+1}\texttt{)}$: Then consider the following two cases.

$\exists 1 \leq j \leq m$ **such that** $[\![e_j]\!]^h \epsilon = \perp$: Then by the induction hypothesis, there exists $q \in Pos(e)$, such that $e_j|_q = \texttt{(}f\ e'_1\ \ldots\ e'_n\texttt{)}$, $\mathcal{H}^h [\![gov(e_j, q)]\!] \epsilon$, $[\![e'_i]\!]^h \epsilon \neq \perp$ for

all $1 \leq i \leq n$, and $[\![e_j|_q]\!]^h\,\epsilon = \bot$. By definition, $e_j|_q = e|_{jq}$ and $gov(e, jq) = gov(e_j, q)$. Therefore, $e|_{jq} = (f\ \ e'_1\ \ \ldots\ \ e'_n)$, $\mathcal{H}^h\,[\![gov(e, jq)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $1 \leq i \leq n$, and $[\![e|_{jq}]\!]^h\,\epsilon = \bot$.

$\forall 1 \leq j \leq m$, $[\![e_j]\!]^h\,\epsilon \neq \bot$: Let $e' = e|_{m+1}$. Then

$$\begin{aligned}
& [\![e|_{m+1}]\!]^h\,\epsilon \\
\{\ \textit{Def. of } e|_{m+1}\ \} \quad &= \quad [\![e_{m+1}[x_i \mapsto e_i]]\!]^h\,\epsilon \\
\{\ \textit{Cor. 8.2.1}\ \} \quad &= \quad [\![e_{m+1}]\!]^h\,\epsilon[x_i \mapsto [\![e_i]\!]^h\,\epsilon]_{i=1}^m \\
\{\ \textit{Semantics of } \texttt{let}\ \} \quad &= \quad [\![e]\!]^h\,\epsilon \\
\{\ \textit{original hypothesis}\ \} \quad &= \quad \bot
\end{aligned}$$

By the induction hypothesis, there exists $q \in Pos(e|_{m+1})$, such that $(e|_{m+1})|_q = (f\ \ e'_1\ \ \ldots\ \ e'_n)$, $\mathcal{H}^h\,[\![gov(e|_{m+1}, q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $1 \leq i \leq n$, and also $[\![(e|_{m+1})|_q]\!]^h\,\epsilon = \bot$. By definition, $e|_{(m+1)q} = (e|_{m+1})|_q$ and $gov(e, (m+1)q) = gov(e|_{m+1}, q)$. Therefore, $e|_{(m+1)q} = (f\ \ e'_1\ \ \ldots\ \ e'_n)$, $\mathcal{H}^h\,[\![gov(e, (m+1)q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $1 \leq i \leq n$, and $[\![e|_{(m+1)q}]\!]^h\,\epsilon = \bot$.

$e = (\texttt{if}\ \ e_1\ \ e_2\ \ e_3)$: Consider the following three cases:

$[\![e_1]\!]^h\,\epsilon = \bot$: Then by the induction hypothesis, there exists $q \in Pos(e_1)$ such that $e_1|_q = (f\ \ e'_1\ \ \ldots\ \ e'_n)$, $\mathcal{H}^h\,[\![gov(e_1, q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $1 \leq i \leq n$, and $[\![e_1|_q]\!]^h\,\epsilon = \bot$. By definition, $e_1|_q = e|_{1q}$ and $gov(e, 1q) = gov(e_1, q)$. Therefore, $e|_{1q} = (f\ \ e'_1\ \ \ldots\ \ e'_n)$, $\mathcal{H}^h\,[\![gov(e, 1q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $1 \leq i \leq n$, and $[\![e|_{1q}]\!]^h\,\epsilon = \bot$.

$[\![e_1]\!]^h\,\epsilon \notin \{\bot, \texttt{nil}\}$: Then by our original hypothesis and the semantics of `if`, $[\![e_2]\!]^h\,\epsilon = \bot$. By the induction hypothesis, there exists $q \in Pos(e_2)$ such that $e_2|_q = (f\ \ e'_1\ \ \ldots\ \ e'_n)$, $\mathcal{H}^h\,[\![gov(e_2, q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $1 \leq i \leq n$, and $[\![e_2|_q]\!]^h\,\epsilon = \bot$. By definition, $e_2|_q = e|_{2q}$, and $gov(e, 2q) = \{e_1\} \cup gov(e_2, q)$. Since $[\![e_1]\!]^h\,\epsilon \notin \{\bot, \texttt{nil}\}$ by the current case hypothesis, and $\mathcal{H}^h\,[\![gov(e_2, q)]\!]\,\epsilon$, we know that $\mathcal{H}^h\,[\![gov(e, 2q)]\!]\,\epsilon$. Therefore, $e|_{2q} = (f\ \ e'_1\ \ \ldots\ \ e'_n)$, $\mathcal{H}^h\,[\![gov(e, 2q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $2 \leq i \leq n$, and $[\![e|_{2q}]\!]^h\,\epsilon = \bot$.

$[\![e_1]\!]^h\,\epsilon = \texttt{nil}$: Then by our original hypothesis and the semantics of `if`, $[\![e_3]\!]^h\,\epsilon = \bot$. By the induction hypothesis, there exists $q \in Pos(e_3)$ such that $[\![e_3]\!]^h\,q = \texttt{(}f\ \ e'_1\ \ \ldots\ \ e'_n\texttt{)}$, $\mathcal{H}^h\,[\![gov(e_3, q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $1 \leq i \leq n$, and $[\![e_3|_q]\!]^h\,\epsilon = \bot$. By definition, $e_3|_q = e|_{3q}$, and $gov(e, 3q) = \{\texttt{(not}\ \ e_1\texttt{)}\} \cup gov(e_3, q)$. Since $[\![e_1]\!]^h\,\epsilon = \texttt{nil}$ by the current case hypothesis, and $\mathcal{H}^h\,[\![gov(e_3, q)]\!]\,\epsilon$, $\mathcal{H}^h\,[\![gov(e, 3q)]\!]\,\epsilon$. Therefore, $e|_{3q} = \texttt{(}f\ \ e'_1\ \ \ldots\ \ e'_n\texttt{)}$, $\mathcal{H}^h\,[\![gov(e, 3q)]\!]\,\epsilon$, $[\![e'_i]\!]^h\,\epsilon \neq \bot$ for all $2 \leq i \leq n$, and $[\![e|_{3q}]\!]^h\,\epsilon = \bot$.

□

By this theorem, we can see that non-termination in our target class of programming languages occurs exactly when there is infinite recursion, *i.e.*, when there is an infinite sequence of recursive calls that never "bottoms out" and returns a value. Therefore, if we can prove that every sequence of recursive calls eventually reaches a base case, we will prove termination.

8.3 Calling Context Graphs

In this section, we introduce calling context graphs (CCGs) and related notions. We also show how CCGs can be used reason about program termination.

Definition 8.3.1. A *calling context* is a triple, $\langle f, G, e\rangle$, where f is the name of a function defined in d, G is a set of expressions whose free variables are all parameters of f, and e is a call of a function in d whose free variables are all parameters of f. This is a *semi-precise calling context* for callsite $e^f|_p$ if $e = e^f|_p$ for some $p \in Pos(e)$, where e^f is the body of f and $\langle \forall \epsilon \in Env :: \mathcal{H}^h\,[\![gov(e^f, p)]\!]\,\epsilon\rangle \Rightarrow \mathcal{H}^h\,[\![G]\!]\,\epsilon\rangle$. It is a *precise calling context* for callsite $e^f|_p$ if it is semi-precise for $e^f|_p$ and $\langle \forall \epsilon \in Env :: \mathcal{H}^h\,[\![gov(e^f, p)]\!]\,\epsilon \equiv \mathcal{H}^h\,[\![G]\!]\,\epsilon\rangle$ The elements of the triple, f, G, and e are referred to as the *function*, *conditions*, and *call* of the context, respectively.

We sometimes refer to a calling context simply as a context. Intuitively, the conditions of a precise context can be thought of as the governors of the call in the function body, and the conditions of a semi-precise context can be thought of as a subset

```
(defun f (x)
  (cond ((or (not (integerp x))
             (= x 0))
         0)
        ((< x 0) (f (+ x 1)))
        (t       (f (- x 1)))))
```

1. ⟨`f`, {`(integerp x)`, `(< x 0)`}, `(f (+ x 1))`⟩
2. ⟨`f`, {`(integerp x)`, `(> x 0)`}, `(f (- x 1))`⟩

Figure 23: Definitions, contexts, and minimal complete CCG for `f`

```
(defun dec (n)
  (if (zp n)
      255
    (- n 1)))

(defun foo (i j)
  (if (= i 1)
      (if (= j 1)
          0
        (foo (dec j)
             (dec j)))
    (foo (dec i) j)))
```

1. ⟨`foo`, {`(= i 1)`, `(not (= j 1))`}, `(foo (dec j) (dec j))`⟩
2. ⟨`foo`, {`(not (= i 1)`}, `(foo (dec i) j)`⟩

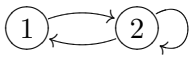


Figure 24: Definitions, contexts, and minimal complete CCG for `foo`.

of the governors. We use the less restrictive notation in the definition to allow us to simplify the governors when appropriate. Definitions and precise contexts for two examples are given in Figures 23 and 24. Note that we already include some simplification of the conditions. For example, the governors for the first call of `f` are actually {`(not (or (not (integerp x)) (= x 0)))`, `(< x 0)`}.

Definition 8.3.2. A set of calling contexts, C, is said to be *(semi-)complete for* d there is a surjection, f, from the recursive callsites of d to C such that, for all callsites, s, $f(s)$ is (semi-)precise for s.

Since d is fixed throughout this chapter, we sometimes refer to a set of contexts that is (semi-)complete for d as just being (semi-)complete. We now introduce the notion of a well-formed sequence of contexts, a notion that is strongly related to termination in *FL*.

Definition 8.3.3. Let $c = \langle\langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})\rangle\rangle_i$ be a sequence of calling contexts and $\vec{v} \in Val^{ar(f_1)}$. Then c is a *well-formed sequence of calling contexts with witness* $\vec{v}$ if there exists a sequence of environments, $\langle \epsilon^{\vec{v}}_{c,i} \rangle_i$ such that $\epsilon^{\vec{v}}_{c,1} = [x_i^{f_1} \mapsto v_i]_{i=1}^{ar(f_1)}$ and for all i, the following conditions hold:

1. for all $1 \leq j \leq ar(f_i)$, $[\![e_{i,j}]\!]^h\, \epsilon^{\vec{v}}_{c,i} \neq \bot$,

2. $\mathcal{H}^h\, [\![G_i]\!]\, \epsilon^{\vec{v}}_{c,i}$, and

3. $\epsilon^{\vec{v}}_{c,i+1} = [x_j^{f_{i+1}} \mapsto [\![e_{i,j}]\!]^h\, \epsilon^{\vec{v}}_{c,i}]_{j=1}^{ar(f_{i+1})}$.

We just say that c is well-formed to mean that c is well-formed with some unspecified witness.

We use the notation $\epsilon^{\vec{v}}_{c,i}$ introduced in the above definition throughout the paper. If the sequence of calling contexts is clear from the context, we use $\epsilon^{\vec{v}}_i$. Termination in *FL* can be expressed in terms of well-formed sequences, as we see in the next theorem.

Theorem 8.3.1. *If every well-formed sequence of a semi-complete set of contexts, C, is finite, then the functions of d terminate on all inputs.*

Proof. We prove this theorem by proving the contrapositive. Suppose $h.f_1\vec{v} = \bot$. By our initial hypothesis and the semantics of our language, $[\![e^{f_1}]\!]^h\, \epsilon^{\vec{v}}_1 = \bot$. By Theorem 8.2.1 on page 82, there is a position, $p \in Pos(e^{f_1})$, such that $e^{f_1}|_p = (f_2\ e_{1,1}\ \ldots\ e_{1,ar(f_2)})$, $[\![e_{1,i}]\!]^h\, \epsilon^{\vec{v}}_1 \neq \bot$ for all $1 \leq i \leq ar(f_2)$, and $\mathcal{H}^h\, [\![gov(e^{f_1}, p)]\!]\, \epsilon^{\vec{v}}_1$. By our initial hypothesis, there exists a semi-precise context for $e^{f_1}|_p$, $\langle f_1, G_1, (f_2\ e_{1,1}\ \ldots\ e_{1,ar(f_2)})\rangle$. By the definition of semi-precise and the fact that $\mathcal{H}^h\, [\![gov(e^{f_1}, p)]\!]\, \epsilon^{\vec{v}}_1$, we have $\mathcal{H}^h\, [\![G_1]\!]\, \epsilon^{\vec{v}}_1$, Therefore, by definition, $\langle\langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})\rangle\rangle_{i=1}^{1}$ is a well-formed sequence of semi-precise contexts with witness $\vec{v}$ such that the last function call evaluates to $\bot$.

Now we show that if $\langle\langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})\rangle\rangle_{i=1}^{k}$ is a well-formed sequence of the contexts of C with witness $\vec{v}$ such that $[\![f_{k+1}(e_{k,1}, \ldots, e_{k,ar(f_k+1)})]\!]^h\, \epsilon^{\vec{v}}_k = \bot$, it can be extended to such a sequence of length $k+1$ with the same witness.

By the semantics of our language and our inductive hypothesis, $[\![e^{f_{k+1}}]\!]^h\, \epsilon^{\vec{v}}_{k+1} = \bot$. By Theorem 8.2.1 on page 82, there is $p \in Pos(e^{f_1})$, such that $e^{f_{k+1}}|_p$ is an expression of the form $(f_{k+2}\ e_{k+1,1}\ \ldots\ e_{k+1,ar(f_{k+2})})$, $[\![e_{k+1,i}]\!]^h\, \epsilon^{\vec{v}}_{k+1} \neq \bot\ \mathcal{H}^h\, [\![gov(e^{f_{k+1}}, p)]\!]\, \epsilon^{\vec{v}}_1$ for all $1 \leq i \leq ar(f_{i+2})$, and $[\![e^{f_{k+1}}|_p]\!]^h\, \epsilon^{\vec{v}}_1 = \bot$. By the initial condition, there exists a context in C of the form $\langle f_{k+1}, G_{k+1}, (f_{k+2}\ e_{k+1,1}\ \ldots\ e_{k+1,ar(f_{k+2})})\rangle$ that is semi-precise for $e^{f_{k+1}}|_p$, By the definition of a semi-precise context, and the fact that $\mathcal{H}^h\, [\![gov(e^{f_{k+1}}, p)]\!]\, \epsilon^{\vec{v}}_1$, we know that

$\mathcal{H}^h [\![G_{k+1}]\!] \epsilon_1^{\vec{v}}$. Therefore, by definition, the sequence $\langle\langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})\rangle\rangle_{i=1}^{k+1}$ is a well-formed sequence of semi-precise contexts with witness $\vec{v}$ such that the last function call evaluates to $\perp$. □

Theorem 8.3.2. *If C is a complete set of contexts, then the functions of d terminate on all inputs iff every well-formed sequence of the contexts of C are finite.*

Proof. We prove each direction of the proof separately.

($\Leftarrow$): We start with the backward direction, which follows from Theorem 8.3.1 on the previous page and the fact that every precise calling context is semi-precise.

($\Rightarrow$): For the forward direction, we prove the contrapositive.

Let $d = \langle (\texttt{defun}\ f_1\ (x_1^{f_1}\ \ldots\ x_{ar(f_1)}^{f_1})\ e_1)\rangle_{i=1}^n$, and $nextfs_d$ be defined for d as in Figure 1 on page 6. Let $\phi_i^0(\vec{v}) = \perp$ for all $1 \leq i \leq n$, and $\langle \phi_i^{j+1}\rangle_{i=1}^n = nextfs_d \langle \phi_i^j \rangle_{i=1}^n$. Finally, for all $0 \leq j$, let $h_j = H[f_i \mapsto \phi_i^j]_{i=1}^n$. Note that $h = \lim_{j\in\omega} h_j$.

Let $W = \{\langle \vec{v}, \langle f_i, G_i, e_i\rangle_{i=1}^\omega\rangle \mid \langle f_i, G_i, e_i\rangle_{i=1}^\omega$ is well-founded with witness $\vec{v}\}$. We inductively prove that, for all $i \geq 0$, $(h_i.f_1)\vec{v} = \perp$ for all $\langle \vec{v}, \langle f_i, G_i, e_i\rangle_{i=1}^\omega\rangle \in W$.

Base Case: By definition, $h_0.f\vec{v} = \perp$ for all $\langle \vec{v}, \langle f_i, G_i, e_i\rangle_{i=1}^\omega\rangle \in W$.

Induction Step: Suppose that the condition holds for i. We show that it holds for $i+1$. Let $\langle \vec{v}_1, \langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})\rangle_{i=1}^\omega\rangle \in W$ and $\vec{v}_2 = \langle [\![e_{1,k}]\!]^{h_i} \epsilon_1^{\vec{v}_1}\rangle_{k=1}^{n_1}$. Then by definition, $\langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})\rangle_{i=2}^\omega$ is well-formed with witness $\vec{v}_2$, and so $\langle \vec{v}_2, \langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})\rangle_{i=2}^\omega\rangle \in W$. Therefore, by the induction hypothesis, $[\![(f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})]\!]^{h_i} \epsilon_1^{\vec{v}_1} = \perp$.

By the semantics of function calls, $h_{i+1}.f_1(\vec{v}_1) = [\![e^{f_1}]\!]^{h_i} \epsilon_1^{\vec{v}_1}$. Since C is complete, there exists $p \in Pos(e^{f_1})$ such that $e^{f_1}|_p = (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})})$ and $\mathcal{H}^h [\![gov(e,p)]\!] \epsilon_1^{\vec{v}_1} \equiv \mathcal{H}^{h_i} [\![G_1]\!] \epsilon_1^{\vec{v}_1}$, which is true by the definition of well-formed sequences. Therefore, by Theorem 8.2.1 on page 82, $h_{i+1}.f_1(\vec{v}_1) = \perp$.

This concludes the inductive proof. By this fact and the semantics of function definition in our language, if there is an infinite sequence of precise contexts, the functions of d do not terminate for all inputs. □

We now define the notion of a *calling context graph* and show that it is a conservative

approximation of the well-formed sequence of contexts.

Definition 8.3.4. A *calling context graph (CCG)*, is a directed graph, $\mathcal{G} = (C, E)$, where C is a set of calling contexts, and for any pair of contexts $c_1, c_2 \in C$, if the sequence (c_1, c_2) is well-formed, then $\langle c_1, c_2 \rangle \in E$. If C is a (semi-)complete set of contexts, then $\mathcal{G}$ is called a *(semi-)complete CCG of d*.

A minimal complete CCG for function `f` in Figure 23 on page 85 is shown in the same figure. Note that there is no edge between the two contexts. This is because if `x` is a positive integer, then decrementing `x` by 1 will not lead to a negative integer. Likewise, adding 1 to `x` if it is a negative integer cannot produce a positive integer. Notice that this mirrors the looping behaviors of the function. Figure 24 on page 85 contains a minimal complete CCG for function `foo`. Notice that if the first context of `foo` is reached, `foo` calls itself, passing in `(dec j)` for both arguments. Since `(dec j)` cannot simultaneously be both equal to 1 and not equal to 1, it is impossible to immediately reach context 1 again. However both contexts can reach context 2, and context 2 can reach context 1. We now prove that CCGs are conservative approximations of well-formed sequences of calling contexts.

Lemma 8.3.1. *Given a CCG, $\mathcal{G} = (C, E)$, every well-formed sequence of calling contexts of C is a path in $\mathcal{G}$.*

Proof. Suppose $\langle\langle f_i, G_i, (f_{i+1}\ \ e_{i,1}\ \ \ldots\ \ e_{i,n_i+1})\rangle\rangle_i$ is a well-formed sequence of contexts with witness $\vec{v}$. Clearly, the sequence $\langle\langle f_i, G_i, (f_{i+1}\ \ e_{i,1}\ \ \ldots\ \ e_{i,n_i+1})\rangle\rangle_{i=1}^{2}$ is well-formed with witness $\vec{v}$. Therefore, there is an edge between the first two contexts of the sequence in the callgraph. For any $1 \leq i$, let $\vec{v}_i = \langle [\![e_{i,j}]\!]^h\, \epsilon_i^{\vec{v}} \rangle_{i=1}^{n_{i+1}}$. Then by definition, $\epsilon_k^{\vec{v}} = \epsilon_{i-k+1}^{\vec{v}_i}$ for all $k > i$. This makes $\vec{v}_j$ the witness for the sequence $\langle\langle f_i, G_i, (f_{i+1}\ \ e_{i,1}\ \ \ldots\ \ e_{i,n_i+1})\rangle\rangle_{i=j+1}^{j+2}$ making this an edge in our context graph for all $j \geq 1$. Thus, the original sequence, $\langle\langle f_i, G_i, (f_{i+1}\ \ e_{i,1}\ \ \ldots\ \ e_{i,n_i+1})\rangle\rangle_i$ is a path in the context graph. □

Note that the converse of the above lemma does not hold. This is because the definition of a CCG only requires local reachability whereas a well-formed sequence of contexts requires that the entire sequence correspond to a single computation. As a result, a CCG is an

abstraction of the actual system. We use CCGs to perform a local analysis which, if successful, can determine that the definitions terminate. To do this, we start by assigning calling context measures to contexts in the CCG.

8.4 Calling Context Measures and Termination

Definition 8.4.1. Given a calling context, $c = \langle f, G, e\rangle$, and a set $S \subseteq \mathit{Val}$, the set of *calling context measures (CCMs) for c over S*, denoted CCM_S^c is the set $\{e \in \mathit{Expr} \mid \langle \forall \epsilon \in \mathit{Env} : \mathcal{H}^h [\![G]\!] \epsilon : \mathit{free}(e) \subseteq \{x_1^f, \ldots x_{ar(f)}^f\} \quad \wedge \quad [\![e]\!]^h \epsilon \in S\rangle\}$.

CCMs simply map the parameters of a function into some set. We annotate CCGs with CCMs using a CCM annotation.

Definition 8.4.2. Given a set of calling contexts, C, and a set, $S \subseteq \mathit{Val}$, a *CCM annotation for C over S* is a function $m : C \to \mathcal{P}(\mathit{Expr})$ such that $\langle \forall c \in C :: m(c) \subseteq CCM_S^c\rangle$.

Now we create a mechanism for comparing the CCM of two adjacent contexts in a CCG.

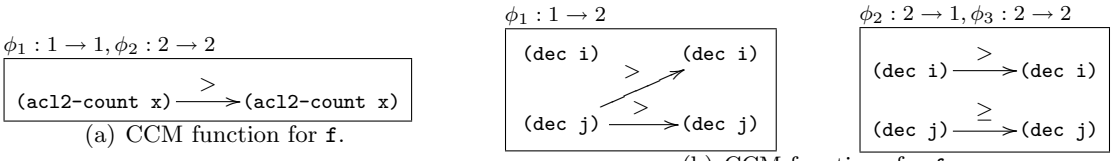
Definition 8.4.3. Let $\mathcal{G} = (C, E)$ be a CCG with $e = \langle c_1, c_2\rangle \in E$, such that $c_1 = \langle f_1, G_1, e_1\rangle$ and $c_2 = \langle f_2, G_2, e_2\rangle$, where $e_1 = (f_2 \ e_{1,1} \ \ldots \ e_{1,ar(f_2)})$. Let $\langle S, \prec\rangle$ be a well-founded structure, and $m : C \to \mathcal{P}(\mathit{Expr})$ be a CCM annotation for C over S. Then a *CCM function (CCMF) for e with annotation m and order* $\prec$ is a function $\phi_{c_2}^{c_1} : m(c_1) \times m(c_2) \to \{>, \geq, \times\}$ such that $\phi(s_1, s_2) = >$ only if $\langle \forall \epsilon \in \mathit{Env} : \langle \forall i : 1 \leq i \leq ar(f_2) : [\![e_{1,i}]\!]^h \epsilon \neq \perp\rangle \quad \wedge \quad \mathcal{H}^\epsilon [\![G_1 \cup G_2\sigma_{e_1}]\!] : [\![s_1]\!]^h \epsilon \succ [\![s_2\sigma_{e_1}]\!]^h \epsilon\rangle$ and $\phi(s_1, s_2) = \geq$ only if $\langle \forall \epsilon \in \mathit{Env} : \langle \forall i : 1 \leq i \leq ar(f_2) : [\![e_{1,i}]\!]^h \epsilon \neq \perp\rangle \quad \wedge \quad \mathcal{H}^\epsilon [\![G_1 \cup G_2\sigma_{e_1}]\!] : [\![s_1]\!]^h \epsilon \succeq [\![s_2\sigma_{e_1}]\!]^h \epsilon\rangle$.

We represent CCM functions for $\langle c_1, c_2\rangle$ graphically with a box containing the CCMs for c_1, c_2 on the left and right, respectively. An edge is drawn from s_1, a left CCM, to s_2, a right CCM, with the label $\phi(s_1, s_2)$ iff it is $>$ or $\geq$. If $\phi(s_1, s_2)$ is $\times$, no edge is drawn.

We now consider some examples. For the function `f` in Figure 23 on page 85, we use the `acl2-count` function in Figure 25 on the next page applied to `f`'s parameter, `x`, as the only CCM for both contexts. The range of `acl2-count` is the set of natural numbers, and the function is designed to mirror common induction schemes, *e.g.*, induction on the size of a list. Notice that for each context in our example, the CCM decreases for all values

```
(defun acl2-count (x)
  (cond ((consp x)
         (+ 1 (acl2-count (car x))
              (acl2-count (cdr x))))
        ((integerp x)
         (integer-abs x))
        ((rationalp x)
         (+ (integer-abs (numerator x))
            (denominator x)))
        ((complex-rationalp x)
         (+ 1 (acl2-count (realpart x))
              (acl2-count (imagpart x))))
        ((stringp x) (length x))
        (t 0)))
```

Figure 25: Definition of `acl2-count`



(a) CCM function for `f`.

(b) CCM functions for `foo`

Figure 26: Example CCMs

of `x` that satisfy the governors of the context. The resulting CCM functions are shown in Figure 26(a). For the function `foo` in Figure 24 on page 85, we use different CCMs. Namely, we apply `dec` to the arguments; note that `dec` always returns a natural number, which is a well-founded domain under the $<$ relation. The result is shown in Figure 26(b).

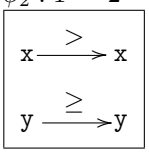
We use CCM functions to show that certain infinite paths are not feasible and also to show that CCGs correspond to terminating functions.

Definition 8.4.4. Given a CCG, $\mathcal{G} = (C, E)$ and a well-founded structure, a set of CCM functions for $\mathcal{G}$, $\{\phi^c_{c'} \mid \langle c, c' \rangle \in E\}$, with annotation m and ordering $\prec$ is *well-founded* if, for all infinite paths $c_1, c_2, \ldots$, in $\mathcal{G}$, there exists $i_0 \geq 1$ and infinite sequence $s_{i_0}, s_{i_0+1}, \ldots$ such that, for all $i \geq i_0$, $s_i \in m(c_i) \quad \wedge \quad \phi^{c_i}_{c_{i+1}}(s_i, s_{i+1}) \neq \times$, and for infinitely many $i \geq i_0$, $\phi^{c_i}_{c_{i+1}}(s_i, s_{i+1}) =>$.

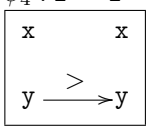
In other words, a set of CCMFs is infinitely decreasing if they reveal that any infinite path through the CCG would cause a value in a well-founded set to never increase and infinitely decrease.

```
(defun ack (x y)
  (if (or (not (integerp x)) (<= x 0))
      1
    (if (or (not (integerp y) (<= y 0)))
        (if (= x 1) 2 (+ x 2))
      (ack (ack (- x 1) y) (- y 1)))))
```

$\phi_1 : 1 \to 1$,
$\phi_2 : 1 \to 2$



$\phi_3 : 2 \to 1$,
$\phi_4 : 2 \to 2$



1. ⟨ack, {(integerp x),(< 0 x),(integerp y),(< 0 y)}, (ack (-x 1) y)⟩
2. ⟨ack, {(integerp x),(< 0 x),(integerp y),(< 0 y)}, (ack (ack (- x 1) y) (- y 1))⟩

Figure 27: Ackermann's function.

Definition 8.4.5. We say that a CCG, $\mathcal{G} = (C, E)$ is *well-founded* if there exists a well-founded structure, an annotation mapping into that structure, and a well-founded set of CCMFs using that annotation and well-founded relation.

It is important to note here that we do not need to fix a CCM for each context in order to satisfy the CCM predicate. Rather, we can select from any of the CCMs for a given context each time it appears in a sequence. For example, consider Ackermann's function, given in Figure 27. Here, if a sequence contains context 2 infinitely often, then y decreases infinitely, and if it does not, then there is an infinite suffix of the sequence that is just context 1, which means that x decreases infinitely often. It is possible to create one measure that decreases in both cases, but this measure requires a well-founded structure more powerful and complex than the natural numbers.

Lemma 8.4.1. *If a CCG, $\mathcal{G} = (C, E)$, is well-founded, then there is no infinite well-formed sequence of the contexts of C.*

Proof. We prove the contrapositive. Let $c = \langle c_i \rangle_{i=1}^{\omega}$ be an infinite well-formed sequence of contexts with witness $\vec{v}$, such that $c_i = \langle f_i \rangle G_i e_i$ and $e_i = f_{i+1}(e_{i,1} \ \ldots \ e_{i,ar(f_{i+1})})$ for all $i \geq 1$. Then c is a path in $\mathcal{G}$.

Suppose that $\mathcal{G}$ were well-founded. Then there exists a well-founded structure, $\langle S, \prec \rangle$, CCM annotation m, and a well-founded set of CCMs, $\{\phi_{c'}^{c} \mid \langle c, c' \rangle \in E\}$. By definition, this means that there exists $i_0 \geq 1$ and a sequence $\langle s_i \rangle_{i=i_0}^{\omega}$, such that, for all $i \geq i_0$, $s_i \in m(c_i)$ and $\phi_{c_{i+1}}^{c_i}(s_i, s_{i+1}) \neq \bot$ and for infinitely many $i \geq i_0$, $\phi_{c_{i+1}}^{c_i}(s_i, s_{i+1}) =>$.

By definition, it is the case that for all $i \geq i_0$, $\mathcal{H}^h \llbracket G_i \rrbracket \epsilon_i^{\vec{v}}$, and $\llbracket e_{i,j} \rrbracket^h \epsilon_i^{\vec{v}} \neq \bot$ for all $1 \leq j \leq ar(f_{i+1})$. By Corollary 8.2.2 on page 80, $\llbracket g_{i+1} \rrbracket^h \epsilon_{i+1}^{\vec{v}} = \llbracket g_{i+1}\sigma_{e_i} \rrbracket^h \epsilon_i^{\vec{v}}$ for all $i \geq i_0$. Therefore, $\mathcal{H}^h \llbracket G_{i+1}\sigma_{e_i} \rrbracket \epsilon_i^{\vec{v}}$, which means that $\mathcal{H}^h \llbracket G_i \cup G_{i+1}\sigma_{e_i} \rrbracket \epsilon_i^{\vec{v}}$ for all $i \geq i_0$. By the same argument, $\llbracket s_{i+1}\sigma_{e_i} \rrbracket^h \epsilon_i^{\vec{v}} = \llbracket s_{i+1} \rrbracket^h \epsilon_{i+1}^{\vec{v}}$ for all $i \geq i_0$.

By the definition of CCMF and the fact that $\phi_{c_{i+1}}^{c_i}(s_i, s_{i+1})$ is never $\times$ and is $>$ for infinitely many $i \geq i_0$, this means that $\llbracket s_i \rrbracket^h \epsilon_i^{\vec{v}} \succeq \llbracket s_{i+1}\sigma_{e_i} \rrbracket^h \epsilon_i^{\vec{v}} = \llbracket s_{i+1} \rrbracket^h \epsilon_{i+1}^{\vec{v}}$ for all $i \geq i_0$, and $\llbracket s_i \rrbracket^h \epsilon_i^{\vec{v}} \succ \llbracket s_{i+1}\sigma_{e_i} \rrbracket^h \epsilon_i^{\vec{v}} \llbracket s_{i+1} \rrbracket^h \epsilon_{i+1}^{\vec{v}}$ for infinitely many $i \geq i_0$. Therefore, $\langle \llbracket s_i \rrbracket^h \epsilon_i^{\vec{v}} \rangle_{i=i_0}^{\omega}$ is an infinitely decreasing sequence, which contradicts the fact that $\prec$ is well-founded over S. Therefore, $\mathcal{G}$ cannot be well-founded. □

It turns out that we only need to consider maximal SCCs (strongly connected components) to establish termination.

Theorem 8.4.1. *Let $\mathcal{G} = (C, E)$ be a CCG, where C is a complete set of contexts for d. If every maximal SCC of $\mathcal{G}$ is well-founded, then all functions of d terminate on all inputs.*

Proof. By Theorem 8.3.2 on page 87, the functions of d all terminate on all inputs if and only if every well-formed sequence of C is finite. By Lemma 8.3.1 on page 88, every such sequence is a path through $\mathcal{G}$.

Every infinite path through $\mathcal{G}$ ends with an infinite suffix in one maximal SCC $\mathcal{G}$. This is because, by the definition of maximal SCC, if a path leaves an SCC it cannot reach it again (since we could then make a larger SCC). Since there are only a finite number of SCCs, an infinite path must enter one SCC and never leave it.

Therefore, if we show that there can be no infinite well-formed sequences through each SCC, we have proven that there are no infinite well-formed sequences. But since every SCC is well-founded, this is true by Lemma 8.4.1 on the previous page. □

8.5 Context Absorption

Notice that the converse of Theorem 8.4.1 does not hold because the paths of a CCG are a superset of the well-formed sequences of contexts. For example, notice that when we split function `f` from Figure 24 on page 85 into several functions, as is the case for the definitions

```
(defun g (x) (f (+ x 1)))
(defun h (x) (f (- x 1)))
(defun f (x)
  (cond ((or (not (integerp x))
             (= x 0))
         0)
        ((< x 0) (g x))
        (t       (h x))))
```

1. ⟨g, {}, (f (+ x 1))⟩
2. ⟨h, {}, (f (- x 1))⟩
3. ⟨f, {(integerp x), (< x 0)}, g(x)⟩
4. ⟨f, {(integerp x),(<= 0 x)}, (h x)⟩

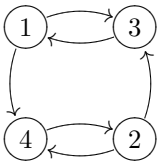
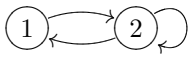


Figure 28: Altered version of function defined in Figure 24 on page 85

```
(defun f (x)
  (cond ((or (not (integerp x))
             (<= x 1))
         1)
        ((= (mod x 2) 1)
         (f (+ x 1)))
        (t
         (+ 1 (f (/ x 2))))))
```



$\phi_1 : 1 \to 2$

size(x)	size(x)

$\phi_2 : 2 \to 1$, $\phi_3 : 2 \to 2$

size(x) $\xrightarrow{>}$ size(x)

1. ⟨f, {(integerp x), (< 1 x), (= (mod x 2) 1)}, (f (+ x 1))⟩
2. ⟨f, {(integerp x), (< 1 x), (not (= (mod x 2) 1))}, (f (/ x 2))⟩

Figure 29: Example of the abstraction inherent in the infinite CCM relation.

given in Figure 28, all the contexts now appear in the same SCC. Why? Consider the function, `g`. Note that `g(2)` results in the call `f(3)`, which leads to context 4. A similar situation arises for `h`. Thus $1, 4, 2, 3, 1, 4, 2, 3, \ldots$ is a valid path through any CCG, even though it is not a well-formed sequence of contexts. Each time through the loop $1, 4, 2, 3$, the value of `x` stays the same, hence, the termination analysis presented so far fails.

Another source of imprecision is due to the local analysis used in determining if a CCG is well-founded. If a value decreases over several steps, but increases for one of those steps, the termination analysis presented so far will fail. Consider the example in Figure 29. When `x` is odd, 1 is added to `x` and when it is even, `x` is divided by 2. This continues until `x` is 1 (or not a positive integer). This results in an overall decrease of the value of `x` despite the initial increase.

In order to gain more accuracy and overcome many of the problems caused by the local nature of our analysis, we introduce the idea of context merging. This essentially enables

us to broaden the scope of our local analysis.

Definition 8.5.1. Let c_1 and c_2 be calling contexts such that $c_1 = \langle f_1, G_1, e_1 \rangle$ and $c_2 = \langle f_2, G_2, e_2 \rangle$. The result of *merging* c_1 and c_2, denoted $c_1; c_2$, is the set $\{\langle f_1, G, e_2\sigma_{e_1} \rangle \mid \langle \forall \epsilon \in Env :: \mathcal{H}^h [\![G]\!] \epsilon \equiv \mathcal{H}^h [\![G_1 \cup G_2\sigma_{e_1}]\!] \epsilon \rangle\}$.

Again, the conditions of any context in the merging can be thought of as being $G_1 \cup G_2\sigma_{e_1}$. The weaker definition is to allow for simplification when appropriate. As an example, note that contexts 1 and 2 of Figure 24 on page 85 are members of the results of merging contexts 3 and 1 and contexts 4 and 2 from Figure 28 on the preceding page, respectively. This makes sense as the example in Figure 28 on the previous page was obtained by splitting `f` into several functions and merging essentially recombines the contexts.

We now use merging to define the notion of absorption and show that given a CCG, we can define an infinite sequence of CCGs such that if we can prove that at least one CCG in the sequence terminates, then so does the original CCG. This can greatly extend the applicability of our analysis.

Definition 8.5.2. Given a CCG, $\mathcal{G} = (C, E)$, and $c \in C$ the result of *absorbing* c *into* $\mathcal{G}$ is a CCG $\mathcal{G}' = (C', E')$ where $C' = C - \{c\} \cup \{cc \in c; c' \mid \langle c, c' \rangle \in E\}$, and $\langle \forall \langle c_1, c_2 \rangle \in E : c_1 \neq c : \langle c_1, c_2 \rangle \in E' \rangle$

Lemma 8.5.1. *Let $\mathcal{G} = (C, E)$ be a CCG such that if there are no infinite well-formed sequences of the contexts of C then the functions in d are terminating for all inputs. Let $c \in C$, and $\mathcal{G}' = (C', E')$ be the result of absorbing c into $\mathcal{G}$. Then there exists an infinite well-formed sequence of the contexts of C if and only if there exists an infinite well-formed sequence of the contexts of C'.*

Proof. We begin by defining the following two functions. The first is $\psi : C'^{\omega} \times \omega \to C^{\omega}$:

$$\psi(\langle c'_i \rangle_{j \geq 1}, i) = \begin{cases} \langle\rangle & \text{if } i = 0 \\ \psi(\langle c'_i \rangle_{j \geq 1}, i-1) \mathbin{@} \langle c, c' \rangle & \text{if } c'_i \in c; c' \text{ for some } c' \\ \psi(\langle c'_i \rangle_{j \geq 1}, i-1) \mathbin{@} \langle c'_i \rangle & \text{otherwise} \end{cases}$$

where @ appends two sequences. Let $\Psi : C'^{\omega} \to C^{\omega}$ be defined by $\Psi(cs') = \lim_{i \to \omega} \psi(cs', i)$. Then Ψ is a bijection, due to the fact that c cannot appear in any sequence in C'^{ω}. Now we define $\phi : C'^{\omega} \times \omega \to \omega$:

$$\phi(\langle c'_j \rangle_{j \geq 1}, i) = \begin{cases} i & \text{if } i \in \{0, 1\} \\ \phi(\langle c'_j \rangle_{j \geq 1}, i-1) + 2 & \text{if } c'_{i-1} \in c; c' \text{ for some } c' \\ \phi(\langle c'_j \rangle_{j \geq 1}, i-1) + 1 & \text{otherwise} \end{cases}$$

Suppose that $cs' = \langle c'_j \rangle_{j \geq 1} \in C'^{\omega}$ and $cs = \langle c_j \rangle_{j \geq 1} = \Psi(cs')$. Then note that, for all $i \geq 1$ such that $c'_i \in c; c'$ for some c', $c_{\phi(cs',i)} = c$ and $c_{\phi(cs',i)+1} = c'$. For all other $i \geq 1$, $c'_i = c_{\phi(cs',i)}$. Now we prove that cs is well-formed if and only if cs' is well-formed.

($\Rightarrow$): Let $cs = \langle c_i \rangle_{i \geq 1} \in C^{\omega}$ be well-formed with witness $\vec{v}$, where, for all $i \geq 1$, $c_i = \langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})}) \rangle$. Next, let $cs' = \langle c'_i \rangle_{i \geq 1} = \Psi^{-1}(cs)$, where, for all $i \geq 1$, $c'_i = \langle f'_i, G'_i, (f'_{i+1}\ e'_{i,1}\ \ldots\ e'_{i,ar(f'_{i+1})}) \rangle$. Then we prove this direction of the theorem by proving the following stronger statement:

$$\langle \forall k \in \omega :: \langle c'_i \rangle_{1 \leq i \leq k} \text{ is well-formed with witness } \vec{v} \text{ and } \epsilon^{\vec{v}}_{cs,\phi(cs',k+1)} = \epsilon^{\vec{v}}_{cs',k+1} \rangle$$

We prove this by induction.

Base Case: $i = 0$. The first 0 elements of cs' are vacuously well-formed. By our definitions of cs and cs', $f_1 = f'_1$, so by the definition of witness, we have $\epsilon^{\vec{v}}_{cs,1} = [x_i^{f_1} \mapsto v_{1,i}]_{i=1}^{ar(f_1)} = [x_i^{f'_1} \mapsto v'_{1,i}]_{i=1}^{ar(f'_1)} = \epsilon^{\vec{v}}_{cs',1}$.

Induction Step: Suppose our statement is true for $k-1$. We prove that it is then true for k. Note that by the induction hypothesis, $\langle c'_i \rangle_{i=1}^{k-1}$ is well-formed, so we only need to prove that the conditions for well-formedness are satisfied for c'_k to show that $\langle c'_i \rangle_{i=1}^{k}$ is well-formed. Consider the following two cases.

For the first case, suppose that $c'_k \notin c; c'$ for any $c' \in C$. Then as we noted earlier, $c'_k = c_{\phi(cs',k)}$. Therefore, since $\epsilon^{\vec{v}}_{cs',k} = \epsilon^{\vec{v}}_{cs,\phi(cs',k)}$ by the induction hypothesis, and the fact that cs is well-formed, it is clearly the case that c'_k fulfills the requirements of well-formedness, and $\epsilon^{\vec{v}}_{cs',k+1} = \epsilon^{\vec{v}}_{cs,\phi(cs',k+1)}$.

For the second case, suppose that $c'_k \in c; c'$ for some $c' \in C$. Then as we noted earlier, $c_{\phi(cs',k)} = c$ and $c_{\phi(cs',k)+1} = c'$. Therefore, by the definition of merging, $\mathcal{H}^h [\![G'_k]\!] \epsilon^{\vec{v}}_{cs,\phi(cs',k)}$ if

and only if $\mathcal{H}^h \left[\!\left[G_{\phi(cs',k)} \cup G_{\phi(cs',k)+1}\sigma_{e_{\phi(cs',k)}} \right]\!\right] \epsilon^{\vec{v}}_{cs,\phi(cs',k)}$, which, by the definition of holding, is equivalent to $\mathcal{H}^h \left[\!\left[G_{\phi(cs',k)} \right]\!\right] \epsilon^{\vec{v}}_{cs,\phi(cs',k)} \quad \wedge \quad \mathcal{H}^h \left[\!\left[G_{\phi(cs',k)+1}\sigma_{e_{\phi(cs',k)}} \right]\!\right] \epsilon^{\vec{v}}_{cs,\phi(cs',k)}$. For all $e \in G_{\phi(cs',k)}$, we have

$$\begin{array}{rcl} & & [\![e]\!]^h \, \epsilon^{\vec{v}}_{cs',k} \\ \{\, \textit{Induction Hypothesis} \,\} & = & [\![e]\!]^h \, \epsilon^{\vec{v}}_{cs,\phi(cs',k)} \\ \{\, \textit{cs well-formed} \,\} & \notin & \{\bot, \texttt{nil}\} \end{array}$$

For all $e \in G_{\phi(cs',k)+1}$, we have

$$\begin{array}{rcl} & & \left[\!\left[e\sigma_{e_{\phi(cs',k)}} \right]\!\right]^h \epsilon^{\vec{v}}_{cs',k} \\ \{\, \textit{Induction Hypothesis} \,\} & = & \left[\!\left[e\sigma_{e_{\phi(cs',k)}} \right]\!\right]^h \epsilon^{\vec{v}}_{cs,\phi(cs',k)} \\ \{\, \textit{Cor. 8.2.2, cs is well-formed} \,\} & = & [\![e]\!]^h \, \epsilon^{\vec{v}}_{cs,\phi(cs',k)+1} \\ \{\, \textit{cs well-formed} \,\} & \notin & \{\bot, \texttt{nil}\} \end{array}$$

Therefore, $\mathcal{H}^h \, [\![G'_k]\!] \, \epsilon^{\vec{v}}_{cs',k}$.

Next consider $e'_{k,i}$, for any $1 \leq i \leq ar(f_k)$. We have that

$$\begin{array}{rcl} & & \left[\!\left[e'_{k,i} \right]\!\right]^h \epsilon^{\vec{v}}_{cs',k} \\ \{\, \textit{Def. of merging} \,\} & = & \left[\!\left[e'_{\phi(cs',k)+1}\sigma_{e_{\phi(cs',k)}} \right]\!\right]^h \epsilon^{\vec{v}}_{cs',k} \\ \{\, \textit{Induction Hypothesis} \,\} & = & \left[\!\left[e'_{\phi(cs',k)+1}\sigma_{e_{\phi(cs',k)}} \right]\!\right]^h \epsilon^{\vec{v}}_{cs,\phi(cs',k)} \\ \{\, \textit{Cor. 8.2.2, cs well-formed} \,\} & = & \left[\!\left[e'_{\phi(cs',k)+1} \right]\!\right]^h \epsilon^{\vec{v}}_{cs,\phi(cs',k)+1} \end{array}$$

From this, we know that $\left[\!\left[e'_{k,i} \right]\!\right]^h \epsilon^{\vec{v}}_{cs',k} \neq \bot$, since by the definition of witness, we know that $\left[\!\left[e'_{\phi(cs',k)+1} \right]\!\right]^h \epsilon^{\vec{v}}_{cs,\phi(cs',k)+1} \neq \bot$. We also know from this that $\epsilon^{\vec{v}}_{cs',k+1} = \epsilon^{\vec{v}}_{cs,\phi(cs',k+1)}$ by the definitions of ϕ and witnesses, as well as the fact that $f'_{k+1} = f_{\phi(cs',k+1)}$. This relieves all the proof obligations for this case, proving the induction, and therefore this direction of the proof.

($\Leftarrow$): Suppose that $cs' = \langle c'_i \rangle_{i \geq 1}$ is well-formed with witness $\vec{v}$, where for all $i \geq 1$, $c'_i = \langle f'_i, G'_i, (f'_{i+1} \;\; e'_{i,1} \;\; \ldots \;\; e'_{i,ar(f'_{i+1})}) \rangle$. Now suppose that there is no well-formed sequence of the contexts of C. We refer to this as the Contradiction Hypothesis. We will use it to draw a contradiction, thereby proving that there is such a sequence.

Note that by the hypotheses of this theorem, all the functions of d must terminate on all inputs. Therefore, by Thm. 8.2.1, every expression is terminating.

Let $cs = \langle c_i \rangle_{i \geq 1} \in C^\omega = \Psi(cs')$, where $c_i = \langle f_i, G_i, (f_{i+1}\ e_{i,1}\ \ldots\ e_{i,ar(f_{i+1})}) \rangle$ for all $i \geq 1$. We prove the following result that contradicts our assumption that there is no well-founded sequence of contexts of C:

$$\langle \forall k \in \omega :: \langle c_i \rangle_{1 \leq i < \phi(cs',k+1)} \text{ is well-formed with witness } \vec{v}, \text{ and } \epsilon^{\vec{v}}_{cs,\phi(cs',k+1)} = \epsilon^{\vec{v}}_{cs',k+1} \rangle$$

We prove this by induction on k.

Base Case: $i = 0$. Then the sequence containing the first 0 elements of cs is vacuously well-formed, and $f'_1 = f_1$, so $\epsilon^{\vec{v}}_{cs,1} = [x_i^{f_1} \mapsto v_{1,i}]_{i=1}^{ar(f_1)} = [x_i^{f'_1} \mapsto v'_{1,i}]_{i=1}^{ar(f'_1)} = \epsilon^{\vec{v}}_{cs',1}$.

Induction Step Suppose our statement is true for $k-1$. We prove that it is then true for k. Note that by the induction hypothesis, $\langle c_i \rangle_{i=1}^{\phi(cs',k-1)}$ is well-formed. Consider the following two cases.

For the first case, suppose that $c'_k \notin c; c'$ for any $c' \in C$. Then as we noted earlier, $c'_k = c_{\phi(cs',k)}$. Also, $\phi(cs', k+1) = \phi(cs', k) + 1$. Therefore, since $\langle c_i \rangle_{i=1}^{\phi(cs',k)-1}$ is well-formed, we only need to prove that $c_\phi(cs', k)$ satisfies the requirements of well-foundedness in order to prove that $\langle c_i \rangle_{i=1}^{\phi(cs',k+1)-1}$ is well-founded. Since $\epsilon^{\vec{v}}_{cs',k} = \epsilon^{\vec{v}}_{cs,\phi(cs',k)}$ by the induction hypothesis, and the fact that cs' is well-formed, it is clearly the case that $c_{\phi(cs',k)}$ fulfills the requirements of well-formedness, and $\epsilon^{\vec{v}}_{cs',k+1} = \epsilon^{\vec{v}}_{cs,\phi(cs',k+1)}$.

For the second case, suppose that $c'_k \in c; c'$ for some $c' \in C$. Then as we noted earlier, $c_{\phi(cs',k)} = c$ and $c_{\phi(cs',k)+1} = c'$. Also, $\phi(cs', k+1) = \phi(cs', k) + 2$. Therefore, we have to prove that both $c_{\phi(cs',k)}$ and $c_{\phi(cs',k)+1}$ fulfill the requirements of well-formedness. By the Contradiction Hypothesis and Thm. 8.2.1, we know that $[\![e_{\phi(cs',k),i}]\!]^h\, \epsilon^{\vec{v}}_{cs,\phi(cs',k),i} \neq \bot$ for any $1 \leq i \leq ar(f_{\phi(cs',k)})$. Also, for any $e \in G_{\phi(cs',k)}$, we know that $e \in G'_k$, by the definition of merging. Therefore,

$$\begin{array}{rcl} & & [\![e]\!]^h\, \epsilon^{\vec{v}}_{cs,\phi(cs',k),i} \\ \{\, \textit{Induction Hypothesis} \,\} & = & [\![e]\!]^h\, \epsilon^{\vec{v}}_{cs',k} \\ \{\, cs' \textit{ is well-founded} \,\} & \notin & \{\bot, \texttt{nil}\} \end{array}$$

Likewise, for every $e \in G_{\phi(cs',k)+1}$, $e\sigma_{e_{\phi(cs',k),i}} \in G'_k$ by the definition of merging. Therefore,

$$
\begin{array}{rcl}
 & & [\![e]\!]^h\, \epsilon^{\vec{v}}_{cs,\phi(cs',k)+1} \\
\{\, \textit{Cor. 8.2.2} \,\} & = & \left[\!\!\left[e\sigma_{e_{\phi(cs',k),i}} \right]\!\!\right]^h \epsilon^{\vec{v}}_{cs,\phi(cs',k)} \\
\{\, \textit{Induction Hypothesis} \,\} & = & \left[\!\!\left[e\sigma_{e_{\phi(cs',k),i}} \right]\!\!\right]^h \epsilon^{\vec{v}}_{cs',k} \\
\{\, \textit{Def. witness} \,\} & \notin & \{\bot, \texttt{nil}\}
\end{array}
$$

Finally, for every $1 \leq i \leq ar(f_{\phi(cs',k)+1})$,

$$
\begin{array}{rcl}
 & & [\![e_{\phi(cs',k)+1,i}]\!]^h\, \epsilon^{\vec{v}}_{cs,\phi(cs',k)+1} \\
\{\, \textit{Cor. 8.2.2} \,\} & = & \left[\!\!\left[e_{\phi(cs',k)+1,i}\sigma_{e_{\phi(cs',k),i}} \right]\!\!\right]^h \epsilon^{\vec{v}}_{cs,\phi(cs',k)} \\
\{\, \textit{Induction Hypothesis} \,\} & = & \left[\!\!\left[e_{\phi(cs',k)+1,i}\sigma_{e_{\phi(cs',k),i}} \right]\!\!\right]^h \epsilon^{\vec{v}}_{cs',k} \\
\{\, \textit{Def. merging} \,\} & = & \left[\!\!\left[e'_{k,i}\sigma_{e_{\phi(cs',k),i}} \right]\!\!\right]^h \epsilon^{\vec{v}}_{cs',k}
\end{array}
$$

From this, we know that $\left[\!\!\left[e'_{\phi(cs',k)+1} \right]\!\!\right]^h \epsilon^{\vec{v}}_{cs,\phi(cs',k)+1} \neq \bot$, since by the definition of witness, $\left[\!\!\left[e'_{k,i} \right]\!\!\right]^h \epsilon^{\vec{v}}_{cs',k} \neq \bot$. We also know that $\epsilon^{\vec{v}}_{cs',k+1} = \epsilon^{\vec{v}}_{cs,\phi(cs',k+1)}$ by the definitions of ϕ and witness, as well as the fact that $f'_{k+1} = f_{\phi(cs',k+1)}$. This relieves all the proof obligations for this case, proving the induction.

Therefore, *cs* is a well-formed sequence of the contexts of C, contradicting our hypothesis that there is no such sequence. Therefore, the theorem is proved. □

Corollary 8.5.1. *Let $C_0, C_1, C_2, \ldots$ be a sequence of sets of calling contexts such that C_0 is a semi-complete set of contexts for d, and for all $i \geq 0$, C_{i+1} is obtained from C_i by absorbing a context in C_i. Then for all $i \geq 0$, an infinite well-formed sequence of contexts in C_i exists only if an infinite well-formed sequence of contexts in C_0 exists.*

Proof. The proof is a simple inductive argument on i, using Lemma 8.5.1 on page 94. □

Note, however, that the converse of this corollary is not true. That is, absorption can result in more precise analysis, *e.g.*, the code in Figure 28 on page 93. Therefore, by analyzing a set of contexts resulting from a sequence of absorption, we are more likely to prove termination.

Theorem 8.5.1. *Let $\mathcal{G}_0, \mathcal{G}_1, \ldots$ be a sequence of CCGs such that $\mathcal{G}_0$ is a semi-complete CCG of d, and $\mathcal{G}_{i+1}$ is obtained from $\mathcal{G}_i$ by absorbing a context. If for some i, every maximal SCC of $\mathcal{G}_i$ is well-founded, then every function in d terminates on all inputs.*

Proof. By Corollary 8.5.1 on the preceding page, there is an infinite well-founded sequence of contexts in C_i if and only if there is an infinite sequence of precise contexts. The theorem then directly follows from Theorem 8.4.1 on page 92. □

8.6 Bibliographic Notes

Due to the overlap in bibliographic data between this and then next two chapters, we postpone discussion to Section 10.2.

8.7 Summary

We introduced the notion of calling contexts and Calling Context Graphs (CCGs) for representing the recursive behavior of functions. We proved via the concept of well-formed sequences of contexts that calling contexts are in some sense complete with regards to termination, and that CCGs are a conservative approximation with regards to termination.

We also introduced Calling Context Measures (CCMs), and showed how to annotate a CCG with them in order to prove termination. This is done by building CCM functions that accurately compare the values of CCMs across single transitions of the CCG, and then used transitivity to show that some CCMs decrease infinitely through every infinite path of the CCG.

Finally, we gave an improvement to the basic CCG-based termination algorithm in the form of absorption, which allows us to combine adjacent contexts and thereby consider the behavior of the CCG and CCMs over multiple transitions at once. This is similar to composing the transition relation, except that absorption allows us to compose a small portion of the transition relation instead of composing it in its entirety.

CHAPTER IX

IMPLEMENTATION

In this chapter, we provide details on our implementation of the CCG-based termination analysis algorithm. We begin in Section 9.1 by giving the straightforward algorithms suggested by the definitions given in Chapter 8. We continue in Section 9.2, by giving a hierarchical analysis, which improves efficiency and provides heuristics for choosing CCMs and employing absorption. This is the algorithm used in our experimental evaluation (presented in Chapter 10).

9.1 General Algorithm

The termination theory presented in Chapter 8 and culminating in Theorem 8.5.1 on page 98 is undecidable in general. In fact, just determining if an edge belongs in a context graph is an undecidable problem. What we present here is an algorithm using a theorem prover that is a conservative approximation of the method described in the last two sections. We begin with a set of helper functions and macros as defined in Figure 30 on the next page. Note that $V(G)$ and $E(G)$ are the vertices and edge of graph G, respectively.

9.1.1 Building the Contexts and Context Graph

The algorithm for obtaining a (semi-)complete set of contexts from a set of function definitions is given in Figure 31. The **cs** function takes the names of the functions being defined (F), the name of the function whose contexts are being computed (f), the governors at the current position in the body of f (C), the expression at the current position (e), and the substitutions suggested by the `let` expressions that are parents of e in e^f.

If e is an `if` expression, we analyze each of its sub-expressions with the appropriate governor added to the set of governors, and union the results together. If it is a `let` expression, we recursively analyze the bound expressions and union them together with the contexts for the body. Note that in the recursive analysis of the body, we update the

$\mathbf{andset}(\{s_1, \ldots, s_n\}) = \texttt{(and } s_1 \ \ldots \ s_n\texttt{)}$
$\mathbf{callee}(\texttt{(}f \ e_1 \ e_2 \ \ldots \ e_n\texttt{)}) = f$
$\mathbf{succ}(c, G) = \{c' \mid \langle c, c' \rangle \in E(G)\}$
$\mathbf{pred}(c, G) = \{c' \mid \langle c', c \rangle \in E(G)\}$
$\mathbf{fn}(\langle f, G, e \rangle) = f$
$\mathbf{conds}(\langle f, G, e \rangle) = G$
$\mathbf{call}(\langle f, G, e \rangle) = e$

Figure 30: Helper functions and macros.

$\mathbf{cs}(F, f, C, e, \sigma)$
1: **if** $e = \texttt{(if } e_1 \ e_2 \ e_3\texttt{)}$ **then**
2: $\quad X := \begin{pmatrix} \mathbf{cs}(F, f, C, e_1, \sigma) \cup \\ \mathbf{cs}(F, f, \{e_1\sigma\} \cup C, e_2, \sigma) \cup \\ \mathbf{cs}(F, f, \texttt{(not } e_1\texttt{)}\sigma \cup C, e_3, \sigma) \end{pmatrix}$
3: **else if** $e = \texttt{(let ((}x_1 \ e_1\texttt{) } \ldots \texttt{ (}x_n \ e_n\texttt{)) } e'\texttt{)}$ **then**
4: $\quad X := \begin{pmatrix} \bigcup_{i=1}^{n} \mathbf{cs}(F, f, C, e_i, \sigma) \cup \\ \mathbf{cs}(F, f, C, e', [x_i \mapsto e_i\sigma]_{i=1}^{n}) \end{pmatrix}$
5: **else if** $e = \texttt{(}g \ e_1 \ \ldots \ e_n\texttt{)}$ **then**
6: $\quad X \leftarrow \bigcup_{i=1}^{n} \mathbf{cs}(F, f, C, e_i, \sigma)$
7: $\quad$ **if** $g \in F$ **then**
8: $\quad\quad X := X \cup \{\langle f, C, e\sigma \rangle\}$
9: $\quad$ **end if**
10: **else**
11: $\quad X := \emptyset$
12: **end if**
13: **return** X

$\mathbf{contexts}(d)$
1: $F := \{f \mid f \text{ defined in } d\}$.
2: **return** $\bigcup_{f \in F} \mathbf{cs}(F, f, \emptyset, e^f, [])$

Figure 31: Algorithm for building contexts

substitution as suggested by Definition 2.1.3 on page 9. If e is a function call, we recursively gather the contexts of the arguments. If e is a call to one of the functions in F, we add the precise context for e. The only other cases in our language are when e is a variable or a value, in which case there are no contexts for e.

The **contexts** function calls the **cs** with the appropriate initial values for each function defined in a set of definitions, d.

The algorithm for constructing a CCG out of a set of contexts is given in Figure 32 on the next page. This is the first place where theorem proving is used. Given an intermediate history, h, let $\mathbf{prove}(e)$ take an expression, e, and return whether the theorem prover can prove that, for all possible environments, ϵ, $[\![e]\!]^h \, \epsilon \notin \{\texttt{nil}, \bot\}$. When building the graph, we need to be sure that we do not omit an edge that should be there. If we do, we cannot be sure that every well-formed sequence of contexts is a path through the graph. We

edge?(c_1, c_2)
1: $e :=$**call**(c_1)
2: **return** $\neg$**prove** $\left(\begin{array}{l} \texttt{(not (and } \mathbf{andset}(\mathbf{conds}(c_1)) \\ \qquad\qquad \mathbf{andset}(\mathbf{conds}(c_2))\sigma_e\texttt{)))} \end{array} \right)$

CCG(C)
1: $E \leftarrow \emptyset$
2: **for all** $c_1 \in C$ **do**
3: **for all** $c_2 \in C$ **do**
4: **if** $(\mathbf{callee}(\mathbf{fn}(c_1)) = \mathbf{fn}(c_2) \quad \wedge \quad \mathbf{edge}?(c_1, c_2))$ **then**
5: $E := E \cup \{\langle c_1, c_2 \rangle\}$
6: **end if**
7: **end for**
8: **end for**
9: **return SCCs**$((C, E))$

Figure 32: Algorithm for building a context graph.

therefore attempt to prove for each pair of contexts that we do not need to add an edge between them. If this proof fails, we add the edge. In other words, given two contexts, $\langle f_1, G_1, e_1 \rangle$ and $\langle f_2, G_2, e_2 \rangle$, such that e_1 is a call to f_2, we want to prove that, for all ϵ, $\neg(\mathcal{H}^h \, [\![G_1]\!] \, \epsilon) \quad \vee \quad \neg(\mathcal{H}^h \, [\![G_2\sigma_{e_1}]\!] \, \epsilon)$. By the semantics of our language, this is equivalent to proving that $[\![\texttt{(not (and } \mathbf{andset}(G_1) \ \mathbf{andset}(G_2)\sigma_{e_1}\texttt{))}]\!]^h \, \epsilon$ is not `nil` or $\perp$.

The algorithm, then, is simply to attempt to prove this expression for every possible edge, and add an edge if the theorem fails.

9.1.2 Absorption

The algorithms to perform merging and absorption are given in Figure 33 on the following page. The merging algorithm is a straightforward implementation of Definition 8.5.1 on page 94.

For the absorption algorithm, we begin by creating V', which will contain the updated vertex set, and C', which will contain pairs of the form $\langle c', c; c' \rangle$. We set V' to the vertex set of $\mathcal{G}$ without c, and initialize C' to the empty set. We then create $c; c'$ for each successor, c', of c, and add the appropriate values to V' and C'. At the end of the loop, V' contains the new set of contexts.

The rest of the algorithm computes the edges for the new CCG. We want to avoid having to reconstruct the entire CCG from scratch. To do this, we use the fact that the merged

```
merge(c, c')
 1: e :=call(c)
 2: return ⟨fn(c), conds(c) ∪ {pσ_e | p ∈conds(c')}, call(c')σ_e⟩

absorb(c, G)
 1: V' := V(G) − {c}
 2: C' := {}
 3: for all c' ∈ succ(c, G) do
 4:    cc := merge(c, c')
 5:    C' := C' ∪ {⟨c', cc⟩}
 6:    V' := V' ∪ {cc}
 7: end for
 8: E' := E(G) − ({⟨c, s⟩ ∈ E(G)} ∪ {⟨p, c⟩ ∈ E(G)})
 9: for all ⟨c', cc⟩ ∈ C' do
10:    for all s ∈ succ(c', G) do
11:       if s = c then
12:          for all ⟨c'', cc'⟩ ∈ C' do
13:             if edge?(cc, cc') then
14:                E' := E' ∪ {⟨cc, cc'⟩}
15:             end if
16:          end for
17:       else if edge?(cc, s) then
18:          E' := E' ∪ {⟨cc, s⟩}
19:       end if
20:    end for
21:    for all p ∈ pred(c, G) − {c'} do
22:       if edge?(p, cc) then
23:          E' := E' ∪ {⟨p, cc⟩}
24:       end if
25:    end for
26: end for
27: return SCCs((V', E'))
```

Figure 33: Algorithm for compaction.

context's conditions are stronger than those of either of the contexts that were merged to make it. Therefore, a context will be a predecessor of $c; c'$ only if it is a predecessor of c, and will be a successor of $c; c'$ only if it is a successor of c'. We therefore start with the original set of edges from the CCG with the edges involving c removed. For each pair, $\langle c', c; c' \rangle$, in C', we cycle through the successors of c' and predecessors of c, removing any edges that can be safely removed. Note that we skip c' when we examine the predecessors of c. This is because the edge $\langle c', c \rangle$ would have already been examined when we processed the successors of c'.

9.1.3 CCM Function Construction

CCMF$(c_1, c_2, S_1, S_2, \prec)$

1: Let $\phi^{c_1}_{c_2} : S_1 \times S_2 \to \{>, \geq, \times\}$
2: $e_1 := \textbf{call}(c_1)$
3: $hyps := \textbf{conds}(c_1) \cup \{p\sigma_{e_1} \mid p \in \textbf{conds}(c_2)\}$
4: **for all** $s_1 \in S_1$ **do**
5: **for all** $s_2 \in S_2$ **do**
6: **if prove**((implies **andset**($hyps$) ($\prec$ $s_2\sigma_{e_1}$ s_1))) **then**
7: $\phi^{c_1}_{c_2}(s_1, s_2) := >$
8: **else if prove**((implies **andset**($hyps$) ($\preceq$ $s_2\sigma_{e_1}$ s_1))) **then**
9: $\phi^{c_1}_{c_2}(s_1, s_2) := \geq$
10: **else**
11: $\phi^{c_1}_{c_2}(s_1, s_2) := \times$
12: **end if**
13: **end for**
14: **end for**
15: **return** ϕ_{c_1,c_2}

CCMFs$(\mathcal{G}, m, \prec)$

1: $S := \emptyset$
2: **for all** $c_1 \in V(\mathcal{G})$ **do**
3: **for all** $c_2 \in V(\mathcal{G})$ s.t. $\langle c_1, c_2\rangle \in E(\mathcal{G})$ **do**
4: $S := S \cup \textbf{CCMF}(c_1, c_2, m(c_1), m(c_2), \prec)$
5: **end for**
6: **end for**
7: **return** S

Figure 34: Algorithm for constructing CCM functions

The algorithm for constructing CCM functions is given in Figure 34. It takes a CCG, $\mathcal{G}$, a function mapping contexts to sets of CCMs, m, and an ordering, $\prec$, and returns the set of CCM functions for the contexts in G. We use the theorem prover again in this step in order to determine what value to map a given pair of value expressions to. If the prover cannot determine that the value should be $>$, it attempts to prove that it should be $\geq$. If it cannot prove that either of these, it sets the value to $\times$. The resulting CCM function therefore returns $>$ only if the actual value function returns $>$. Likewise, it only returns $\geq$ if the actual CCM function returns $>$ or $\geq$. Thus, when we construct the infinite CCM relation, we will not find an infinite sequence of CCMs that is infinitely decreasing if there is no such sequence. The rest of the algorithm is straight-forward. A CCM function is constructed for each pair of contexts that are adjacent in the approximation of the context graph. All the functions are accumulated in S and returned.

9.1.4 Well Foundedness

The final component of the analysis is demonstrating that a set of CCMFs for a given CCG are well-founded (See Definition 8.4.4 on page 90). This problem is equivalent to the complete Size-Change Termination (SCT) property (see Section 11.3 for a more detailed explanation). There are two well-known algorithms for deciding SCT [65], either of which may be used to decide if a set of CCMFs are well-founded.

9.2 The Hierarchical Algorithm

In this section, we provide a full algorithm for implementing the CCG analysis. It is on this algorithm that our ACL2 implementation is based. In addition to discussing how we choose CCMs and employ absorption, we develop a hierarchy of termination analyses that applies lightweight analyses first, and resorts to slower but more powerful analyses only when the faster ones fail. This hierarchy is designed to maximize reuse of information from one stage in the hierarchy to the next. That is, each analysis has the potential to discover new information that is relevant to the termination proof even if it fails to completely prove termination. Rather than dispose of this information, our hierarchical analysis reuses and adds to that information at each stage. The result, as we will see in the empirical evaluation presented in Chapter 10, is an analysis that more efficient than a straightforward implementation of the CCG analysis, but just as powerful.

9.2.1 Choosing CCMs

We describe the heuristics we use for annotating calling contexts with measures. In most termination analyses, this is the most critical and difficult step in the analysis. One measure must be found that decreases with each step of the program. One of the strengths of the CCG analysis is that this is no longer necessary. Instead, we can use simple heuristics to choose candidate measures. Then, the rest of the analysis, which can be completely automated with the help of a theorem prover, determines if some combination of these measures can be used to construct a termination proof. Through experimentation and analysis of failed CCG proof attempts, we have determined that the following heuristics

```
(defun upto (i max)
  (if (and (integerp i)
           (integerp max)
           (<= i max))
      (+ 1 (upto (+ 1 i) max))
    0))
```

Figure 35: The `upto` function.

work well in practice for ACL2.

9.2.1.1 Formal Sizes

For each function formal, x_i^f, of the function containing a context, c, `(acl2-count` x_i^f`)` is included in the CCMs of c (see Figure 25 on page 90 for the definition of `acl2-count`). For the majority of functions, these CCMs suffice for proving termination. Many simple functions simply walk down a list formal until it ends, or decrease the absolute value of an integer formal until it is 0. Using these CCMs, such functions are easily proven terminating. Even for more complex recursive functions, such CCMs are often sufficient. For example, the `ack` function from Figure 27 on page 91 is easily proven terminating by our analysis using the CCMS `(acl2-count x)` and `(acl2-count y)`.

9.2.1.2 Shrinking Differences

If the conditions for a calling context, c, contain an expression of the form `(<` e e'`)` or `(not (<` e' e`))`, where e and e' are any expressions, then we add `(acl2-count (+ 1 (-` e' e`)))` to the CCMs of c. This CCM is especially helpful when analyzing functions that mimic the behavior of for-loops. An example of this is the `upto` function, defined in Figure 35. Here, `i` is a counter that is increased until it surpasses `max`. This loop is proven terminating using the CCM `(acl2-count (- max i))`.

9.2.1.3 Natural numbers and Integers

A third heuristic for choosing CCMs is to add `(acl2-count` e`)` to the CCMs of c when `(not (zp` e`))` is a condition of c. Likewise, when `(not (zip` e`))` is a condition of c, we add `(acl2-count` e`)` as a CCM for c. Recall that the `zp` function returns true when its argument is not a positive integer. Likewise, the `zip` function returns true when its

```
(defun upto0 (i max)
  (upto i max))

(defun upto (i max)
  (if (and (integerp i)
           (integerp max)
           (<= i max))
      (+ 1 (upto0 (+ 1 i) max))
    0))
```

Figure 36: A new definition for `upto`.

argument is not a non-zero integer (*i.e.*, if it is zero or not an integer). These two functions are used in functions whose intended domains are either natural numbers or integers. Such functions often decrease some natural number or integer magnitude until it reaches 0. In order to create a total function that behaves in this manner, such functions often default to the base case when a value outside the intended domain is given. Again, the `ack` function from Figure 27 on page 91 is an example of where `zp` would be useful. The two base cases occur when `x` or `y` are not positive integers. Instead of `(or (not (integerp x)) (<= x 0))`, most users would instead simply say `(zp x)`. Since these functions are so popular for use in rulers and governors, it makes sense to use them as a hint that the expression they contain is a valid CCM.

9.2.1.4 Lists

The list equivalents to `zp` are `(atom x)`, `(endp x)`, and `(not (consp x))`. These three are logically synonymous, and return true if `x` is not a non-empty list. Many functions walk through or manipulate lists until they are empty. For such functions, one of the three expressions listed above are applied to the list to determine when a base case is reached. Therefore, if `(not (atom` e`))`, `(not (endp` e`))`, or `(consp` e`))` are expressions in the conditions of c, we add `(acl2-count` e`)` to the CCMs of c.

9.2.1.5 CCM propagation

Consider the alternate definition of the `upto` function given in Figure 36. Here, we have split the definition of `upto` into two functions. The `upto0` function does nothing but call `upto`,

and `upto` calls `upto0` recursively. Now suppose that we used the heuristics listed so far to choose CCMs for the contexts of these functions. Both would include `(acl2-count i)` and `(acl2-count max)`. However, the context for the call to `upto0` in the body of `upto` would also include `(acl2-count (- max i))`, while the other context would not. Now consider the CCG termination analysis using these CCMs. The value of `(acl2-count max)` is non-increasing across both contexts. The value if `(acl2-count i)` is non-increasing across the call to `upto`, and increasing across the other call. Finally, there is not enough information to compare the CCM `(acl2-count (- max i))` with either CCM from the other context. Therefore, the CCG analysis fails.

The problem in this scenario is that we did not realize that `(acl2-count (- max i))` should also be a CCM for `upto0`. When a loop contains more than one context, we need to keep track of the CCMs throughout the loop to see how they change from one iteration to the next.

To accomplish this, we use *CCM propagation*. An algorithm for this is given in Figure 37 on the next page. The idea is to do a backward breadth-first-search through the CCG, propagating the CCM back from the successor by applying the call substitution of the current context to the CCM of the successor. The result is a CCM that is provably equivalent to the next one along that edge. We only do this once for each context to avoid a blow-up in the number of CCMs. In our example, this will result in `(acl2-count (- max i))` being added to the CCMs for the context representing the call to `upto` in the body of `upto0`. Now we know that the value of `(acl2-count (- max i))` is non-increasing across this call, and decreasing across the call to `upto0` in the body of `upto`.

9.2.2 Invoking the Prover

Theorem proving plays a critical role in our implementation of the CCG analysis. We use it to minimize CCGs and boost the accuracy of the CCMFs. In this section, we discuss the issues involved in integrating theorem prover calls with the overall CCG algorithm.

Recall that an edge must exist from context $c = \langle f, R, e\rangle$ to context $c = \langle f', R', e'\rangle$ if e is a call to f' and $\neg\mathcal{H}^h\,[\![R \cup R'\sigma_e]\!]\,\epsilon$ for all $\epsilon \in Env$. For CCMs s and s' for c and c',

Require: $\mathcal{G} = (C, E)$ a CCG, $c \in C$, s a CCM for c
1: {Initialize the visited list}
2: **for all** $c' \in C$ **do**
3: $visited(c') := false$
4: **end for**
5: $visited(c) := true$
6: {Visit the initial context}
7: Let *queue* be an empty queue.
8: $ccm(c) := s$
9: **for all** $\langle c', c\rangle \in E$ **do**
10: **if** $\neg visited(c')$ **then**
11: Add $\langle c', c\rangle$ to *queue*
12: $visited(c') := true$
13: **end if**
14: **end for**
15: {Visit other contexts in backwards breadth-first order}
16: **while** *queue* is not empty **do**
17: Let $\langle c'', c'\rangle$ be the next element of *queue*
18: Let e be the call of c''
19: $ccm(c'') := ccm(c')\sigma_e$
20: **for all** $\langle c''', c''\rangle \in E$ **do**
21: **if** $\neg visited(c''')$ **then**
22: $visited(c''') := true$
23: **end if**
24: **end for**
25: **end while**
26: **return** *ccm*

Figure 37: CCM propagation algorithm.

respectively, $\phi^c_{c'}(s, s') =>$ when $\mathcal{H}^\epsilon [\![R \cup R'\sigma_e]\!] \Rightarrow s'\sigma_e \prec s$. Likewise, $\phi^c_{c'}(s, s') =\geq$ when $\mathcal{H}^\epsilon [\![R \cup R'\sigma_e]\!] \Rightarrow s'\sigma_e \preceq s$. The more accurately we can determine that an edge can be eliminated from a CCG or that a CCM is decreasing or non-increasing, the more effective our analysis will be.

Recall that by definition, $\mathcal{H}^h [\![E]\!]\, \epsilon$ means that for all $e \in E$, $[\![e]\!]^h\, \epsilon \notin \{\bot, \texttt{nil}\}$. Since all the functions in H have been proven terminating, the only way that $[\![e]\!]^h\, \epsilon = \bot$ is if e contains a call to a function of d that is non-terminating. Since we have yet to determine if the functions of d terminate on all inputs, trying to determine whether the rulers are non-terminating leads to circular reasoning. So, we substitute the following weaker prover query for determining if an edge can be eliminated from the CCG: $[\![\texttt{(not (and } r_1 \ \ldots \ r_n \ r'_1\sigma_e \ \ldots \ r'_{n'}\sigma_e\texttt{))}]\!]^h\, \epsilon \neq$

`nil`, where $R = \{r_1, \ldots, r_n\}$ and $R' = \{r'_1, \ldots, r'_n\}$.

We take a similar approach for the CCMF conditions. That is, we set $\phi^c_{c'}(s, s')$ to $>$ when the theorem prover can show that

$$\left[\!\left[\texttt{(implies (and } r_1 \ \ldots \ r_n \ r'_1\sigma_e \ \ldots \ r'_{n'}\sigma_e\texttt{) (}\prec\ s\ s'\texttt{))}\right]\!\right]^h \epsilon \neq \texttt{nil}$$

and similarly for setting $\phi^c_{c'}(s, s')$ to $\geq$. In our implementation, $\prec$ will be `o<`.

In addition to formulating the proofs, interacting with the theorem prover presents its own challenges. These stem from the differences in the interactions with the prover intended by the developers of ACL2 and those required by our analysis. The theorem prover is designed primarily as a tool for verification. That is, given a formula that the user believes to be true, ACL2 by default uses all of the proof techniques at its disposal to verify that the formula is in fact a theorem. For example, when proving termination, the user provides a measure (or one is guessed using static analysis), and the theorem prover attempts to prove that the functions are measure admissible using that particular measure. The prover is therefore not designed to give up in a timely manner or at all if it can continue to make some kind of progress on the suggested query.

Contrast this to our own intended use of the theorem prover. We desire a "yes" or "no" response. Either the query can be proven true in a "reasonable" amount of time, or it can not, in which case we proceed with a conservative analysis. We expect some, and in many cases most, of our queries to be unprovable or even false. For example, when pruning the CCG, we will ask the theorem prover if we can remove each edge, fully expecting that in most cases, we will not.

One possible solution that we explored was the use of a new time-out feature for ACL2. The feature, called `with-prover-time-limit` is given a rational time limit in seconds and an expression that requires theorem proving. If the prover cannot complete the proof within the specified time limit, it fails. This solution is problematic for two reasons. First, choosing a "reasonable" time limit for prover queries depends on the context. Some theories are relatively simple and can prove the necessary queries quickly, while others are complex leading to longer proofs and slower prover performance. Secondly, using time limits could

```
(or (and (length-exceedsp (car id) i) ;;limit the induction
         (endp (cdadr id))
         (= (cddr id) 0)
         '(:computed-hint-replacement t :do-not-induct :otf-flg-override))
     (and (> (cddr id) 20) ;; avoid infinite loops
          '(:computed-hint-replacement t
           :do-not (eliminate-destructors eliminate-irrelevance
                    generalize fertilize)
           :in-theory nil)))
```

Figure 38: The computed hint used to enforce termination.

result in the unfortunate consequence of making termination provable on some computers and unprovable on others. That is, if someone using a new, relatively fast computer proves a function terminating, it could still be the case that the theorem prover would time out on the same problem for a user who is using an older and slower machine.

The other solution, and the one we have ultimately implemented, is to enforce the timely termination of the theorem prover by regulating the proof techniques that it is allowed to use, causing the prover to give up if it cannot prove the given query with a reasonable amount of effort. This is done using a *computed hint*, which is a user-supplied hint to the theorem prover that is triggered by context as the theorem prover works on a proof attempt.

The computed hint used for this purpose is given in Figure 38. It is parameterized by a number, i, which is used to specify the number of inductions allowed. The hint consists of two parts. The first conjunct tells ACL2 to stop performing induction after the ith induction. For example, if i is 0, and ACL2 wants to perform an induction on a subgoal, it will instead give up. If i is 1, and ACL2 is processing an induction subgoal (*i.e.*, the base case or an induction step of an inductive proof), then it will give up if it wants to perform another induction. This is a valuable rule, since ACL2 uses induction as a "last resort". If it cannot prove a goal using any of its other proof techniques it uses induction if possible. This part of the hint will limit that behavior. In practice, we use an i of 0 or 1.

The second conjunction in the computed hint turns off all proof techniques if a given subgoal has been worked on for 20 steps without proving it. This subgoal is necessary because there are other prover techniques other than induction that are not guaranteed to

terminate. Most goals are proven within fewer than 20 steps if they are going to be proven at all. Therefore, 20 is a "reasonable" limit to the prover's efforts.

In addition to this computed hint, there is one simple and fast prover technique that can be used by itself to obtain an incredibly lightweight prover analysis. Users of ACL2 will notice that when proving termination using ACL2's current measure-based termination analysis, the prover will occasionally say "The admission of F is trivial...". This message is given when the termination proof is verified using a technique known as built-in clauses. This is a simple pattern-matching technique that matches against well-known recursive behaviors. For example, one built-in clauses states that `(implies (zp x) (o< (acl2-count (- x 1)) (acl2-count x)))`. Therefore, if any measure goal contains `(zp` e`)` in the hypotheses and `(o< (acl2-count (-` e `1)) (acl2-count` e`))` as the goal, it will immediately recognize the goal as a theorem using built-in clauses. The exclusive use of built-in clauses gives us a lightweight but surprisingly effective termination analysis. For example, the `ack` function from Figure 27 on page 91 can be solved using only this technique, since the CCMF theorems needed to prove termination match the built-in clause just given. That is, recognizing that `x` decreases in the inner recursive call and that `y` decreases on the inner recursive call only requires realizing that the `acl2-count` of `x` (or `y`) is larger than that of `(- x 1)` (or `(- y 1)`) when `(zp x)` (or `(zp y)`) is one of the rulers. The result is an extremely fast termination proof for `ack`, which takes on the order of milliseconds.

9.2.3 Minimizing Prover Time

As we will see in more detail in Chapter 10, the running time for the CCG analysis is dominated by the time taken by the theorem prover. In order to minimize running time while maintaining the accuracy of the analysis, we introduce two heuristics for minimizing the number of calls made to the theorem prover during successful termination proof.

9.2.3.1 Per-Context CCMFs

The first heuristic involves computing CCMFs on a "per-context" basis instead of a "per-edge" basis. In the full implementation of the CCG analysis, CCMs are assigned to each context separately, and the CCMF is created for each edge, using the conditions of the two

adjacent contexts to determine if a $>$ or $\geq$ value can be assigned to a given pair of CCMs. This means the construction of as many as $|C|^2$ CCMFs, where C is the set of contexts in the CCG.

Now suppose that all the contexts from a given function were assigned the same set of CCMs. Now suppose that, for each context, $c = \langle f, R, e \rangle$, such that e is a call to function f', we compare each CCM s for function f with each CCM s' for function f' using only the conditions of c. That is, we prove that $\mathcal{H}^h [\![R]\!] \epsilon \Rightarrow s'\sigma_e \prec s$ or $\mathcal{H}^h [\![R]\!] \epsilon \Rightarrow s'\sigma_e \preceq s$ to determine which value to assign to $\phi_c(s, s')$. Clearly, this implies the conditions for assigning the values of $\phi_c(s, s')$ for each edge $\langle c, c' \rangle$ in the CCG. However, this results in the creation of only $|C|$ distinct CCMFs. For the majority of functions that can be proven terminating using our analysis, this simpler and more efficient method for computing CCMFs is enough.

Consider once again the `ack` example from Figure 27 on page 91. For the full analysis using "per-edge" CCMFs, the CCMs `(acl2-count x)` and `(acl2-count y)` must each be compared against each other, resulting in 4 proof attempts per CCMF. The CCG is complete, so there are 4 edges. This results in 16 proof attempts. On the other hand, the necessary CCMs can be proven non-increasing and decreasing using "per-context" CCMFs. This results in 8 proofs, effectively halving the time spent by the theorem prover. For problems where the "per-context" analysis fails, we can then refine each CCMF with the full information from each edge.

9.2.3.2 CCM Comparison Hierarchy

A second heuristic for minimizing the prover time is to put in place a hierarchy for comparing CCMs in CCMFs. Suppose we are constructing a CCMF for the CCMs of $c = \langle f, R, e \rangle$ and $c = \langle f', R', e' \rangle$. The hierarchy provides rules for when we attempt the proofs for determining the value of $\phi^c_{c'}(s, s')$ (or $\phi_c(s, s')$). The levels of the hierarchy are as follows:

1. **ACROSS** If $f = f'$, attempt the proofs for s and s' if $s = s'$. If $f \neq f'$, resort to the EQUAL level of the hierarchy.

2. **EQUAL** Attempt the proofs for s and s' when $\mathit{free}(s) = \mathit{free}(s'\sigma_e)$.

3. **ALL** Attempt the proofs for s and s' when $\mathit{free}(s) \subseteq \mathit{free}(s'\sigma_e)$.

4. **SOME** Attempt the proofs for s and s' when $\mathit{free}(s) \cap \mathit{free}(s'\sigma_e) \neq \emptyset$.

5. **NONE** Always attempt the proofs for s and s'.

The intuition is that if CCMs mention the same variables, they are more likely to be related. For example, CCMs $s =$ `(acl2-count x)` and $s'\sigma_e =$ `(acl2-count (- x 1))` would get analyzed in the first round of the hierarchy. CCMs $s =$ `(acl2-count x)` and $s'\sigma_e =$ `(acl2-count (- x y))` would get analyzed in the next. The last round would analyze CCMs such as $s =$ `(acl2-count x)` and $s'\sigma_e =$ `(acl2-count y)`, which are far less likely to be provably non-increasing or decreasing.

To effectively use the hierarchy, all the CCMF values are computed at each hierarchy level, with checks for well-foundedness interspersed. So, in the `ack` example from Figure 27 on page 91, the analysis would be able to compute all the necessary CCMF values in the **ACROSS** stage, resulting in only 4 proofs when using "per-context" CCMFs, as opposed to 8 if all values were compared.

9.2.4 Absorption

Absorption is a powerful tool, as it allows us to consider how the program behaves over several steps at certain points in the program without composing the entire transition relation, which can result in a quadratic increase in the size of the CCG. We use absorption in a conservative manner, avoiding increases in the size of the CCG. This is done by absorbing in the following two cases:

1. We absorb a context c if $\langle c, c\rangle$ is not an edge in the CCG and every successor of c has only one predecessor (*i.e.*, c itself).

2. We absorb a context c that has only one successor c' such that $\langle c', c'\rangle$ is not an edge in the CCG.

These two rules are illustrated in Figure 39 on the following page. Note that in both cases, the number of contexts is actually decreased. This means that, in addition to providing a more accurate representation of program behavior, this absorption heuristic has the

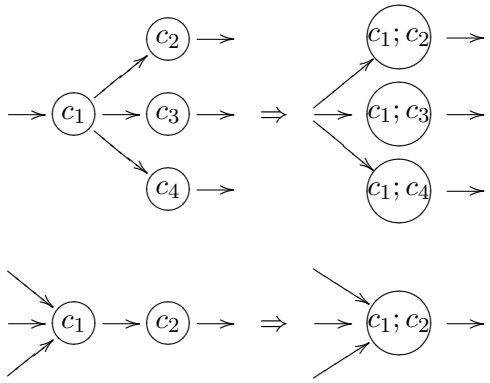


Figure 39: Illustrating the two absorption heuristics.

added benefit of decreasing the number of prover queries required in our analysis.

9.2.5 Proving Well-Foundedness

The well-foundedness analysis reduces to the size-change problem, whose solution is given in [65] (see Section 11.3 for a more detailed explanation), where they prove that the problem is PSPACE complete. However, a polynomial algorithm has been found which solves most size-change problems in practice [8]. While the full analysis tends to run quickly, it is still exponential in the worst case. We therefore use the polynomial algorithm and resort to the full algorithm when the polynomial algorithm fails.

9.2.6 Putting It All Together: Hierarchical CCG Analysis

Using all of the techniques of this chapter, we build a hierarchical CCG analysis algorithm that avoids unnecessary work whenever possible. In addition to the other observations made throughout this chapter, it is important to note that, even with these speed improvements, the CCG analysis still tends to be slower than ACL2's current analysis on those problems for which both can prove termination. We therefore allow ACL2 to attempt a termination proof using its current measure-based analysis early in the algorithm, with hints to limit ACL2 to a "reasonable" amount of work proving termination before we proceed with our own algorithm.

The entire algorithm is given in Figure 40 on the next page. We begin by building the naïve CCG for the given definitional axiom. That is, the resulting CCG will have an

```
1: SCCs := (SCC(Build-Naïve-CCG(d))
2: SCCs := Perform-Absorption(SCCs)
3: Annotate(SCCs)
4: Per-Context-CCMFs(SCCs)
5: SCCs := SCP(SCCs)
6: if SCCs = ∅ then
7:    return terminating
8: end if
9: if Measure-Admissible(d) then
10:   return terminating
11: end if
12: Refine-CCMFs(SCCs, T, 0, ACROSS)
13: SCCs := SCP(SCCs)
14: if SCCs = ∅ then
15:   return terminating
16: end if
17: for all CH ∈ {ACROSS, EQUAL, ALL, SOME, NONE} do
18:   Refine-CCMFs(SCCs, T, 1, CH)
19:   SCCs := SCP(SCCs)
20:   if SCCs = ∅ then
21:      return terminating
22:   end if
23: end for
24: SCCs := Prune-CCGs(SCCs)
25: for all CH ∈ {ACROSS, EQUAL, ALL, SOME, NONE} do
26:   Refine-CCMFs(SCCs, NIL, 1, CH)
27:   SCCs := SCP(SCCs)
28:   if SCCs = ∅ then
29:      return terminating
30:   end if
31: end for
32: SCCs := SCT(SCCs)
33: if SCCs = ∅ then
34:   return terminating
35: end if
36: return failed
```

Figure 40: Hierarchical CCG algorithm.

edge from c to c' when the call of c is a call to the function containing c'. This CCG is immediately broken into SCCs. Next, we perform absorption on each SCC as described in Section 9.2.4. The SCCs are then annotated by assigning CCMs to each context so that all the contexts from a given function have the same CCMs, as described in Section 9.2.1. The CCMFs for the SCCs are then built per context as described in Section 9.2.3.1, using only built-in clauses for the proofs. The SCCs are then analyzed using SCP, which applies the polynomial size-change algorithm to each SCC, returning only those SCCs for which the analysis failed to prove well-foundedness. If it returned no SCCs, we have proven termination, and we stop. Otherwise, we use ACL2 to try to prove termination using its measure-based analysis, limited to one induction. If this succeeds, we stop.

In the next step, we call Refine-CCMFs, which take the annotated SCCs, a value indicating whether to use per-context CCMFS (T for yes, NIL for no), a maximum number of inductions to use, and a CCM comparison hierarchy as given in Section 9.2.3.2. In this case, we continue to use per-context CCMFs, and refine with 0 inductions using the ACROSS level of the comparison hierarchy. We then attempt the polynomial size-change analysis again.

In the next stage of the analysis, we cycle through the CCM comparison hierarchies, refining the CCMFs using per-context CCMs, and 1 induction. We call the polynomial size-change analysis after each refinement step.

If this fails, we prune the SCCs using 1 induction to determine when it is safe to remove an edge. We then repeat the previous loop, this time with per-edge CCMFs. If all else fails, we attempt to prove termination using the full exponential size-change algorithm, which we call SCT here. If this fails, the algorithm stops in failure.

9.3 Bibliographic Notes

Due to the overlap in bibliographic data between this and the next chapter, we postpone discussion to Section 10.2.

9.4 Summary

We presented a detailed algorithm for the CCG analysis that is hierarchical in nature. Beginning with lightweight versions of the CCG analyses, we add more power as needed, until the full CCG analysis is applied when all else fails. Techniques for creating these lightweight analyses include

- Computing CCMFs on a per-context basis instead of a per-edge basis.
- Following a hierarchy for comparing CCMs when computing CCMFs that allow us to focus first on those CCMs that are most likely to be helpful to the analysis.
- Using lightweight theorem proving techniques such as built-in clauses, and limiting the number of inductions allowed during proof attempts.
- Using the polynomial algorithm approximating the size-change analysis to determine well-foundedness whenever possible.

The resulting algorithm allows for efficient solving of simpler termination problems, while maintaining the full power of the CCG analysis. In addition, we presented heuristics for guessing and propagating CCMs, and performing absorption on CCGs in such a way as to avoid increasing the size of the CCG.

CHAPTER X

EMPIRICAL EVALUATION

In this chapter, we empirically evaluate the theory of Calling Context Graphs introduced in Chapter 8, as well as the hierarchical implementation of the CCG termination analysis presented in Chapter 9. We have implemented the hierarchical algorithm in ACL2 and run it over the entire ACL2 regression suite (for a description of the regression suite, refer to Section 2.4). Recall that every function in the regression suite has been proven to terminate in ACL2 using its traditional measure-based analysis. In many cases, ACL2 could not automatically prove termination, and human guidance was required. Therefore, in order to accurately judge the effectiveness of our analysis to automatically prove termination, we need to factor out this assistance. We distinguish two forms of guidance.

The first is *explicit* guidance, in which the user provides the measure or gives explicit hints to the theorem prover on how to prove termination. This form of assistance is easy to factor out, as we can simply ignore the measure and prover hints. For example, here is a a function from the regression suite that specifies an explicit measure: an ordinal constructed using ordinal multiplication (`o*`), ordinal addition (`o+`), the first infinite ordinal (`(omega)`), and several auxiliary functions (*e.g.*, `tuple-set-max-first`).

```
(defun tuple-set->ordinal-partial-sum (k S i)
  (declare (xargs
            :measure (o+ (o* (omega) (nfix k))
                         (nfix (- (tuple-set-max-first 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))))))
```

Our system automatically proves that the above function terminates without the measure information provided by the user. It does so using the CCMs `(acl2-count k)`, which is always included as a CCM by default, and `(acl2-count (- (tuple-set-max-first S) i))`, which is included by our heuristics, since `(<= (tuple-set-max-first S) i)` is one of the guards.

The other form of guidance is *implicit* guidance, which is given when users prove auxiliary lemmas which help ACL2 to complete the termination proof. While it is difficult to identify the theorems used solely to prove termination, it is clear that many termination proofs require auxiliary lemmas and substantial human effort.

Factoring out implicit guidance is more difficult than factoring out explicit guidance. Ideally, we would like to ignore all the auxiliary lemmas required for termination proofs, while leaving the rest of the lemmas and theorems which comprise the theory being developed by the user. However, it is impossible to tell exactly which lemmas were added specifically for termination proofs, and which were added as part of the overall theory being developed. In order to estimate the performance of our termination analysis, we therefore designed two experiments to capture the two "extremes" of implicit guidance. In the first, we ignore all theorems and enable all definitions that occur beyond the ground-zero theory. That is, the only parts of the theory we are allowed to use are the definitions and executable counterparts (for evaluating ground terms) of functions, along with the bare-bones theory provided by ACL2 on start-up. This ensures that all implicit guidance has been factored out and then some, giving us a lower bound on our tools performance in "typical" circumstances. The other experiment uses the entire current theory when proving termination. Any theorems and definitions that are enabled in the theory at the point of definition are used to prove termination. This gives us an upper bound on our tool's performance, as it includes any implicit guidance provided by the user the first time around. In both experiments, no explicit guidance is given, ACL2's current analysis is run under the same conditions as the CCG analysis, and a 150 second time limit is given to both ACL2 and the

Table 7: Result summary when theorems are disabled

Problems	Total	CCG	ACL2
Non-Trivial	**1763**	**1329 (75.38%)**	**1056 (59.89%)**
Recursive	4349	3915 (90.02%)	3642 (83.74%)
All	11303	10869 (96.16%)	10596 (93.74%)

CCG algorithm.

A summary of the results from the first experiment, in which all the theorems were disabled and all the theorems enabled, is given in Table 7. The results are broken down into three classes of functions. The first, and most interesting category are the "non-trivial" functions. Recall from Section 9.2.2, that ACL2 begins its termination proof by using the extremely lightweight prover technique using built-in clauses, which matches simple but widely used recursive behaviors. For example, these include subtracting 1 from a natural number until it is 0, walking down a list one element at a time until reaching the end. Such functions are trivially proven terminating, and do not provide much information on the relative effectiveness of both termination analyses. Factoring all such functions out, we find that our analysis can solve over 75% of the remaining 1,762 functions, while ACL2 can solve about 60% of them. This is a 15% improvement when dealing with moderate to difficult functions. The second line reports the results for all recursive functions, both trivial and non-trivial. Even with the trivial functions thrown in, there is a significant 6.25% improvement. The final line reports the results for all functions, including non-recursive functions, and is included for completeness.

In order to gauge the effectiveness of the hierarchical implementation of the CCG algorithm, we present the in-depth results from our first experiment in Table 8 on the next page. Here, the results are reported for each level of the hierarchy, including timing results. The CPN header denotes if the CCMFs are computed per-context (see Section 9.2.3.1. BIC specifies that all the prover queries used only built-in clauses, and no call to the full theorem prover (see Section 9.2.2). The CH column indicates what level of the CCM comparison hierarchy was used (see Section 9.2.3.2). The Ind header specifies the number of inductions allowed in each proof (see Section 9.2.2). Num indicates the number of problems solved

Table 8: Detailed results when theorems are disabled

CPN	BIC	CH	Ind	Num	PTime	WFTime	ACL2	ATime	CTime
T	T	NONE	N/A	3245	.006	0	3072	.001	.003
ACL2 w/ 1 induction				564	.204	0	564	.151	.204
T	NIL	ACROSS	0	86	8.923	0	0	0	0
T	NIL	ACROSS	1	9	10.032	.002	3	.436	.970
T	NIL	EQUAL	1	0	0	0	0	0	0
T	NIL	ALL	1	0	0	0	0	0	0
T	NIL	SOME	1	0	0	0	0	0	0
T	NIL	NONE	1	0	0	0	0	0	0
NIL	NIL	ACROSS	1	10	2.629	0	0	0	0
NIL	NIL	EQUAL	1	0	0	0	0	0	0
NIL	NIL	ALL	1	0	0	0	0	0	0
NIL	NIL	SOME	1	1	38.010	.010	0	0	0
NIL	NIL	NONE	1	0	0	0	0	0	0
Using Exponential Alg.				0	0	0	0	0	0
CCG Failed				434	63.389	.001	3	.993	55.110

by that stage of the hierarchical analysis. PTime is the average time spent theorem proving for problems solved at that level. WFTime presents the average time spent doing the well-foundedness analysis. The ACL2 column gives the number of problems solved at this level that were also solved by ACL2. The ATime and CTime columns give the average time spent proving termination by ACL2 and the CCG analysis, respectively, for those problems that both ACL2 and the current level of the hierarchical analysis solved.

Unsurprisingly, given the number of functions reported as "trivial" in Table 7 on the preceding page, a large number of problems are solved using only built-in clauses. Such problems are solved in .005 seconds on average. Note that there were 173 problems that the CCG analysis could solve using this extremely lightweight analysis, that ACL2 could not solve at all. An example of such a problem is the `ack` example, as we noted in Section 9.2.2. Many of the rest of the problems are solved using ACL2's termination analysis, limited to one induction, in the second step of the analysis. Such problems are also proven quickly, taking only .204 seconds on average. However, there are still 106 that are handled by later stages of the analysis, only 3 of which can be proven terminating in ACL2 with no user guidance. The third and fourth stages, using the ACROSS level of the comparison hierarchy

are significantly slower than the first two stages, but still relatively fast and effective. Note that no functions are proven terminating using the EQUAL, ALL, SOME, or NONE levels of the comparison hierarchy. While this may indicate that stages 5, 6, and 7 can be removed from the algorithm, we prefer to keep them in, since using per-context CCMFs leads to better measured subsets (See Section 11.4). In the last stage, given in the second-to-the-last row of the table, we use the full exponential algorithm. But note that there were no problems that could not be handled by the polynomial approximation that could be handled by the full exponential algorithm.

The last column gives the failed CCG analysis attempts. We examined these failures to ascertain why the CCG analysis failed. We found that in 53 of these cases, the failures were due to ACL2's lack of theorems for reasoning about the `floor` and `mod` functions in the ground-zero theory. In all of these cases, reasoning about these functions played an integral role in the surrounding theory, and would have been there even if these functions could be proven terminating without it. In other words, in typical ACL2 usage, these functions would have been proven terminating by the CCG analysis. Another 62 of these functions could not be proven because they involved encapsulated functions for which the full definition was not available. Recall that the encapsulate feature of ACL2 allows users to admit constrained functions into the theory without admitting the full definition. Without the theorems constraining these functions' behavior, no analysis would be able to reason about these functions in order to prove termination. The remainder of the failures seemed to be legitimate failures. The reasons varied from cases in which the necessary measures were too complex to guess with our heuristics, to cases in which the termination proof seemed to require implicit guidance. However, factoring out the failures due to `floor`, `mod`, and encapsulation, we find that our analysis would actually be able to solve at least 1444 or 81.90% of the non-trivial termination problems, a 5.52% better performance than the numbers for this experiment imply.

A summary of the results for the second experiment, in which the entire current theory is used to prove termination, is given in Table 9 on the next page. Note that the advantage of 15% for our analysis over ACL2's has grown to 20% in this experiment. Also, note that

Table 9: Result summary when theorems are enabled

Problems	Total	CCG	ACL2
Non-Trivial	**1762**	**1535 (87.11%)**	**1180 (66.96%)**
Recursive	4346	4119 (94.77%)	3764 (86.60%)
All	11295	11068 (97.99%)	10713 (94.84%)

Table 10: Detailed results when theorems are enabled

CPN	BIC	CH	Ind	Num	PTime	WFTime	ACL2	ATime	CTime
T	T	NONE	N/A	3240	.006	0	3071	.001	.003
ACL2 w/ 1 induction				689	.099	0	689	.066	.099
T	NIL	ACROSS	0	175	2.680	0	0	0	0
T	NIL	ACROSS	1	4	6.222	0	1	.870	9.070
T	NIL	EQUAL	1	0	0	0	0	0	0
T	NIL	ALL	1	0	0	0	0	0	0
T	NIL	SOME	1	0	0	0	0	0	0
T	NIL	NONE	1	0	0	0	0	0	0
NIL	NIL	ACROSS	1	10	.903	0	0	0	0
NIL	NIL	EQUAL	1	0	0	0	0	0	0
NIL	NIL	ALL	1	0	0	0	0	0	0
NIL	NIL	SOME	1	1	26.250	.020	0	0	0
NIL	NIL	NONE	1	0	0	0	0	0	0
With Ruler Rewriting				0	0	0	0	0	0
Using Exponential Alg.				0	0	0	0	0	0
CCG Failed				227	45.503	.001	3	.873	48.576

the overall effect of enabling theorems is a 11.73% improvement in performance for these problems.

The detailed results from the second experiment are given in Table 10. Much of this data looks similar to that from the previous experiment. However, there is a significant decrease in the running time. In typical ACL2 usage, users build up theories about function definitions, and then disable those definitions. The result is that the theorem prover spends less time reasoning about definitions and re-proving already proven results. As a result, proof attempts are faster, resulting in the significant speed-ups observed here.

Table 11: Results when analyzing examples from the PolyRank distribution.

	1		2		3		4		6	
O	0.30	†	0.05	†	0.11	†	0.50	†	0.10	†
P	1.42	✓	0.82	✓	1.06	†	2.29	†	2.61	†
PR	0.21	✓	0.13	✓	0.44	✓	1.62	✓	3.88	✓
T	453.23	✓	61.15	✓	T/O	-	T/O	-	75.33	✓
CCG	4.69	✓	1.20	✓	T/O	-	T/O	-	120.01	✓

	7		8		9		10		11		12	
O	0.17	†	0.16	†	0.12	†	0.35	†	0.86	†	0.12	†
P	1.28	†	0.24	†	1.36	✓	1.69	†	1.56	†	1.05	†
PR	0.11	✓	2.02	✓	1.33	✓	13.34	✓	174.55	✓	0.15	✓
T	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	10.31	†
CCG	19.83	✓	37.79	✓	365.97	†	T/O	-	T/O	-	T/O	-

10.1 Comparisons with Other Tools

In this section, we empirically contrast our tools to several state-of-the-art termination analyses.

This is a difficult task, as several current state-of-the-art analyses are designed and implemented for different domains. These include nondeterministic algebraic transition systems, C programs, and Term Rewriting Systems (TRSs). Because of these different domains, large-scale experimental comparisons are infeasible. Instead, we provide comparisons using hand-translated examples.

We compare our analysis empirically with four existing analyses: OctaTerm and PolyTerm [9], PolyRank [14], and Terminator [27]. The benchmarks and performance of these four tools is from [9]. There are two sets of examples. Unfortunately, we were unable to recreate the original experimental data, so we present the data from the original paper here for these four analyses, and add results for our own CCG analysis. OctaTerm, PolyTerm, and PolyRank were run on a 2GHz AMD64 processor using Linux 2.6.16, while Terminator was run on a 3GHz Pentium 4 using Windows XP SP2. Our own analysis was run on a dual 2.8GHz Intel Xeon machine running Linux 2.6.9. Our analysis does not make use of the multiple processors.

The first set of examples are distributed with the PolyRank tool. They were encoded by the PolyRank authors, and represent simply stated but "tricky" arithmetic loops. Included are two different GCD algorithms, and one that computes the McCarthy 91 function. It is important to note here that the McCarthy 91 encoding is not reflexive, as is the traditional encoding of McCarthy 91, but is a loop that is semantically equivalent. This is an important distinction as the difficulty of proving McCarthy 91 terminating lies mainly in its reflexive nature.

The results for these examples are listed in Table 11 on the previous page. Here O is OctaTerm, P is PolyTerm, PR is PolyRank, T is Terminator, and CCG is our own analysis. The numbers indicate the time of the analysis in seconds. A timeout of 500 seconds was used. A ✓ indicates a successful termination proof, ∅ signifies that the analysis correctly labeled the example as non-terminating, and † indicates a failure to prove a terminating example to be terminating.

PolyRank analyzes non-deterministic polynomial loops over the reals. Since ACL2 does not have a representation for the reals, we restricted our analysis to the rationals. Non-determinism is modeled using an encapsulated function whose exact definition is hidden from the CCG analysis. Our heuristics for choosing CCMs are designed to be effective for typical code written by ACL2 users. Since these examples do not fall into that category, we altered one of our heuristics: the shrinking differences heuristic described in Section 9.2.1.2. When one of the context conditions is of the form `(<` e e'`)` or `(not (<` e' e`))`, we add `(nfix (floor (+ 1 (-` e' e`)) 1))` to the list of CCMs. Here `(nfix x)` returns `x` if `x` is a natural number, and `0` otherwise. The expression `(floor x y)` returns the floor of the value of `x` divided by `y`. We also loaded an existing library for reasoning about rational numbers in ACL2.

Not surprisingly, PolyRank performs the best, proving all the examples terminating. Of the remaining tools, our CCG analysis performs the best, proving 5 of the 11 benchmarks terminating. Terminator and PolyTerm prove 3 terminating, and OctaTerm fails on all examples.

The other set of examples are taken from Windows device drivers, which are the domain

Table 12: Results when analyzing examples taken from Windows device drivers

	1		2		3		4		5	
O	1.42	✓	1.67	∅	0.47	∅	0.18	✓	0.06	✓
P	4.66	✓	6.35	∅	1.48	∅	1.10	✓	1.30	✓
PR	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
T	10.22	✓	31.51	∅	30.65	∅	4.05	✓	12.63	✓
CCG	0.66	✓	90.15	∅	33.94	∅	0.65	✓	0.44	✓

	6		7		8		9		10	
O	0.53	✓	0.50	✓	0.32	✓	0.14	∅	0.17	✓
P	1.60	✓	2.65	✓	1.89	✓	2.42	∅	1.27	✓
PR	T/O	-	T/O	-	T/O	-	T/O	-	0.31	✓
T	67.11	✓	298.45	✓	444.78	✓	T/O	-	55.28	✓
CCG	0.67	✓	0.15	✓	0.15	✓	21.73	∅	2.69	✓

for which Terminator is designed and implemented. These results are given in Table 12. PolyRank is unable to prove termination for all but one of these examples. However, the other analyses correctly analyze all the benchmarks, with the exception of benchmark 9, for which Terminator times out. While drawing conclusions about timing is difficult given that the analyses have been run in different machines, it is still clear that our analysis seems to be competitive with OctaTerm and PolyTerm on all but 3 examples (2, 3, and 9), and outperforms Terminator by one or more orders of magnitude on 8 of the 10 examples.

10.2 Bibliographic Notes

Termination is one of the oldest problems in computing science and it has received a significant amount of attention. Here we will briefly review recent work on automating termination analysis.

One of the most often cited techniques for the proving termination of programs is called the *size change principle* [65]. This method involves using a well-order on function parameters, analyzing recursive calls to label any clearly decreasing or non-increasing parameters. Then, all infinite paths are analyzed to ensure that some parameter never increases and infinitely decreases over each path. We use this path analysis in step 7 of our algorithm.

The size change principle has several limitations, *e.g.*, it does not show how to take governors into account and it does not provide any method for determining the sizes of the outputs of user-defined functions. Both of these considerations are almost always important for establishing termination in realistic programming languages.

Much work has gone into developing termination analyses for term rewriting systems and logic programs, *e.g.*, [2, 43, 25]. However, these methods do not scale to the complexity of total functional programming languages. For example, the authors of the AProVE tool [43], winner of the termination competition for TRSs, provided us with a version of AProVE specifically tweaked to prove the termination of TRSs obtained by hand-translating ACL2 functions. However, this tool was unable to prove the termination of the following trivial function definition in ACL2:

```
(defun f (x)
  (if (zp x)
      0
    (+ 1 (f (- x 1)))))
```

There has been a significant amount of work on proving the termination of algebraic loops, *i.e.*, loops whose behavior is governed by arithmetic over the integers, rationals, or reals. One of the interesting questions in this domain is: for what classes of algebraic loops is the termination question decidable? Podelski and Rybalchenko found a class of loops with linear behaviors and no nesting for which the question of whether a linear ranking function exists is decidable [93]. Tiwari found another class of linear loops for which termination is decidable even if no linear ranking function exists [111]. Recently, this result was generalized to a larger class of linear loops [16].

Recently, there has been work asking similar decidability questions about loops involving lists [12]. What was found is that both safety properties and termination are decidable for a restricted form of non-nested loops involving singly-linked lists of integers. Such a result is especially applicable to loops that sort or reverse lists.

The problem of finding solvable subsets of the termination problem is an interesting and useful one. Such results can be used in general termination analyses to prove termination of individual loops that match one of these solvable classes. As we shall see shortly, the

Terminator termination analysis does this. However, such tools do not satisfy our need for a general analysis that can be used in the presence of any looping behavior. None of these analyses, for example, can deal with trees, graphs, or complex numbers.

In addition to linear algebraic loops, several analyses have been developed to perform termination analysis on loops with polynomial behaviors [15, 29]. As we saw in Section 10.1, such tools are not complete but are effective at proving the termination of some kinds of polynomial loops.

The Terminator, to which we empirically compare our analysis in Section 10.1, is due to Cook, Podelski, and Rybalchenko. At the heart of the Terminator algorithm is a result by Podelski and Rybalchenko which states that a program is terminating if there is some finite set of well-founded relations whose union contains the transitive closure of the transition relation of the program [94]. In other words, if every valid path through the program can be shown to be well-founded by one of the well-founded relations contained in the finite set, the program is terminating. This leads to an abstraction-refinement framework for termination [26]. The core algorithm, presented in [27], begins with an empty set of well-founded relations, W. A new program is built based on the one being analyzed that maintains history variables to keep track of previous variable values, and contains error states that are entered when the current and previous variable values do not satisfy any of the relations in W. A safety checker is used to check the property that the error state is never entered (and therefore that the transitive closure of the transition relation is in W). If the safety checker fails, it returns a counter-example, which is used to generate a new ranking function using a complete method for finding linear ranking functions for loops with linear semi-algebraic behavior [93]. That ranking function is added to W, and the process is repeated until no counter-example is found by the safety checker. At this point, the program has been proven terminating. Recent improvements to the algorithm include a method for proving termination in the presence of shape-shifting heaps [10] and in the presence of multiple threads [28].

As we saw in Section 10.1, our analysis compares well with Terminator on our hand-translated examples. We are able to identify a program as terminating or non-terminating

```
main () {
   int n = nondet();
   int x = nondet();
   int y = nondet();

   if (n >= 0) {
      while (x = 0) {
         if (y <= 0) {
           y = n;
           if (x < 0)
             x = -1 - x;
           else
             x = 1 - x;
         } else {
           y--;
           if (x < 0)
             x = x + 1;
           else
             x = x - 1;
         }
      }
   }
}
```

```
(defun main (n x y)
  (cond ((or (zp n)
             (not (integerp x))
             (= x 0))
         y)
        ((zp y)
         (if (< x 0)
             (main n (- -1 x) n)
           (main n (- 1 x) n)))
        (t
         (if (< x 0)
             (main n (+ x 1) (- y 1))
           (main n (- x 1) (- y 1))))))
```

Figure 41: An example illustrating a fundamental difference between Terminatior and CCG analysis.

on every problem Terminator correctly analyzes, plus three more.

As an example of the kinds of problems that our CCG analysis can handle but Terminator cannot, consider the code in Figure 41. The definition in C on the left and the ACL2 code on the right have the same looping behavior: the integer `x` gets closer to 0 with every step, but every `n` steps, `x` is also negated. So, `x` keeps switching between being positive and negative, but always approaches `x`. Terminator cannot prove this program terminating. The problem is that there is no linear ranking function that decreases when `y` is 0 and `x` is non-zero. This is when `x` switches between being positive and negative. The CCG analysis proves this terminating automatically, using the absolute value of `x` as a CCM for all contexts. In other words, the CCG analysis can prove termination of functions that the Terminator cannot handle by using non-linear measures.

Perhaps more interestingly, the CCG analysis can prove termination using only the linear

CCMs `x` and `(- x)`. This is because the CCG analysis can compare different measures as well as comparing a measure to itself. This is something that the Terminator framework can not do, since it must match each loop to a single ranking function. It would need a rank finding algorithm that could suggest the absolute value of `x` as a ranking function.

This suggests a difference between the CCG framework and Terminator framework at a more fundamental level that is independent of the techniques used to guess measures or ranking functions. One of the keys to the automation provided by Terminator and our CCG analysis is that, instead of having to construct one ranking function that decreases over every transition, these frameworks break the problem into smaller pieces so that simpler ranking functions or measures can be found for each piece. What this example suggests is that for certain instances, the CCG algorithm can prove termination using simpler measures than those required by Terminator to prove termination for that same program. Specifically, this appears to be true when termination can be proven by comparing the values of multiple measures with each other, rather than using one ranking function per loop.

Efficiency is another significant difference between Terminator and the CCG analysis. As we saw in Section 10.1, our analysis often outperforms Terminator. In addition, using our hierarchical algorithm, we can prove the termination of the function for calculating the nth Fibbonacci number and for Ackermann's function (see Figure 27 on page 91) in 0.05 seconds apiece. Terminator takes 300 and 600 seconds, respectively, to prove termination for these two problems. As we saw in Chapter 10, our CCG analysis provides impressive results even when limited to 150 seconds. We explore two possible reasons for this time difference.

The first is that Terminator uses the SLAM safety checker [3] to perform its safety checks. SLAM is designed to check that device drivers interface properly with operating system APIs through the use of model checking, theorem proving, and predicate abstraction. It uses heuristics specifically geared toward the efficient analysis of device drivers for these types of properties. These heuristics are not designed to find termination related errors, which may significantly slow down Terminator's performance. In contrast, our analysis uses static analysis and theorem proving to immediately abstract away information that

is irrelevant to the termination analysis. This is the purpose of the calling context graph. Doing this analysis up front allows us to simplify the analysis, leading to better runtimes.

One of the strengths of Terminator is its use of counter-example guided abstraction and refinement (CEGAR) to analyze and learn from failures. However, Terminator begins its analysis with no well-founded relations ensuring that every loop will fail to be proven terminating on the first iteration of the CEGAR loop. Since each failure leads to the generation of a ranking function designed for a single loop, this can lead to several iterations of the CEGAR loop to find ranking functions that might be found using simple up-front analyses. The CCG analysis does the opposite. It uses heuristics to find candidate CCMs up front that work well in practice, but does not use refinement to learn from failures. An interesting question is whether these two techniques could be effectively combined, so that simple ranking functions or CCMs could be found with lightweight heuristic analyses, and more difficult analyses could be used to analyze failed loops and find new CCMs or ranking functions.

A new class of termination analyses based on the same general principle as Terminator were recently developed in [9]. Among these are the OctaTerm and PolyTerm analyses with which we compared our analysis in Section 10.1. This work provides a framework for using invariant analyses to reason about termination. The idea is to find the invariant at the beginning of a loop, seed it with history variables, and ask for the loop invariant. The resulting invariant describes the change in values after any number of iterations through the loop. This invariant can then be fed to a well-foundedness analysis to determine if the given invariant describes a well-founded relation between the history variables and the current state. If the invariants for all the loops in a program can be determined to be well-founded, then by the previously mentioned result by Podelski and Rybalchenko [93], the program is terminating.

The OctaTerm and PolyTerm analysis use this framework along with the Octagonal [81] and the New Polka Polyhedra library [52] invariant analyses, respectively. As the experimental results of this chapter show, our analysis compares favorably to these two analyses. As with terminator, we correctly analyze more examples than either of these analyses. In

addition, these analyses are limited to the domain of their underlying invariance analysis. In the cases of OctaTerm and PolyTerm, these analyses are limited to algebraic programs, and can not handle data structures, as our analysis can.

Another instantiation of the previously mentioned framework for using invariant analyses to prove termination uses the Sonar heap analysis, which analyzes the shape and size of data structures on the heap. Such a termination analysis can therefore reason about the changing sizes of data structures, which are helpful when walking through a list, tree, or graph. Such analyses have been used in ACL2 to prove termination since its inception. The difference is that ACL2 has a built-in notion of such data structures, as opposed to C, where they must be built using structures and pointers. Sonar is therefore not of use to our CCG analysis currently, as the size and shape of data structures can be determined in applicative first-order functional languages without a heap analysis. However, such an analysis would be helpful should we decide to adapt the CCG analysis to a language such as C.

CHAPTER XI

ACL2 INTEGRATION

In this chapter, we explore the issues involved in integrating the CCG analysis with ACL2. Our primary obligation is to ensure that if ACL2 uses our analysis, it will not alter the logic. To do this, we more formally describe ACL2's current termination requirement, known as *measure admissibility*, propose the concept of *CCG admissibility* in Section 11.2, and prove that CCG admissibility implies measure admissibility in Section 11.3. This result ensures that the integration of the CCG analysis into ACL2 would result in no change to the ACL2 logic.

In Section 11.4, we turn to the engineering challenges involved in integrating the CCG analysis with the theorem prover in such a way so that the heuristics and algorithms used by the prover are not negatively affected.

11.1 Preliminaries: Measure Admissibility in ACL2

In order to demonstrate that our analysis does not change the ACL2 logic, we need a more formal understanding of how ACL2 deals with termination. In ACL2, all functions must be proven to be terminating using *measures*, in a way similar to that described in Section 2.1.1. However, ACL2 has additional requirements that ensure that the function or functions are provably terminating in the ACL2 logic. We therefore begin by reviewing several important concepts in first-order logic for understanding ACL2's logic.

The set of function symbols occurring in a set, F, of formulas is called the *language* of F. A *theory* is a set, T, of formulas that are first-order derivable from a set of *axioms*, F, whose extra-logical symbols all occur in F. These formulas are often called *theorems* of F or of T. A theory, T_1, is a *conservative extension* of a theory, T_0, if $T_0 \subseteq T_1$ and for every theorem, $\phi \in T_1$ such that ϕ is in the language of T_0, ϕ is a theorem of T_0.

The primitives of applicative Common Lisp are axiomatized in ACL2, as are the basic data types, including natural numbers, integers, rationals, complex rationals, ordered pairs,

symbols, characters, and strings. These form the initial axioms of ACL2. Upon start-up, ACL2 provides the user with a conservative extension of these axioms, known as the *ground-zero* theory, GZ. Included in this theory as of version 2.8, is the representation of the ordinals up to ε_0 presented in Section 4.3. They are recognized by the function `o-p` That is, `(o-p` v`)` is true if and only if v is an ordinal in ACL2's representation. The ordering relation on the ordinals in ACL2's representation is `o<`. That is, if `(o-p` v_1`)` and `(o-p` v_2`)`, then `(o<` v_1 v_2`)` would return true if and only if v_1 represented an ordinal less than that represented by v_2.

ACL2 provides techniques for extending a given theory. Two of these are deriving new theorems from T and by adding new *definitional axioms*. A theorem is is simply an expression, e, in the ACL2 language for which it is first-order derivable that for all valuations of $free(e)$, e evaluates to a non-`nil` value. A definitional axiom is given by a set of `defun` statements, d, as defined in Section 2.1. The new theory created is

$$T' = T \cup (\bigwedge_{1 \leq i \leq k} (f_i \ x_1^{f_i} \ \ldots \ x_{ar(f_i)}^{f_i}) = e^{f_i})$$

However, not just any function definition can be added as an axiom to ACL2. The goal is to ensure that T' is a consistent and conservative extension of T. For example, the definition `(defun f (x) (not (f x)))` should not be allowed since it leads to an inconsistency. Therefore, ACL2 has the notion of a *definitional principle*, which governs when a definitional axiom is safe to add to a theory. The following are used to define the definitional principle.

Definition 11.1.1. A *measure* for d defined over a theory, T, associates each f_i with a first-order definable function, m_{f_i} with respect to T of the same arity as f_i.

For the remainder of this section, we assume a fixed measure as denoted in the previous definition. We now define the parallel notion of well-founded structures in ACL2. Rather than reasoning directly about possibly infinite sets, ACL2 uses predicates that decide whether or not an object from the ACL2 universe is in a given set. Also, since the ordinals up to ε_0 form the basis of well-founded reasoning in ACL2, we must demonstrate an embedding from our well-founded structure into ε_0 that preserves the ordering. In the

following definition, we show how this is done using a predicate, mp, an ordering, mr, and a mapping from the set recognized by mp into the ordinals, mm.

Definition 11.1.2. $\langle mp, mr\rangle \in Fun^2$ are *T-well-founded* if mp is a unary function in the language of T, `mr` is a binary function in the language of T, and there exists a binary function, mm in the language of T such that the following is a theorem in T:

```
(and (implies (mp x) (o-p (mm x)))
     (implies (and (mp x1) (mp x2)
                   (mr x1 x2))
              (o< (mm x1) (mm x2))))
```

Definition 11.1.3. Let $\langle mp, mr\rangle \in Fun^2$ be T-well-founded. Then d is *measure admissible* in logic T if, for all $1 \leq i \leq k$, the following conditions hold:

- `(`mp `(`m_{f_i} $x_1^{f_i}$ ... $x_{ar(f_i)}^{f_i}$`))` is a theorem of T.
- For all $p \in Pos(e^{f_i})$ such that $e = e^{f_i}|_p$ is a function call of the form `(`f_j e_1 ... $e_{ar(f_j)}$`)` to a function, f_j, defined in d, and $R =$`(and` r_1 ... r_n`)` where $rulers(e^{f_i}, p) = \{r_1, \ldots, r_n\}$, then `(implies` R `(`mr `(`m_{f_j} e_1 ... $e_{ar(f_j)}$`)` `(`m_{f_i} $x_1^{f_i}$ ... $x_{ar(f_i)}^{f_i}$`)))` is a theorem of T.

The definitional principle states that a function must be measure-admissible before it is admitted into the logic.

11.2 CCG Admissibility

In the same vein as the previous section, we develop the notion of CCG Admissibility, which provides a new condition under which to admit new definitional axioms. Let T be a fixed theory and d be the new definitional axioms. We begin with the concept of *ruler semi-complete* calling contexts, which mirror the callsites and their rulers.

Definition 11.2.1. The set of *ruler semi-complete* calling contexts for d is the semi-complete set of contexts whose conditions are the rulers of the call in the body of the function in which the call appears.

Using ruler semi-complete contexts, we build a CCG as follows.

Definition 11.2.2. A *semi-complete T-CCG* for d is a CCG, $\mathcal{G} = (C, E)$ such that C is the set of ruler semi-complete calling contexts of d and for every pair $\langle c, c' \rangle \in C^2$ where $c = \langle f, \{r_1, r_2, \ldots, r_n\}, e \rangle$ and $c' = \langle f', \{r'_1, r'_2, \ldots r'_m\}, e' \rangle$ such that e is a call to f', $\langle c, c' \rangle \in E$ unless $T \vdash$ `(not (and` r_1 r_2 ... r_n $r'_1\sigma_e$ ... $r'_m\sigma_e$`))`.

For the CCM annotation, we need to be sure that it is first-order provable that the expressions always map into a given set. We do this by requiring that there be a first-order definable function *mp* that recognizes the elements of some set, and that we can prove in theory T that the CCMs of the CCG always satisfy *mp*.

Definition 11.2.3. Given a semi-complete T-CCG,$\mathcal{G} = (C, E)$, and *mp* $\in$ *Fun* a unary function in the language of T, a mapping $m : C \to \mathcal{P}(Expr)$ is a *T-expressible mp CCM annotation*, if for all $c \in C$, for all $s \in m(c)$, s is an expression over the language of T and $T \vdash$ `(`*mp* s`)`

In practice, we want *mp* to recognize elements in some woset. We therefore need the concept of T-well-orderedness, which mirrors the previously defined concept of T-well-foundedness. In addition, we will require that *mp* be embeddable into some ordinal strictly less than ε_0.

Definition 11.2.4. A pair $\langle mp, mr \rangle \in Fun^2$ is *T-well-ordered* if there exists *mm* such that the following is a theorem in T:

```
(and (implies (mp x) (o-p (mm x)))
     (implies (and (mp x) (mp y) (mr x y))
              (o< (mm x) (mm y)))
     (implies (and (mp x) (mp y))
              (or (mr x y) (mr y x) (equal x y))))
```

Further, $\langle mp, mr \rangle$ is *T-ordinal-bound* by $v \in Val$ such that the following is a theorem in T:

```
(and (o-p v)
     (not (equal v 0))
     (implies (mp x)
              (o< (mm x) v)))
```

Note that it is first-order provable in T that if $\langle mp, mr\rangle$ is T-well-ordered, it is T-well-founded. Now consider the CCMFs we construct for our CCG. Again, we must be able to prove that they are safe using theory T.

Definition 11.2.5. Given a semi-complete T-CCG, $\mathcal{G} = (C, E)$, $\langle mp, mr\rangle \in \mathit{Fun}^2$ that is T-well-founded, a T-expressible mp CCM annotation, m, and $\langle c, c'\rangle \in E$ such that $c = \langle f, \{r_1, \ldots, r_n\}, e\rangle$ and $c' = \langle f', \{r'_1, \ldots, r'_m\}, e'\rangle$, a CCMF, $\phi^c_{c'}$, for m is T-*compatible*, if, for all $s \in m(c), s' \in m(c')$, the following conditions hold:

- $(\phi^c_{c'}(s, s') =>) \Rightarrow (T \vdash$ `(implies (and` r_1 ... r_n $r'_1\sigma_e$ ... $r'_m\sigma_e$`)` `(`mr $s'\sigma_e$ s`)))`
- $(\phi^c_{c'}(s, s') =\geq) \Rightarrow$

$$(T \vdash \texttt{(implies (and } r_1 \ \ldots \ r_n \ r'_1\sigma_e \ \ldots \ r'_m\sigma_e\texttt{) (not (}mr \ s \ s'\sigma_e\texttt{))))}$$

Putting these concepts together, we obtain the notion of CCG admissibility.

Definition 11.2.6. A definitional axiom, d is *CCG admissible* in theory T if there exists a semi-complete T-CCG, $\mathcal{G} = (C, E)$, $\langle mp, mr\rangle \in \mathit{Fun}^2$ that is T-well-ordered and T-ordinal-bounded by some $v \in \mathit{Val}$, a T-expressible mp CCM annotation, m, and CCMF set $\Phi = \{\phi^c_{c'} \mid \langle c, c'\rangle \in E \ \ \wedge \ \ \phi^c_{c'}$ is T compatible$\}$, such that Φ is well-founded.

11.3 Compatibility With the ACL2 Logic

We show that CCG admissibility implies measure admissibility. Our proof relies on a result by Lee [64] which he uses to prove the existence of a ranking function when size-change termination analysis succeeds. We begin with a quick proof outline to give some intuition as to where we are heading.

In order to leverage Lee's result, we must formalize the connection between our concept of well-founded CCMFs and the concept of size-change graphs that satisfy the size-change termination condition (SCT). We begin by giving a quick overview of size-change graphs and the size-change termination condition, as well as a statement of the result from Lee's

paper. This result, while not explicitly giving a ranking function, implies the existence of one that can be proven decreasing for each step of computation using only information collected from the size-change graphs that satisfy SCT. This ranking function will map parameters to tuples of parameters and constants.

We then demonstrate how to convert CCMFs into size-change graphs in such a way that a set of CCMFs are well-founded if and only if the corresponding size-change graphs satisfy the size-change termination condition. Using this result, we transfer Lee's result about SCT to a result about CCMFs.

Using this result about well-founded CCMFs we will demonstrate the existence of a "per-context" measure —*i.e.*, an expression that is first-order provably decreasing from one context to the next in the CCG. This measure will map the parameters of the function of a context to lists of the CCMs of the context and contants.

Finally, we will show how to combine the "per-context" measures of all the contexts of a given function into a traditional ACL2 measure that will be provably decreasing across every function call. In fact, we will derive two such measures. One will be simpler, and will be applicable only when CCG admissibility is proven using only "per-context" CCMFs as described in Section 9.2.3.1, and one that will be more complex, but apply in all cases when the function definitions are CCG admissible.

11.3.1 Background: Size-Change Termination and Ranking Functions

We begin with a review of size-change termination and Lee's result about the existence of ranking functions.

Definition 11.3.1. Let $F \subset \mathit{Fun}$ be a finite set of function names. For each $f \in F$, we write $Par_f \subseteq \mathit{Var}^+$ to denote a set of distinct finite set of variables, called the *parameters* of f. A *size-change graph*, G, from $f \in F$ to $f' \in F$, denoted $G : f \to f'$ is a labeled bipartite directed graph, $G = (Par_f \cup Par_{f'}, E)$ such that $E \subseteq \{\langle x, \gamma, x' \rangle \mid x \in Par_f \quad \wedge \quad x' \in Par_{f'} \quad \wedge \quad \gamma \in \{>, \geq\}\}$, and $\neg(\langle x, >, x' \rangle \in E \quad \wedge \quad \langle x, \geq, x' \rangle \in E)$. We denote the fact that G contains the edge $\langle x, \gamma, x' \rangle$ by $x \xrightarrow{\gamma}_G x'$.

The full size change analysis takes as input a set of functions and constructs a size-change graph, $G : f \to f'$ for each recursive call to f' in f. The edges of G reflect the relationship between the values of the parameters of f and the values being passed to f' in the call. However, for our purposes we do not need to formalize the connection between the language semantics and size-change graphs.

We now define what it means for a set of size-change graphs to be size change terminating.

Definition 11.3.2. A sequence of size-change graphs, $G_1 : f_1 \to f_1', G_2 : f_2 \to f_2', \ldots$ is *control-flow legal* if, for all i, $f_i' = f_{i+1}$.

G satisfies *size-change termination (SCT)*, if for any control-flow legal sequence of size-change graphs, $G_1 : f_1 \to f_1', G_2 : f_2 \to f_2', \ldots$, there exists $i_0 \geq 1$ and a sequence $x_{i_0}, x_{i_0+1}, \ldots$, such that, for all $i \geq i_0$, $x_i \in Par_{f_i}$ and there exists $\gamma_i \in \{>, \geq\}$, and for infinitely many $i \geq i_0$, $\gamma_i = >$.

Note the similarity between size-change graphs and CCMFs, and between the SCT condition for size-change graphs and the well-foundedness condition for CCMFs. This is why we can use the SCT algorithm as the back end to our CCG analysis.

We now present definitions that culminate in the result from Lee's paper that is key to our argument that CCG admissibility implies measure admissibility.

Notation 11.3.1. *By $(i)_{\langle S,\prec\rangle}$, where $i \in \omega$, we mean the unique element, $v \in S$, such that $|\{u \in S \mid u \prec v\}| = i$ (note that this is a unique element since S is well-ordered by $\prec$). Let $Par_{f,n} = Par_f \cup \{(i)_{\langle S,\prec\rangle} \mid 0 \leq i \leq n\}$, and $Par_{f,n}^k$ denote k-tuples of the elements of $Par_{f,n}$.*

If $G : f \to f'$ is a size-change graph, $n \in \omega - \{0\}$, $z \in Par_{f,n}$, and $z' \in Par_{f',n}$, then we write $z' <_G z$ if $z' \in Par_{f'}$, $z \in Par_f$, and $z \xrightarrow{>}_G z'$, or if $z', z \in S$ and $z' \prec z$. We write $z' \leq_G z$ if $z' \in Par_f$, $z \in Par_{f'}$, and $z \xrightarrow{\geq}_G z'$, or if $z', z \in S$ and $z' \preceq z$, or if $z' = (0)_{\langle S,\prec\rangle}$ and $z \in Par_f$.

Definition 11.3.3. Given a size-change graph $G : f \to f'$, $\vec{z} = \langle z_i \rangle_{i=1}^k \in Par_{f,n}^k$, and $\vec{z}' = \langle z_i' \rangle_{i=1}^k \in Par_{f',n}^k$, we write $\vec{z}' <_G \vec{z}$ if $k > 0$ and one of the following conditions hold:

- $z'_1 <_G z_1$ or
- both of the following conditions hold
  - $z'_1 \leq_G z_1$ and
  - $\langle z_i \rangle_{i=2}^k <_G \langle z'_i \rangle_{i=2}^k$

Notice that $<_G$ defines a kind of static lexicographic order. The central result, paraphrased in the following theorem, is the main result of Lee's paper for our purposes [64].

Theorem 11.3.1. *Let $F \subset Fun$ be finite, and $\mathfrak{G}$ be a set of size-change graphs from functions in F to functions in F. Then if G satisfies SCT, there exists $k, n \in \omega$, and $\{S_f \subseteq (\mathcal{P}(Par_{f,n}^k) - \{\emptyset\}) \mid f \in Fun\}$, such that, for all $G : f \to f' \in \mathfrak{G}$ and $Z \in S_f$, there exists $z \in Z$, and $Z' \in S_{f'}$ such that, for all $z' \in Z'$, $z' <_G z$.*

11.3.2 CCMFs and Size-Change Graphs

In order to apply Theorem 11.3.1 to our CCG analysis, we must describe how to reduce the problem of determining if a CCG is well-founded to the problem of determining if a set of size-change graphs satisfy SCT. We begin by fixing the following values.

Notation 11.3.2. *Let T be a theory, and d be a definitional axiom. Let $\langle \texttt{mp}, \texttt{m<} \rangle \in Fun^2$ be T-well-ordered, and T-ordinal-bounded by $v \in Val$. Let $\mathcal{G} = (C, E)$ be a ruler semi-complete T-CCG for d, and m be a T-expressible mp CCM annotation. Finally, let $\{R_e \in Expr \mid e \in E\}$ and $\Phi = \{\phi^c_{c'} \mid \langle c, c' \rangle \in E\}$ be a set of CCMFs for $\mathcal{G}$ such that, for all $c = \langle f, R, e \rangle c' = \langle f', R', e' \rangle \in C$ such that $\langle c, c' \rangle \in E$, $\phi^c_{c'}(s, s') =>$ implies that $T \vdash$ `(implies` $R_{\langle c,c' \rangle}$ `(m<` $s' \sigma_e$ s`))` and $\phi^c_{c'}(s, s') =\geq$ implies that $T \vdash$ `(implies` $R_{\langle c,c' \rangle}$ `(not (m<` s $s' \sigma_e$`)))`.*

We denote $\{v \in Val \mid T \vdash$ `(mp` v`)`$\}$ as $\mathcal{M}$ and $\{\langle v, v' \rangle \mid T \vdash$ `(m<` v v'`)`$\}$ as $\prec_{\mathcal{M}}$.

Fix these values for the remainder of the section. Note that if all the $\phi^c_{c'} \in \Phi$ are T-compatible, we can choose $R_{\langle c,c' \rangle}$ to be the conjunction of the rulers of c and c' (with the call-substitution for c applied to the rulers of c'). Also, note that $\langle \mathcal{M}, \prec_{\mathcal{M}} \rangle$ is a well-ordered structure.

We now give a method for constructing size-change graphs from CCMFs.

Notation 11.3.3. *For each $c \in C$, let $f_c \in Fun$ denote a function name that does not appear in T. We denote the CCMs for c, $m(c)$, as $s^c_1, \ldots s^c_{|m(c)|}$, and Par_{f_c} as $\{x^c_1, \ldots x^c_{|m(c)|}\}$.*

Definition 11.3.4. Let $\phi^c_{c'} \in \Phi$. Then the size-change graph for $\phi^c_{c'}$, is $G : f_c \to f_{c'}$, such that, for all $x^c_i \in Par_{f_c}$ and $x^{c'}_j = inPar_{f_{c'}}$, $x^c_i \xrightarrow{\geq}_G x^{c'}_j$ iff $\phi^c_{c'}(s^c_i, s^{c'}_j) = \geq$, and there is an edge $x^c_i \xrightarrow{>}_G x^{c'}_j$ iff $\phi^c_{c'}(s^c_i, s^{c'}_j) = >$. We denote the set of size-change graphs for the CCMFs of Φ to be G_Φ.

Using this transformation, we get the following theorem.

Theorem 11.3.2. *Φ is well-founded if and only if G_Φ satisfies SCT.*

Proof. Follows directly from the definitions of the well-foundedness of CCMFs, SCT, and G_Φ. □

11.3.3 Constructing the Per-Context Measure

We now show how to use the previous result to construct a per-context measure.

Definition 11.3.5. For every $c \in C$, let $CCM_c = m(c)$, $CCM_{c,n} = CCM_c \cup \{(i)_{\langle \mathcal{M}, \prec_{\mathcal{M}} \rangle} \mid 1 \leq i \leq n\}$, and $CCM^k_{c,n} = \{\texttt{(list } e_1 \ \ldots \ e_k\texttt{)} \mid \forall 1 \leq i \leq k,\ e_i \in CCM_{c,n}\} \subseteq Expr$. Also, for each $c \in C$, let $\varphi_c : Par^k_{f_c,n} \to CCM^k_{c,n}$ be defined as $\varphi_c(\langle z_i \rangle^k_{i=1}) = \texttt{(list } e_1 \ \ldots \ e_k\texttt{)}$ where, for all $1 \leq i \leq k$, $e_i = z_i$ if $z_i \in \mathcal{M}$, and $e_i = s^c_i$ if $z_i = x^c_i$.

Figure 42 on the following page gives the definitions of `lmp` and `l<`, which recognize tuples of values that satisfy `mp` and compute the lexicographic ordering on tuples that are the same size, respectively. Tuples are stored as lists, which are tuples of values that satisfy `mp` if they are empty (representing 0-tuples), or if their first element satisfies `mp` and the rest of the list represents such a tuple. The ordering function, `l<` firsts tests if the tuples are of the same size (`len` returns the length of a list), and then calls `d<`, which returns true either if the first element of `x` is less than that of `y` by `m<`, or they are equal and the rest of `x` is less than the rest of `y`.

Given that $\langle \texttt{mp}, \texttt{m<} \rangle$ is T-well-ordered and T-ordinal-bound, it is the case that $\langle \texttt{lmp}, \texttt{l<} \rangle$ is also T-well-ordered. Also, both `m<` and `l<` are transitive on their respected domains.

```
(defun lmp (x)
  (if (endp x)
      (eq x nil)
    (and (mp (car x))
         (lmp (cdr x)))))

(defun d< (x y)
  (and (consp x)
       (or (m< (car x) (car y))
           (and (equal (car x) (car y))
                (d< (cdr x) (cdr y))))))

(defun l< (x y)
  (and (= (len x) (len y))
       (d< x y)))
```

Figure 42: Defining `mp` tuples in ACL2

We have verified these properties in ACL2 version 3.2.1. We note the following connection between $Par^k_{f_c,n}$ and `lmp`.

Lemma 11.3.1. *Let $c \in C$, $k, n \in \omega$, and $\vec{z} \in Par^k_{f_c,n}$. Then $T \vdash$* `(lmp` $\varphi_c(\vec{z})$`)`

Proof. Proof is by induction on k. If k is 0, then clearly `(lmp (list))` is a theorem, since `(list)` returns the empty list.

Otherwise, suppose that the lemma is true for $k-1$. Let $\vec{z} = \langle z_i \rangle_{i=1}^k$ and $\varphi_c(\vec{z}) =$`(list` $e_1 \ \ldots \ e_k$`)`. Then

$T \vdash$ `(lmp (list` $e_1 \ \ldots \ e_k$`))`

$\equiv$ { *Axioms of* `lmp`, `car`, `cdr`, `list` }

$T \vdash$ `(and (mp` e_1`) (lmp (list` $e_2 \ \ldots \ e_k$`)))`

$\equiv$ { *Boolean reasoning* }

$T \vdash$ `(mp` e_1`)` $\wedge$

$T \vdash$ `(lmp (list` $e_2 \ \ldots \ e_k$`))`

$\equiv$ { *Induction Hypothesis* }

$T \vdash$ `(mp` e_1`)`

At this point, there are two cases. One is that $e_1 = (i)_{\langle \mathcal{M}, \prec_{\mathcal{M}} \rangle}$ for some $1 \leq i \leq n$. In this case $T \vdash$ `(mp` e_1`)` by the definition of $\mathcal{M}$. Otherwise, $e_1 \in CCM_c = m(c)$. Since, m is a T-expressible `mp` CCM annotation, $T \vdash$ `(mp` e_1`)` by definition. $\square$

A similar connection exists between $<_G$ and `l<`.

In the first case, $z_1 \in Par_{f_c} = m(c)$, $z_1' \in Par_{f_{c'}}$, and $z_1 \xrightarrow{>}_G z_1'$. By the definition of G_Φ, this means that $\phi^c_{c'}(e_1, e_1') =>$. By the definition of $\phi^c_{c'}$ in Notation 11.3.2 on page 141, this means that $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(m<` $e_1'\sigma_e$ e_1`))`. Therefore, by the definition of $d <$, $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(d<` $\varphi_{c'}(\vec{z}')\sigma_e$ $\varphi_c(\vec{z})$`))`.

In the second case, $\langle z_i' \rangle_{i=2}^k <_{\langle c,c'\rangle} \langle z_i \rangle_{i=2}^k$ and one of the following three cases hold: either $z_1 \in Par_{f_c,n}$ and $z_1' = (0)_{\langle \mathcal{M}, \prec_{\mathcal{M}}\rangle}$, or $z_1, z_1' \in Par_{f_c,n}$ and $z_1 \xrightarrow{\geq}_G z_1'$, or $z_1 = z_1' \in \mathcal{M}$.

If $z_1 \in Par_{f_c,n}$ and $z_1' = (0)_{\langle \mathcal{M}, \prec_{\mathcal{M}}\rangle}$, then $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(not (m<` e_1 e_1'`)))`. Since $\langle$`mp, m<`$\rangle$ is T-well-ordered, this is equivalent to $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(or (equal` e_1' e_1`) (m<` e_1' e_1`)))`. In either case, $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(d<` $\varphi_{c'}(\vec{z}')\sigma_e$ $\varphi_c(\vec{z})$`))` by the definition of `d<` and the induction hypothesis.

If $z_1, z_1' \in Par_{f_c,n}$ and $z_1 \xrightarrow{\geq}_G z_1'$ then $\phi^c_{c'}(z_1, z_1') =\geq$. By the definition of $\phi^c_{c'}$ in Notation 11.3.2 on page 141, this means that $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(not (m<` $e_1'\sigma_e$ e_1`)))`. Therefore, by the same argument as in the previous case, $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(d<` $\varphi_{c'}(\vec{z}')\sigma_e$ $\varphi_c(\vec{z})$`))`.

Finally, if $z_1 = z_1' \in \mathcal{M}$. Then $e_1 = z_1$ and $e_1' = z_1'$. Then we have that $T \vdash$ `(equal` e_1 e_1'`)`. Therefore, $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(d<` $\varphi_{c'}(\vec{z}')\sigma_e$ $\varphi_c(\vec{z})$`))` by the definition of `d<`. □

Now we apply Lee's result (Theorem 11.3.1 on page 141) to the original CCG problem.

Lemma 11.3.3. *If Φ is well-founded, there exist $k, n \in \omega$ and $\{S_c \subseteq (\mathcal{P}(CCM^k_{c,n}) - \{\emptyset\} \mid c \in C\}$ such that, for all $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle, c' = \langle f', \{r_1', \ldots, r_m'\}, e'\rangle \in C$ such that $\langle c, c'\rangle \in E$ and $S \in S_c$, there exists $s \in S$ and $S' \in S_{c'}$ such that, for all $s' \in S'$, $T \vdash$* `(implies` $R_{\langle c,c'\rangle}$ `(l<` $s'\sigma_e$ s`))`.

Proof. If Φ is well-founded, then by Theorem 11.3.2 on page 142, G_Φ satisfies SCT. Therefore, by Theorem 11.3.1 on page 141, there exists $k, n \in \omega$, and $\{S_{f_c} \subseteq (\mathcal{P}(Par^k_{f,n}) - \{\emptyset\}) \mid c \in C\}$, such that, for all $G : f_c \to f_{c'}$ and $Z \in S_{f_c}$, there exists $z \in Z$, and $Z' \in S_{f_{c'}}$ such that, for all $z' \in Z'$, $z' <_G z$.

But if $z' <_G z$, then by Lem 11.3.2 on the preceding page, $T \vdash$ `(implies` $R_{\langle c,c'\rangle}$ `(l<` $\varphi_{c'}(z')\sigma_e$ $\varphi_c(z)$`)))`. Therefore, if we let $S_c = \{\{\varphi_c(z) \mid z \in Z\} \mid Z \in S_{f_c}\}$ for each $c \in C$,

```
(defun min-l< (lst)
  (cond ((endp lst) nil)
        ((endp (cdr lst)) (car lst))
        (t
         (let ((min (min-l< (cdr lst))))
           (if (l< (car lst) min)
               (car lst)
             min)))))

(defun max-l< (lst)
  (cond ((endp lst) nil)
        ((endp (cdr lst)) (car lst))
        (t
         (let ((max (max-l< (cdr lst))))
           (if (l< (car lst) max)
               max
             (car lst))))))

(defun max-l<-lst (lst)
  (if (endp lst)
      nil
    (cons (max-l< (car lst))
          (max-l<-lst (cdr lst)))))
```

Figure 43: `min-l<`, `max-l<`, and `max-l<-lst` functions.

then the theorem holds for these values. □

11.3.4 Measure Admissibility

Now we must consolidate these sets of sets of expressions into a single measure for each function. The first step in doing this is to create a single expression for each context that is provably decreasing along each edge of the CCG. To do this we use the functions in Figure 43. The first of these, `min-l<` takes a list and returns the minimum value in the list according to the ordering `l<`. The second, `max-l<`, is similar, but returns the maximum value of the list. Finally, `max-l<-lst` takes in a list of lists, $(l_1 \ \ldots \ l_n)$ and returns the list $(e_1 \ \ldots \ e_n)$ such that e_i is the maximum value in l_i by ordering `l<` for all $1 \leq i \leq n$.

For the remainder of the section, define $\mathcal{L}$ for each $c \in C$ to be $\{e \in \mathit{Expr} \mid T \vdash$ `(lmp` e`)`. As we have seen, $\mathcal{L}$ includes $CCM^k_{c,n}$ for all k. Further, let $\mathcal{L}_k$ be $\{e \in \mathcal{L} \mid T \vdash$ `(equal (len` e`)` k`)`. Also, given a set, $\mathit{listof}(S)$ be the set of lists whose elements are

members of S. We begin by showing that `max-l<` and `min-l<` preserve types.

Lemma 11.3.4. *If $e \in listof(\mathcal{L}_k)$ for some $k \in \omega$, then* `(max-l<` e`)` $\in \mathcal{L}_k$.

Proof. Proof is by induction over the length of e. □

Lemma 11.3.5. *If for some $k \in \omega$, $e \in listof(listof(\mathcal{L}_k))$, then* `(min-l< (max-l<-lst` e`))` $\in \mathcal{L}_k$.

Proof. Proof is by induction over the length of e. □

Next, we tie the functions of Figure 43 on the preceding page to the conditions of Lemma 11.3.3 on page 144 using the following four lemmas.

Lemma 11.3.6. *Let $\langle c, c' \rangle \in E$, $R \in Expr$, $s \in \mathcal{L}_k$ and non-empty $S' = \{s'_i \mid 1 \leq i \leq r\} \subseteq \mathcal{L}_k$. Then $\langle \forall s' \in S' :: T \vdash$* `(implies` R `(l<` $s'\sigma_e$ s`))`$\rangle$*, if and only if $T \vdash$* `(implies` R `(l<` `(max-l<` `(list` s'_1 ... s'_r`))`σ_e s`))`.

Proof. The proof is by a simple inductive argument on r. □

Lemma 11.3.7. *Let $\langle c, c' \rangle \in E$, $R \in Expr$, $s \in \mathcal{L}_k$ and non-empty $S' = \{s'_i \mid 1 \leq i \leq r\} \subseteq \mathcal{L}^k_{c'}$. Then $\langle \exists s' \in S' :: T \vdash$* `(implies` R `(l<` $s'\sigma_e$ s`))`$\rangle$*, if and only if $T \vdash$* `(implies` R `(l<` `(min-l<` `(list` $s'_1\sigma_e$ ... $s'_r\sigma_e$`))` s`))`.

Proof. The proof is by a simple inductive argument on r. □

Lemma 11.3.8. *Let $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle, c' = \langle f', \{r'_1, \ldots, r'_{m'}\}, e'\rangle \in C$ such that $\langle c, c' \rangle \in E$. Let $R \in Expr$, non-empty $S = \{s_i \mid 1 \leq i \leq r\} \subseteq\in \mathcal{L}_k$ and $s' \in \mathcal{L}^k_{c'}$ such that. Then $\langle \exists s \in S :: T \vdash$* `(implies` R `(l<` $s'\sigma_e$ s`))`$\rangle$*, if and only if $T \vdash$* `(implies` R `(l<` $s'\sigma_e$ `(max-l<` `(list` s_1 ... s_r`))))`.

Proof. The proof is by a simple inductive argument on r □

Lemma 11.3.9. *Let $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle, c' = \langle f', \{r'_1, \ldots, r'_{m'}\}, e'\rangle \in C$ such that $\langle c, c' \rangle \in E$. Let $R \in Expr$, non-empty $S = \{s_i \mid 1 \leq i \leq r\} \subseteq \mathcal{L}^k_c$ and $s' \in \mathcal{L}^k_{c'}$. Then $\langle \forall s \in S :: T \vdash$* `(implies` R `(l<` $s'\sigma_e$ s`))`$\rangle$*, if and only if $T \vdash$* `(implies` R `(l<` $s'\sigma_e$ `(min-l<` `(list` s_1 ... s_r`))))`.

Proof. The proof is by a simple inductive argument on r.

□

Lemma 11.3.10. *If Φ is well-founded, there exist $k \in \omega$ and $\{m_c \in \mathcal{L}_k \mid c \in C\}$ such that, for all $c \in C$, $T \vdash$* `(lmp` m_c`)` *and, for all $\langle c, c' \rangle \in E$, $T \vdash$* `(implies` $R_{\langle c,c' \rangle}$ `(l<` $m_{c'}\sigma_e$ m_c`))`

Proof. By Lemma 11.3.3 on page 144, there exist $k, n \in \omega$ and $\{S_c \subseteq (\mathcal{P}(CCM^k_{c,n}) - \{\emptyset\} \mid c \in C\}$ such that, for all $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle, c' = \langle f', \{r'_1, \ldots, r'_m\}, e'\rangle \in C$ such that $\langle c, c' \rangle \in E$ and $S \in S_c$, there exists $s \in S$ and $S' \in S_{c'}$ such that, for all $s' \in S'$, $T \vdash$ `(implies` $R_{\langle c,c' \rangle}$ `(l<` $s'\sigma_e$ s`))`.

Denote each S_c as $\{S^c_1, \ldots, S^c_{n_c}\}$ where for all $1 \leq i \leq n_c$, $S^c_i = \{s^c_{i,1}, \ldots, s^c_{i,n^c_i}\}$. Then for each $c \in C$, let $L_c =$`(list` L^c_1 $\ldots$ $L^c_{n_c}$`)` where for all $1 \leq i \leq n_c$, $L^c_i =$`(list` $s^c_{i,1}$ $\ldots$ s^c_{i,n^c_i}`)`. Finally, let $m_c =$`(min-l< (max-l<-lst` L_c`))`.

The first part of the theorem, that for all $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle \in C$, $T \vdash$ `(lmp` m_c`)` follows directly from Lemma 11.3.5 on the previous page.

Consider the second claim. Let $e = \langle c, c' \rangle \in E$. Then,

$\langle \forall S^c_i \in S_c :: \langle \exists s^c_{i,j} \in S^c_i, S^{c'}_a \in S_{c'} :: \langle \forall s^{c'}_{a,b} \in S^{c'}_a :: T \vdash$ `(implies` R_e `(l<` $s^{c'}_{a,b}\sigma_e$ $s^c_{i,j}$`))`$\rangle\rangle\rangle$

$\equiv$ { *Lem. 11.3.6* }

$\langle \forall S^c_i \in S_c :: \langle \exists s^c_{i,j} \in S^c_i, L^{c'}_a \in L_{c'} :: T \vdash$ `(implies` R_e `(l<` `(max-l<` $L^{c'}_a$`)`σ_e $s^c_{i,j}$`))`$\rangle\rangle$

$\equiv$ { *Lem. 11.3.8* }

$\langle \forall L^c_i \in L_c :: \langle \exists L^{c'}_a \in L_{c'} :: T \vdash$ `(implies` R_e `(l<` `(max-l<` $L^{c'}_a$`)`σ_e `(max-l<` L^c_i`)))`$\rangle\rangle$

$\equiv$ { *Lem. 11.3.7, Def. of* `max-l<-lst` }

$\langle \forall L^c_i :: T \vdash$ `(implies` R_e `(l<` `(min-l<` `(max-l<-lst` $L^{c'}_a\sigma_e$`))` `(max-l<` L^c_i`)))`$\rangle$

$\equiv$ { *Lem. 11.3.9, Def. of* `max-l<-lst` }

$T \vdash$ `(implies` R_e `(l<` `(min-l<` `(max-l<-lst` $L_{c'}$`))`σ_e`)` `(min-l<` `(max-l<-lst` L_c`)))`

$\equiv$ { *Def. of* m_c }

$T \vdash$ `(implies` R_e `(l<` $m_{c'}\sigma_e$ m_c`))`

□

So now we have a single expression for each context that provably decreases along every

edge of $\mathcal{G}$. Now we must combine these "context measures" into a single measure for each function. While the ultimate goal is to do this for T-compatible CCMFs, we first note the following, which will be useful when dealing with "per-context" CCMFs as discussed in Section 9.2.3.1.

Theorem 11.3.3. *Suppose that Φ is well-founded. For all $c \in C$, let m_c be as described in Lemma 11.3.10 on the preceding page. For each f defined in d, let $C_f = \{c_i^f \mid 1 \leq i \leq n_f\}$ be the contexts whose function is f, and $m_f =$* `(max-l< (list` $m_{c_1^f}$ `...` $m_{c_{n_f}^f}$ `))`.

Then if, for every $c = \langle f, \{r_1, \ldots, r_n\}, e\rangle, c' = \langle f', \{R'\}, e'\rangle \in C$ such that e is a call to f', $\langle c, c'\rangle \in E$ and $R_{\langle c,c'\rangle} =$ `(and` r_1 `...` r_n`)`*, then d is measure admissible by the measure that maps each f defined in d to m_f.*

Proof. We have two obligations.

The first is to show that $T \vdash m_f$. But this follows directly from Lemma 11.3.4 on page 146 and the properties of each m_c as given in Lemma 11.3.10 on the preceding page.

Our second obligation is to show that, for all f defined in d and $p \in Pos(e^f)$ such that $e = e^f|_p$ is a call to function f' and $R =$`(and` r_1 `...` r_m`)` where $rulers(e^f, p) = \{r_1, \ldots, r_m\}$, it is the case that $T \vdash$ `(implies` R `(m<` $m_{f'}\sigma_e$ m_f`))`. Since $\mathcal{G}$ is ruler semi-complete, there exists $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle \in C_f$. Also, by the hypotheses of our theorem, for all $c' \in C_{f'}$, $\langle c, c'\rangle \in E$. Therefore, by Lemma 11.3.4 on page 146 and the hypotheses of this theorem, $T \vdash$ `(implies` R `(m<` m_c $m_{c'}$`))` for every $c' \in C_{f'}$. Therefore, by Lemmas 11.3.6 on page 146 and 11.3.9 on page 146, $T \vdash$ `(implies` R `(m<` $m_{f'}\sigma_e$ m_f`))`. □

For the more general case, where all the $\phi_{c'}^c$ are T-compatible, the proof of measure admissibility is a bit more difficult. To construct the measure, we take the maximum of the context measures whose contexts' conditions are satisfied. In ACL2, this expression is of the form

```
(let* ((m (list v_0))
       (m (if R_1 (cons (cons (1)_⟨M,≺M⟩ m_c1) m) m))
       ...
       (m (if R_r (cons (cons (1)_⟨M,≺M⟩ m_cn) m) m)))
    (max-l< m)))
```

where v_0 is the minimal element of the $k+1$ tuples of $\mathcal{M}$, the R_i are the conjunction of the conditions of each context of the function, and the m_i are the context measures of the contexts of the given function. The inclusion of v_0 in the list is to ensure that `m` is non-empty even in the base case (when none of the R_i are true). The m_i are all prefixed with the second-to-the-least element of $\mathcal{M}$ to ensure that the measure decreases when the measure for the next step is v_0. Note that this expression potentially violates one of the requirements of measures. The problem is that measures should be expressible over T, which does not include functions of d, while this measure may mention the functions of d, which might appear in the callsite rulers. After much discussion with one of the developers of ACL2, we determined that this requirement is not necessary to maintain soundness, and therefore can be loosened to allow calls to the functions of d. We may therefore proceed to show that the original functions defined in d are measure admissible by this measure. We begin by showing that the measure is always provably recognized by `mp`.

Lemma 11.3.11. *Let $c_1, \ldots, c_r \in C$, and $e_1, e_2, \ldots, e_r \in Expr$, $\langle \forall 1 \leq i \leq r :: e_i \in \mathcal{L}_k \rangle$. Further, let $\{r_{i,1}, \ldots, r_{i,m_i}\}$ be the conditions of c_i, and $R_i =$`(and` $r_{i,1}$ ... r_{i,m_i}`)`. Finally, let $v_0 \in Val$ is the list of length $k+1$ whose elements are all $(0)_{\mathcal{M},\prec_{\mathcal{M}}}$. Then the following is a theorem of T:*

```
(lmp (let* ((m (list v0))
            (m (if R1 (cons (cons (1)⟨M,≺M⟩ e1) m) m))
            ...
            (m (if Rr (cons (cons (1)⟨M,≺M⟩ er) m) m)))
       (max-l< m))
```

Proof. Let e_i' denote `(cons` $(1)_{\langle \mathcal{M},\prec_{\mathcal{M}}\rangle}$ e_i`)` for each $1 \leq i \leq r$. Then, using the previous result, note the following.

$T \vdash$ `(lmp (max-l< (if` R_i `(cons` e_i' m`) `m`)))`

$\equiv$ { *Boolean Reasoning* }

$T \vdash$ `(implies` R_i `(lmp (max-l< (cons` e_i' m`))))` $\wedge$

$T \vdash$ `(implies (not` R_i`) (lmp (max-l<` m`)))`

$\equiv$ { *Def. of* `max-l<`, *Axioms of* `car`, `cdr` }

$T \vdash$ `(implies` R_i `(lmp (if (l<` e_i' `(max-l<` m`)) ` m e_i'`)))` $\wedge$

$T \vdash$ `(implies (not` R_i`) (lmp (max-l<` m`)))`

$\equiv$ { *Boolean Reasoning* }

$T \vdash$ `(implies (and` R_i `(not (l<` e_i' `(max-l<` m`))) (lmp` e_i'`)))` $\wedge$

$T \vdash$ `(implies (and` R_i `(l<` e_i' `(max-l<` m`))) (lmp` m`)))` $\wedge$

$T \vdash$ `(implies (not` R_i`) (lmp (max-l<` m`)))`

$\Leftarrow$ { *Boolean Reasoning, Def. of* `l<`, e_i' }

$T \vdash$ `(implies (and` R_i `(not (l<` e_i' `(max-l<` m`))) (lmp` e_i`)))` $\wedge$

$T \vdash$ `(lmp` m`))`

$\equiv$ { *Def. of* $\mathcal{L}$ }

$T \vdash$ `(lmp` m`)`

Now on to the main result. For the sake of brevity, let $b_i =$`(if` R_i `(cons (cons` $(1)_{\langle \mathcal{M}, \prec_{\mathcal{M}} \rangle}$ e_i`) m))` for $1 \leq i \leq r$. The proof is by induction on r.

If $r = 0$, this is trivially true by the definitions of e_0 and Lemma 11.3.4 on page 146.

Otherwise, let $m' =$`(let* ((m (list` e_0`)) (m` b_1`) ... (m` b_{r-1}`)) m)`. Then

$T \vdash$ `(lmp (let* ((m (list` e_0`)) (m` b_1`) ... (m` b_r`)) (max-l< m)))`

$\equiv$ { *Language Semantics* }

$$T \vdash \left(\begin{array}{l} \texttt{(lmp (let* ((m (list } e_0\texttt{)) (m } b_1\texttt{) ... (m } b_{r-1}\texttt{))} \\ \qquad\qquad \texttt{(max-l< (if } R_r \texttt{ (cons (cons } (1)_{\langle \mathcal{M}, \prec_{\mathcal{M}} \rangle} \texttt{ } e_r\texttt{) m)))))} \end{array} \right)$$

$\equiv$ { *Language Semantics* }

$T \vdash$ `(lmp (max-l< (if` R_r `(cons (cons` $(1)_{\langle \mathcal{M}, \prec_{\mathcal{M}} \rangle}$ e_r`)` m'`)` m'`)))`

$\Leftarrow$ { *Previous Result* }

$T \vdash$ `(lmp (max-l<` m'`))`

$\equiv$ { *Language Semantics* }

$T \vdash$ `(lmp (let* ((m (list` e_0`)) (m` b_1`) ... (m` b_{r-1}`)) (max-l< m)))`

$\equiv$ { *Induction Hypothesis* }

true

□

Next, we prove the following lemma which will be helpful in proving that the measure decreases across each recursive call.

Lemma 11.3.12. *Let $c = \langle f, \{r_1, \ldots, r_n\}, e\rangle$, where e is a call to f', and let $C_{f'} = \{c_i = \langle f', \{r_1^i, \ldots r_{n_i}^i\}, e_i\rangle | 1 \leq i \leq m\}$ be a subset of the contexts of C whose function is f'. Also, let $R =$(and r_1 ... r_n) and for all $1 \leq i \leq m$, let $R_i =$(and r_1^i ... $r_{n_i}^i$). Finally, let $m_c \in \mathcal{L}_k$ and for all $1 \leq i \leq m$, $m_{c_i} \in \mathcal{L}_k$ such that $T \vdash$(implies (and R $R_i\sigma_e$) (l< m_{c_i} m_c)) for all $1 \leq i \leq m$. Then the following is a theorem of T:*

```
(implies R
         (l< (let ((m (list v_0))
                   (m (if R_1 (cons (cons (1)_⟨M,≺_M⟩ m_{c_1}) m) m))
                   ...
                   (m (if R_m (cons (cons (1)_⟨M,≺_M⟩ m_{c_m}) m) m)))
              (max-l< m))σ_e
             (cons (1)_⟨M,≺_M⟩ m_c))
```

where $v_0 \in Val$ is the list of length $k+1$ whose elements are all $(0)_{\mathcal{M},\prec_{\mathcal{M}}}$.

Proof. The proof is by induction on m. If $m = 0$, then this is equivalent to $T \vdash$(implies R (l< v_0 (cons $(1)_{\langle\mathcal{M},\prec_{\mathcal{M}}\rangle}$ m_c))) which is clearly true since $T \vdash$(m< $(0)_{\langle\mathcal{M},\prec_{\mathcal{M}}\rangle}$ $(1)_{\langle\mathcal{M},\prec_{\mathcal{M}}\rangle}$).

Now suppose that $m > 0$. Let $m'_c =$(cons $(1)_{\langle\mathcal{M},\prec_{\mathcal{M}}\rangle}$ m_c) and for $1 \leq i \leq m$, let $m'_{c_i} =$(cons $(1)_{\langle\mathcal{M},\prec_{\mathcal{M}}\rangle}$ m_{c_i}) For $1 \leq i \leq m$, let $b_i =$(if R_i (cons m'_{c_i} m) m). Also, let $s =$(let ((m (list v_0)) (m b_1) ... (m b_{m-1})) m). Then

$$T \vdash \left(\begin{array}{l} \texttt{(implies } R \texttt{ (l< (let ((m (list } v_0\texttt{))) (m } b_1\texttt{) ... (m } b_{m-1}\texttt{))} \\ \qquad\qquad\qquad\qquad \texttt{(max-l< m))} \\ \qquad\qquad\qquad\quad m'_c\texttt{))} \end{array} \right)$$

$\equiv$ { *Language Semantics* }

$T \vdash$ (implies R (l< (max-l< (if R_m (cons m'_{c_m} s) s))σ_e m'_c))

$\equiv$ { *Def. of substitution* }

$T \vdash$ (implies R (l< (max-l< (if $R_m\sigma_e$ (cons $m'_{c_m}\sigma_e$ $s\sigma_e$) $s\sigma_e$)) m'_c))

$\equiv$ { *Boolean Reasoning* }

$$\left(\begin{array}{l} T \vdash \texttt{(implies (and } R\ R_m\sigma_e\texttt{) (l< (max-l< (cons } m'_{c_m}\sigma_e\ s\sigma_e\texttt{)) } m'_c\texttt{))} \wedge \\ T \vdash \texttt{(implies (and } R\texttt{ (not } R_m\sigma_e\texttt{)) (l< (max-l< } s\sigma_e\texttt{) } m'_c\texttt{))} \end{array} \right)$$

$\equiv$ { *Def. of* `max-l<`*, Boolean Reasoning* }

$$\left(\begin{array}{l} T \vdash \texttt{(implies (and } R\ R_m\sigma_e\ \texttt{(l<}\ m'_{c_m}\sigma_e\ \texttt{(max-l<}\ s\sigma_e\texttt{)))}\ \texttt{(l<}\ m'_{c_m}\sigma_e\ m'_c\texttt{))} \wedge \\ T \vdash \left(\begin{array}{l} \texttt{(implies (and } R\ R_m\sigma_e\ \texttt{(l<}\ m'_{c_m}\sigma_e\ \texttt{(max-l<}\ s\sigma_e\texttt{)))} \\ \qquad\qquad \texttt{(l<}\ \texttt{(max-l<}\ s\sigma_e\texttt{)}\ m'_c\texttt{))} \end{array}\right) \wedge \\ T \vdash \texttt{(implies (and } R\ \texttt{(not } R_m\sigma_e\texttt{))}\ \texttt{(l<}\ \texttt{(max-l<}\ s\sigma_e\texttt{)}\ m'_c\texttt{))} \end{array}\right)$$

$\equiv$ { *Induction Hypothesis* }

$T \vdash \texttt{(implies (and } R\ R_m\sigma_e\ \texttt{(l<}\ m'_{c_m}\sigma_e\ \texttt{(max-l<}\ s\sigma_e\texttt{)))}\ \texttt{(l<}\ m'_{c_m}\sigma_e\ m'_c\texttt{))}$

$\equiv$ { *Def. of* `l<`*,* $m'_c\ m'_{c_m}$ }

$T \vdash \texttt{(implies (and } R\ R_m\sigma_e\ \texttt{(l<}\ m_{c_m}\sigma_e\ \texttt{(max-l<}\ s\sigma_e\texttt{)))}\ \texttt{(l<}\ m_{c_m}\sigma_e\ m'_c\texttt{))}$

$\equiv$ { *Initial Hypothesis* }

true

□

Finally, we obtain the main result.

Theorem 11.3.4. *If every* $\phi^c_{c'} \in \Phi$ *is* T*-compatible and* Φ *is well-founded, then* d *is measure admissible.*

Proof. By Lemma 11.3.10 on page 147, there exists $k \in \omega$ and $\{m_c \in \mathcal{L}_k | c \in C\}$ such that, for all $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle \in C$, $T \vdash$ `(implies (and` $r_1\ \ldots\ r_m$`) (lmp` m_c`))` and, for all $c = \langle f, \{r_1, \ldots, r_m\}, e\rangle, c' = \langle f', \{r'_1, \ldots, r'_m\}, e'\rangle \in C$ such that $\langle c, c'\rangle \in E$, the following is a theorem of T:

```
(implies (and r_1 ... r_m r'_1σ_e ... r'_m'σ_e)
         (l< m_c'σ_e m_c))
```

Let F be the names of the functions defined in d, and for each $f \in F$, let $C_f = \{c^f_i = \langle f, \{r^f_{i,1}, \ldots, r^f_{i,n^f_i}\}, e^f_i\rangle \mid 1 \leq i \leq |C_f|\}$ be the contexts of C whose function is f. For all $f \in F$ and $1 \leq i \leq |C_f|$, let $R^f_i =$`(and` $r^f_{i,1}\ \ldots\ r^f_{i,n^f_i}$`)` and $m'_{c^f_i} =$`(cons` $(1)_{\langle \mathcal{M}, \prec_{\mathcal{M}}\rangle}\ m_{c^f_i}$`)`. Also, let $v_0 \in \mathit{Val}$ is the list of length $k+1$ whose elements are all $(0)_{\mathcal{M},\prec_{\mathcal{M}}}$. Then for each $f \in F$, let m_f be the following expression.

```
(let ((m (list v_0))
      (m (if R^f_1 (cons m'_{c^f_1} m) m))
```

```
      ...
      (m (if R_n^f (cons m'_{c_n^f} m) m)))
  (max-l< m))
```

We will show that d is measure admissible by the measure mapping each $f \in F$ to m_f and $\langle \texttt{lmp}, \texttt{l<} \rangle$.

By Lemma 11.3.11 on page 149, $T \vdash \texttt{(lmp } m_f\texttt{)}$. So, our only remaining obligation is to show that the measure decreases for every recursive call. Let $f \in F$, $p \in Pos(e^f)$ such that $e = e^f|_p$ is a function call, and let $R = \texttt{(and } r_1 \ \ldots \ r_n\texttt{)}$ where $rulers(e^f, p) = \{r_1, \ldots, r_n\}$. But note that, by construction, there exists $c_i^f = \langle f, \{r_{i,1}^f, \ldots, r_{i,n_i^f}^f\}, e_i^f \rangle \in C_f$. Such that $rulers(e^f, p) = \{r_{i,1}^f, \ldots, r_{i,n_i^f}^f\}$ and $e = e_i^f$. Therefore $R = R_i^f$. Let f' be the function called by e. So, we need to prove that $T \vdash \texttt{(implies } R \ \texttt{(l<} \ m_{f'}\sigma_e \ m_f\texttt{))}$. We do this by proving the stronger result that, for every $j \geq i$, $T \vdash \texttt{(implies } R_i^f \ \texttt{(l<} \ m_{f'}\sigma_{e_i^f} \ \texttt{(max-l<} \ \texttt{(if } R_j^f \ \texttt{(cons } m'_{c_j^f} \ m'\texttt{)} \ m'\texttt{))))}$, where m' is the following expression:

```
(let ((m (list v_0))
      (m (if R_1^f (cons m'_{c_1^f} m) m))
      ...
      (m (if R_{j-1}^f (cons m_{c_{j-1}^f} m) m)))
    (max-l< m))
```

We prove this by induction on $j - i$. If $j = i$, then

$T \vdash \texttt{(implies } R_i^f \ \texttt{(l<} \ m_{f'}\sigma_{e_i^f} \ \texttt{(max-l<} \ \texttt{(if } R_j^f \ \texttt{(cons } m'_{c_j^f} \ m'\texttt{)} \ m'\texttt{))))}$

$\equiv$ { $i = j$, *Language Semantics* }

$T \vdash \texttt{(implies } R_i^f \ \texttt{(l<} \ m_{f'}\sigma_{e_i^f} \ \texttt{(max-l<} \ \texttt{(cons } m'_{c_j^f} \ m'\texttt{))))}$

$\equiv$ { *Def. of* `max-l<`, *Boolean Reasoning* }

$$\left(\begin{array}{l} T \vdash \texttt{(implies (and } R_i^f \ \texttt{(not (l<} \ m'_{c_j^f} \ \texttt{(max-l<} \ m'\texttt{))))} \ \texttt{(l<} \ m_{f'}\sigma_{e_i^f} \ m'_{c_j^f}\texttt{))} \ \wedge \\ T \vdash \texttt{(implies (and } R_i^f \ \texttt{(l<} \ m'_{c_j^f} \ \texttt{(max-l<} \ m'\texttt{)))} \ \texttt{(l<} \ m_{f'}\sigma_{e_i^f} \ \texttt{(max-l<} \ m'\texttt{)))} \end{array} \right)$$

$\equiv$ { *Lem. 11.3.12, Transitivity of* `l<` }

true

Now suppose that $j > i$. Then,

$T \vdash \texttt{(implies } R_i^f \ \texttt{(l<} \ m_{f'}\sigma_{e_i^f} \ \texttt{(max-l<} \ \texttt{(if } R_j^f \ \texttt{(cons } m'_{c_j^f} \ m'\texttt{)} \ m'\texttt{))))}$

$\equiv$ { *Boolean Reasoning* }

$$\left(\begin{array}{l} T \vdash \texttt{(implies (and } R_i^f \texttt{ (not } R_j^f\texttt{)) (l<} \ m_{f'}\sigma_{e_i^f} \texttt{ (max-l<} \ m'\texttt{))) } \wedge \\ T \vdash \texttt{(implies (and } R_i^f \ R_j^f\texttt{) (l<} \ m_{f'}\sigma_{e_i^f} \texttt{ (max-l< (cons} \ m'_{c_j^f} \ m'\texttt{)))) } \end{array} \right)$$

$\equiv$ { *Induction Hypothesis, Language Semantics* }

$T \vdash \texttt{(implies (and } R_i^f \ R_j^f\texttt{) (l<} \ m_{f'}\sigma_{e_i^f} \texttt{ (max-l< (cons} \ m'_{c_j^f} \ m'\texttt{))))}$

$\equiv$ { *Def. of* `max-l<`*, Boolean Reasoning* }

$$\left(\begin{array}{l} T \vdash \texttt{(implies (and } R_i^f \texttt{ (not (l<} \ m'_{c_j^f} \texttt{ (max-l<} \ m'\texttt{)))) (l<} \ m_{f'}\sigma_{e_i^f} \ m'_{c_j^f}\texttt{)) } \wedge \\ T \vdash \texttt{(implies (and } R_i^f \texttt{ (l<} \ m'_{c_j^f} \texttt{ (max-l<} \ m'\texttt{))) (l<} \ m_{f'}\sigma_{e_i^f} \texttt{ (max-l<} \ m'\texttt{))) } \end{array} \right)$$

$\equiv$ { *Inductive Hypothesis, Language Semantics, Transitivity of* `l<` }

true

□

This gives us the result we have been after.

Corollary 11.3.1. *If d is CCG admissible, then it is measure admissible.*

11.4 Measured Subsets, Books, and Encapsulation

In this section, we explore the challenges of integrating our analysis with the theorem prover.

The first issue involves the concept of *measured subsets* of function formals. In traditional measure-based termination proofs in ACL2, this refers to the subset of the function formals that appear in the measure used to prove measure admissibility. This information is used by the theorem prover in its heuristics for choosing induction schemes. While a complete discussion of these heuristics is beyond the scope of this dissertation, an example will help illustrate the role that measured subsets play.

Consider the ACL2 events in Figure 44 on the next page. Here, `in` is a function that returns a non-nil value if and only if `a` is in list `b`. It is easily shown to be measure admissible using the measure `(acl2-count b)`. The `del` function removes `a` from list `b`. It is measure admissible by the same measure. The function `perm` returns true if list `x` is a permutation of list `y`, *i.e.*, every element of `x` appears in list `y` the same number of times. It is also

```
(defun in (a b)
  (cond ((atom b) nil)
        ((equal a (car b)) t)
        (t (in a (cdr b)))))

(defun del (a b)
  (cond ((atom b) nil)
        ((equal a (car b)) (cdr b))
        (t (cons (car b) (del a (cdr b))))))

(defun perm (x y)
  (cond ((atom x) (atom y))
        (t (and (in (car x) y)
                (perm (cdr x)
                      (del (car x) y))))))

 (defthm perm-reflexive
         (perm x x))
```

Figure 44: Definition of permutation predicate in ACL2

easily proven terminating, using measure `(acl2-count x)`. The measured subset for `perm` is therefore {`x`}, since this is the only variable that appears in the measure.

Finally, consider the theorem `perm-reflexive`, which states that any element, `x` is a permutation of itself. This theorem is proven automatically in ACL2 by using an induction scheme based on perm:

```
(and (implies (and (not (atom x))
                   (not (in (car x) x)))
              (perm x x))
     (implies (and (not (atom x))
                   (in (car x) x)
                   (perm (cdr x) (cdr x)))
              (perm x x))
     (implies (atom x) (perm x x)))
```

If we can prove all three of the formulas in this conjunction, then the original formula is proved by the principle of induction. Now suppose that, instead of {`x`}, the measured subset for `perm` were {`x`, `y`}. In this case, ACL2 cannot find an appropriate induction scheme, and therefore cannot automatically prove `perm-reflexive`. The problem is that ACL2 believes that we need both `x` and `y` to justify the induction scheme for `perm`, but such an induction

scheme cannot be instantiated using just the one variable, `x`, that appears in `(perm x x)`.

In order to integrate our CCG analysis with the ACL2 theorem prover, we needed to provide measured subsets in much the same way that ACL2 does. The requirement is that there must be a measure that can be built using only the formals in the measured subset that can be used to demonstrate measure admissibility of the definitional axioms. It is always sound to simply return the entire list of formals as the measured subset. However, as we have seen, this can adversely affect theorem prover performance. In order to come up with a "reasonable" measured subset, we turn to Theorem 11.3.3 on page 148 as well as Theorem 11.3.4 on page 152. These were the two theorems used to prove that CCG admissibility implies measure admissibility. Recall that this was accomplished by demonstrating that a measure exists that can be used to prove measure admissibility. Theorem 11.3.3 on page 148 does so when the CCG is proven well-founded using per-context CCMFs. Theorem 11.3.4 on page 152 does so for the general case. The solution, then, is to return the formals mentioned in these measures.

For the general case, this includes all the measures mentioned in the rulers for the recursive calls, plus all the CCMs needed to prove termination. For the per-context case, all we need are the CCMS that were needed to prove termination. While our heuristics for choosing CCMs result in the appearance of all of the formals in the CCMs for each function, there are documented algorithms for removing CCMs that are not useful for the well-foundedness proof [8].

In addition, our hierarchical algorithm helps keep unnecessary information about decreasing and non-increasing CCMs to a minimum. Consider, for example, the `perm` function. Using the full power of the CCG analysis, we discover that both `x` and `y` decrease with every step. However, in the hierarchical method, we realize that `x` decreases using only built-in clauses. This is enough to prove termination, and therefore we can return {`x`} as our measured subset.

But this causes another difficulty, related to ACL2 *events*. An event is an expression in ACL2 that changes the *logical world* which is the database ACL2 uses to keep track of information regarding the the theory and theorem prover settings. Examples of events

include definitional axioms and `defthm`, which defines a new theorem to be added to the theory. Other events include `encapsulate` and `include-book`, which are the sources of the aforementioned difficulties.

Encapsulation is a feature in ACL2 that, among other things, allows users to hide lemmas. For example, it is commonly the case that a lemma, `lem`, is needed to prove a theorem, `thm`, but `lem` is not useful in general. In fact, in some cases, `lem` may actually slow down or otherwise cause problems in future proofs. In such a case, `lem` and `thm` can be placed inside an `encapsulate` event and `lem` can be declared local to the encapsulation. Then `lem` does not exist in the theory resulting from the admission of the encapsulation form, while the main goal, `thm`, is. The way this is implemented in the theorem prover is that ACL2 makes two passes over the encapsulated code. On the first pass, all proofs are executed to verify the soundness of the encapsulated events. On the second pass, local events are skipped, but non-local events are admitted into the logical world without proof.

Books work in a similar way, but at the file level. Users organize theorems, definitions, and other events into books. These books can then be certified using ACL2, which involves verifying all the necessary proofs. They can then be *included*, or reloaded without repeating the proofs. Again, users can declare lemmas or other events local to a book, in which case they will not appear in the theory resulting from loading the book in the future.

For measure admissibility, these features do not cause a problem. Measures are either provided by the user, or guessed using a static analysis that requires no proofs. In either case, once the measure admissibility obligations are proven in the first pass, the proofs can be skipped in the second pass, and the measured subset can be computed using the measure provided or guessed by ACL2 when the function is admitted. CCG admissibility, however, presents a unique challenge, in that prover queries are necessary to compute the measured subset. Therefore, we cannot skip the proofs and still provide this information to the prover.

A first pass at a solution would be to ignore ACL2's directive to skip proofs on the second pass, and simply repeat all the proofs to compute the measured subset. However, there is a problem with this solution that is caused by local events. Because of this feature, the theory in which termination is proved on the first pass may be different than that of the

second pass. For example, suppose some local lemma, `lem` in a book is critical in order to prove CCG admissibility for some non-local function, `f` in the book. Then when including the book, `lem` would be skipped, and the proof of the termination of `f` would fail.

The solution to this problem is provided by a new ACL2 feature called `make-event`. The idea behind this feature is that it allows users to compute events. That is, based on the current environment, a new event is made. An important feature of `make-event` is that the new event is computed once and then saved. Thus if ACL2 makes a second pass over the `make-event` in a different environment, the same new event is created. This alleviates the problem caused by the two pass system employed by ACL2 for encapsulation and books.

To use this feature to our advantage, we alter ACL2's built in definitional utility, `defun`, so that it can be told explicitly which CCMs are important to the termination analysis. Then, when given a definition without this hint, we run the CCG-based analysis, compute the relevant CCMs, and use `make-event` to create a new `defun` in which the these CCMs are explicitly given as a hint. Then, when loading a book or making a second pass over an encapsulation, ACL2 can forgo the CCG analysis and use the CCMs given to calculate the measured subset for the function. More information on `make-event` may be found in the ACL2 documentation [57] starting with Version 3.0.

11.5 Summary

In this chapter we have discussed the challenges of integrating the CCG termination analysis into the ACL2 logic and theorem prover. We introduced the notion of CCG admissibility, which provides logical conditions under which the CCG analysis produces termination proofs that can, in theory, be verified by ACL2 using measure admissibility. This was proven in detail, culminating in the result that CCG admissibility implies measure admissibility. Therefore, the full integration of the CCG analysis into ACL2 would not affect the soundness or power of ACL2's logic, even while improving the theorem prover's ability to automatically prove termination.

We have also discussed the engineering concerns that arise when integrating the CCG termination analysis with ACL2's theorem prover. These revolve around the notion of

measured subsets, which naturally derive from ACL2's measured admissibility analysis, but which are more challenging to derive from the CCG analysis. This leads to further challenges brought about by ACL2's treatment of encapsulation and book certification. We showed how to use ACL2's new `make-event` mechanism to overcome these challenges to provide sound and useful measured subset information to the theorem prover.

PART IV

Future Work and Conclusions

CHAPTER XII

FUTURE WORK

We discuss possible avenues of future work.

One direction would be to extend our ordinal arithmetic algorithms to notations over larger ordinals than ε_0. For example, it has been shown that the ordinal Γ_0, which is much larger than ε_0, is needed to prove termination for some term rewriting systems [34]. Despite its magnitude, Γ_0 is still countable and there exists a well-known notation for representing the ordinals less than Γ_0 [40]. By extending our algorithms to Γ_0, and integrating this ordinal representation into ACL2, we would strengthen ACL2's logic, allowing it to reason about the termination of systems currently beyond its capabilities.

A second future direction relating to the ordinal arithmetic work is to implement new arithmetic algorithms for ε_0. These could include ordinal division and logarithms. Also, there exist operations known as "natural addition" and "natural multiplication" that are distinct from the standard addition and multiplication operations presented in this dissertation, and which enjoy the properties of commutativity, associativity, and distributivity [51]. Because of these properties, these operations may be more intuitive for people who are new to ordinal reasoning.

Given the power of our ordinal arithmetic library, another interesting future direction is to integrate this work with existing tools for guessing ranking functions or proving termination, and using ACL2 to mechanically verify the termination proofs. The result would be a ranking function generator that can be highly trusted and used for the most critical termination and liveness proofs.

Several interesting questions prompted by the CCG analysis experiments in Chapter 10 suggest interesting future directions.

For example, a significant number of the problems that failed when disabling the surrounding theory could easily be solved by including a basic arithmetic or data structures

library. However, including such libraries was not possible in our experiments due to theory conflicts. There has been an ongoing conversation in the ACL2 community about including these basic results in ACL2's ground-zero theory. Currently, few of these results are included, and users must either design their own theory or choose from multiple existing libraries to use such reasoning. The result is that different libraries use different theories, causing incompatibilities between the books. Possible solutions to this problem are to include such results in the ground-zero theory, or to officially advocate the use of one particular library for each of these basic theories. Another option would be to develop a mechanism of "reflexive" book loading. In such a system, ACL2 would check if a library for reasoning about a given topic is already included. If it is, ACL2 does nothing. If not, ACL2 includes a default library requested by the user.

Another class of problems that the CCG analysis had problems with are functions with similar looping behaviors that all require the same set of CCMs to prove termination, which the CCG analysis is not able to guess with existing heuristics. In such a situation, it would be useful to provide the user with a mechanism for creating their own CCM guessing heuristics. This would allow the user to provide a one-time hint to the CCG analysis that could be used in multiple future proof attempts to automatically prove termination.

Another useful interface enhancement for the CCG analysis would be a mechanism for reporting failed termination proofs for which the CCG analysis still made some progress. For example, if a program has multiple looping behaviors, and the CCG analysis is able to show that all but one of the loops will always terminate, it could use this loop to create a new function definition that mimics the behavior of this single loop and present it to the user as a simplified version of the termination proof whose solution would imply the termination of the original function. This would enhance the interactive nature of the CCG analysis, and allow it to help users prove termination even when it cannot automatically complete such a proof.

Longer-term projects related to the CCG analysis include adapting it to other domains such as higher order functional languages and imperative languages. In the domain of high order languages, Haskell would be an interesting target for the CCG analysis, given

that it is purely-functional, like our current domain. Interesting challenges in this domain include reasoning about high order functions, and reasoning about strongly typed programs. An imperative language of interest to us is Java. Since there is already an almost-complete model of the JVM implemented in ACL2 [66], we could continue to leverage ACL2's theorem proving support in our implementation and immediately focus on the challenges of reasoning about destructive updates and object oriented code.

CHAPTER XIII

CONCLUSION

The purpose of this dissertation has been to defend the following thesis statement:

> *A highly automatic, general, interactive, and efficient termination analysis is possible for feature-rich, first-order, purely functional programming languages.*

We have done this by developing novel and general techniques for mechanically reasoning about termination in this domain. We have demonstrated these techniques to be highly automatic, general, interactive, and efficient by implementing and empirically evaluating them in ACL2.

The first of these techniques is a powerful library of theorems designed to automate reasoning algebraically about the ordinals up to ε_0. The foundation of this work is the development of the first known complete set of algorithms for ordinal comparison, addition, subtraction, multiplication, and exponentiation. We presented these algorithms with detailed proofs of their correctness and complexity. We implemented the algorithms in ACL2 and mechanically verified them by proving that they satisfy well-known algebraic properties of the ordinals. We then created a library that significantly automates mechanical reasoning about the algebraic properties of the ordinals up to ε_0 and demonstrated through case studies that this library supports legacy results while enabling users to prove new results that were intractably difficult to prove in ACL2 previously.

Our second termination analysis technique is a new automatic termination analysis. The core of this analysis is the calling context graph (CCG), a manageable but surprisingly accurate abstraction of the looping behavior of a program. We have implemented the CCG analysis in ACL2 using a hierarchical algorithm that uses lightweight techniques to prove the termination of simpler programs, while using the full power of the CCG analysis for more difficult termination proofs. The result is an efficient and effective termination analysis,

as we demonstrate in our empirical evaluation. The ACL2 implementation of our CCG analysis can prove termination for 96% of functions, including 79.9% of the most difficult 1.5% of function definitions, using only ACL2's ground-zero theory and function definitions. We have shown that the CCG termination analysis represents a conservative extension to the ACL2 logic, and can be integrated with ACL2 in a sound and effective manner.

The result of integrating these two termination analysis techniques in ACL2 is a powerful framework for reasoning about termination. This framework can automatically prove termination of all but the most difficult of functions. It is efficient, completing its termination analysis in 1 to 3 seconds on average. It is general, and can be used to reason about any looping behavior allowed in our chosen class of languages. It is interactive, allowing users to work with the theorem prover to find new termination proofs when the automatic analysis fails.

REFERENCES

[1] Apt, K. R. and Olderog, E.-R., *Verification of Sequential and Concurrent Programs*. New York: Springer-Verlag, 1991.

[2] Arts, T. and Giesl, J., "Termination of term rewriting using dependency pairs," *Theoretical Computer Science*, vol. 236, pp. 133–178, 2000.

[3] Ball, T. and Rajamani, S. K., "The slam project: debugging system software via static analysis.," in *POPL*, pp. 1–3, 2002.

[4] Bancerek, G., "The reflection theorem," *Journal of Formalized Mathematics*, vol. 2, 1990.

[5] Belinfante, J. G., "Computer proofs in Gödel's class theory with equational definitions for composite and cross," *Journal of Automated Reasoning*, vol. 22, no. 3, pp. 311–339, 1999.

[6] Belinfante, J. G. F., "On computer-assisted proofs in ordinal number theory," *Journal of Automated Reasoning*, vol. 22, no. 3, pp. 341–378, 1999.

[7] Belinfante, J. G. F., "Reasoning about iteration in Gödel's class theory," in *Automated Deduction - CADE-19, Proceedings of the 19th International Conference on Automated Deduction* (Baader, F., ed.), vol. 2741 of *LNAI*, pp. 228–242, Springer-Verlag, 2003.

[8] Ben-Amram, A. M. and Lee, C. S., "Program termination analysis in polynomial time," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 1, p. 5, 2007.

[9] Berdine, J., Chawdhary, A., Cook, B., Distefano, D., and O'Hearn, P., "Variance analyses from invariance analyses," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 211–224, ACM Press, 2007.

[10] Berdine, J., Cook, B., Distefano, D., and O'Hearn, P. W., "Automatic termination proofs for programs with shape-shifting heaps.," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (Ball, T. and Jones, R. B., eds.), vol. 4144 of *Lecture Notes in Computer Science*, pp. 386–400, Springer, 2006.

[11] Bertot, Y. and Castéran, P., *Interactive Theorem Proving and Program Development, Coq'Art: the calculus of inductive constructions*. Texts in Theoretical Computer Science., Springer-Verlag, May 2004.

[12] Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., and Vojnar, T., "Programs with lists are counter automata.," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (Ball, T. and Jones, R. B., eds.), vol. 4144 of *Lecture Notes in Computer Science*, pp. 517–531, Springer, 2006.

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

[14] Bradley, A. R., Manna, Z., and Sipma, H. B., "The polyranking principle," in *Proc. 32^{nd} International Colloquium on Automata, Languages and Programming* (Caires, L., Italiano, G. F., Monteiro, L., Palamidessi, C., and Yung, M., eds.), vol. 3580 of *Lecture Notes in Computer Science*, pp. 1349–1361, Springer Verlag, 2005.

[15] Bradley, A. R., Manna, Z., and Sipma, H. B., "Termination of polynomial programs.," in Cousot [30], pp. 113–129.

[16] Braverman, M., "Termination of integer linear programs," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (Ball, T. and Jones, R. B., eds.), vol. 4144 of *Lecture Notes in Computer Science*, pp. 372–385, Springer, 2006.

[17] Brock, B. and Hunt, Jr., W. A., "Formal analysis of the motorola CAP DSP," in *Industrial-Strength Formal Methods*, Springer-Verlag, 1999.

[18] Brock, B. and Moore, J. S., "A mechanically checked proof of a comparator sort algorithm," in *Engineering Theories of Software Intensive Systems*, IOS Press, Amsterdam, 2005 (to appear).

[19] Brock, B., Kaufmann, M., and Moore, J. S., "ACL2 theorems about commercial microprocessors," in *Formal Methods in Computer-Aided Design (FMCAD'96)* (Srivas, M. and Camilleri, A., eds.), pp. 275–293, Springer-Verlag, 1996.

[20] Buchberger, B., "A theoretical basis for the reduction of polynomials to canonical forms," *SIGSAM Bull.*, vol. 10, no. 3, pp. 19–29, 1976.

[21] Cantor, G., "Beiträge zur Begründung der transfiniten Mengenlehre," *Mathematische Annalen*, vol. xlvi, pp. 481–512, 1895.

[22] Cantor, G., "Beiträge zur Bgründung der transfiniten Mengenlehre," *Mathematische Annalen*, vol. xlix, pp. 207–246, 1897.

[23] Cantor, G., *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover Publications, Inc., 1952. Translated by Philip E. B. Jourdain.

[24] Church, A. and Kleene, S. C., "Formal definitions in the theory of ordinal numbers," *Fundamenta mathematicae*, vol. 28, pp. 11–21, 1937.

[25] Codish, M. and Taboch, C., "A semantic basis for the termination analysis of logic programs," *The Journal of Logic Programming*, vol. 41, no. 1, pp. 103–123, 1999.

[26] Cook, B., Podelski, A., and Rybalchenko, A., "Abstraction refinement for termination," in *Static Analysis: 12th International Symposium, SAS 2005*, vol. 3672 of *LNCS*, pp. 87–102, September 2005.

[27] Cook, B., Podelski, A., and Rybalchenko, A., "Termination proofs for systems code," in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, (New York, NY, USA), pp. 415–426, ACM Press, 2006.

[28] Cook, B., Podelski, A., and Rybalchenko, A., "Proving thread termination," in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, (San Diego), June 2007.

[29] Cousot, P., "Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming.," in Cousot [30], pp. 1–24.

[30] Cousot, R., ed., *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, vol. 3385 of *Lecture Notes in Computer Science*, Springer, 2005.

[31] Cowles, J., Gamboa, R., and Van Baalen, J., "Using ACL2 arrays to formalize matrix algebra," in Kaufmann and Moore [61]. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/` (08/2007).

[32] Cowles, J. R. and Gamboa, R., "Unique factorization in ACL2: Euclidean domains.," in *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2006, Seattle, Washington, USA, August 15-16, 2006* (Manolios, P. and Wilding, M., eds.), pp. 21–27, ACM, 2006.

[33] Dennis, L. A. and Smaill, A., "Ordinal arithmetic: A case study for rippling in a higher order domain," in *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001* (Boulton, R. and Jackson, P., eds.), vol. 2152 of *LNCS*, pp. 185–200, Springer-Verlag, 2001.

[34] Dershowitz, N. and Okada, M., "Proof-theoritic techniques for term rewriting theory," in *3rd IEEE Symp. on Logic in Computer Science*, pp. 104–111, 1988.

[35] Dershowitz, N. and Reingold, E. M., "Ordinal arithmetic with list structures," in *Logical Foundations of Computer Science*, pp. 117–126, 1992.

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

[37] Dickson, L. E., "Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors," *American Journal of Mathematics*, vol. 35, pp. 413–422, 1913.

[38] Doner, J., "Definability in the extended arithmetic of ordinal numbers," *Dissertationes Mathematicae*, vol. 96, 1972.

[39] Doner, J. and Tarski, A., "An extended arithmetic of ordinal numbers," *Fundamenta Mathematicae*, vol. 65, pp. 95–127, 1969.

[40] Gallier, J. H., "What's so special about Kruskal's theorem and the ordinal Γ_0? A survey of some results in proof theory," *Annals of Pure and Applied Logic*, pp. 199–260, 1991.

[41] Gamboa, R. and Cowles, J., "Implementing a cost-aware evaluator for ACL2 expressions," in *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, (New York, NY, USA), pp. 71–80, ACM Press, 2006.

[42] Gentzen, G., "Die Widerspruchsfreiheit der reinen Zahlentheorie," *Mathematische Annalen*, vol. 112, pp. 493–565, 1936. English translation in M. E. Szabo (ed.), The Collected Works of Gerhard Gentzen, pp. 132-213, North Holland, Amsterdam, 1969.

[43] Giesl, J., Thiemann, R., Schneider-Kamp, P., and Falke, S., "Automated termination proofs with AProVE," in *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04)*, vol. 3091 of *LNCS*, pp. 210–220, Springer–Verlag, 2004.

[44] Gordon, M. J. C. and Melham, T. F., eds., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[45] Greve, D. A., Kaufmann, M., Manolios, P., Moore, J. S., Ray, S., Ruiz-Reina, J. L., Sumners, R., Vroon, D., and Wilding, M., "Efficient execution in an automated reasoning environment," *Journal of Functional Programming*. To Appear.

[46] Greve, D. A., Kaufmann, M., Manolios, P., Moore, J. S., Ray, S., Ruiz-Reina, J. L., Sumners, R., Vroon, D., and Wilding, M., "Efficient execution in an automated reasoning environment," Tech. Rep. TR-06-59, The University of Texas at Austin, November 2006.

[47] Greve, D. and Wilding, M., "A separation kernel formal security policy," in *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)* (Kaufmann, M. and Moore, J. S., eds.), July 2003. See URL `http://-www.cs.utexas.edu/users/moore/acl2/workshop-2003/` (08/2007).

[48] Greve, D., Wilding, M., and Hardin, D., "High-speed, analyzable simulators," in Kaufmann *et al.* [55], pp. 113–135.

[49] Greve, D. A., "Symbolic simulation of the JEM1 microprocessor," in *Formal Methods in Computer-Aided Design – FMCAD*, LNCS, Springer-Verlag, 1998.

[50] Henkin, L., Monk, J. D., and Tarski, A., "Cylindric algebras, part i," 1971.

[51] Hessenberg, G., "Grundbegriffe der mengenlehre," 1906.

[52] Jeannet, B., "New polka polyhedra library." `http://pop-art.inrialpes.fr/-people/bjeannet/newpolka/` (08/2007).

[53] Jr., W. A. H., Krug, R. B., and Moore, J. S., "Linear and nonlinear arithmetic in ACL2.," in *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings* (Geist, D. and Tronci, E., eds.), vol. 2860 of *Lecture Notes in Computer Science*, pp. 319–333, Springer, 2003.

[54] Kaufmann, M., Manolios, P., Moore, J. S., and Vroon, D., "Integrating CCG analysis into ACL2," in *Extended Abstracts of the 8th International Workshop on Termination, WST'06* (Geser, A. and Sondergaard, H., eds.), August 2006.

[55] Kaufmann, M., Manolios, P., and Moore, J. S., eds., *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

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

[57] Kaufmann, M. and Moore, J. S., "ACL2 homepage." See URL `http://www.cs.utexas.edu/users/moore/acl2` (08/2007).

[58] Kaufmann, M. and Moore, J. S., eds., *Proceedings of the ACL2 Workshop 2000*, The University of Texas at Austin, Technical Report TR-00-29, November 2000.

[59] Kaufmann, M. and Moore, J. S., "Structured theory development for a mechanized logic," *J. Autom. Reason.*, vol. 26, no. 2, pp. 161–203, 2001.

[60] Kaufmann, M. and Moore, J. S., eds., *Proceedings of the ACL2 Workshop 2002*, 2002.

[61] Kaufmann, M. and Moore, J. S., eds., *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/` (08/2007).

[62] Kaufmann, M. and Moore, J. S., eds., *Integrating Nonlinear Arithmetic into ACL2*, November 2004. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/` (08/2007).

[63] Kunen, K., *Set Theory - An Introduction to Independence Proofs*, vol. 102 of *Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland, 1980.

[64] Lee, C. S., "Ranking functions for size-change termination," August 2007. Under review. See `http://www.diku.dk/~amirben/downloadable/rf-wst.pdf` (08/2007).

[65] Lee, C. S., Jones, N. D., and Ben-Amram, A. M., "The size-change principle for program termination," in *ACM Symposium on Principles of Programming Languages*, vol. 28, pp. 81–92, ACM Press, 2001.

[66] Liu, H. and Moore, J. S., "Executable jvm model for analytical reasoning: a study," *Sci. Comput. Program.*, vol. 57, no. 3, pp. 253–274, 2005.

[67] Manolios, P., "Mu-calculus model-checking," in Kaufmann *et al.* [55], pp. 93–111.

[68] Manolios, P. and Moore, J. S., "Partial functions in ACL2," in Kaufmann and Moore [58].

[69] Manolios, P. and Moore, J. S., "Partial functions in ACL2," *Journal of Automated Reasoning*, vol. 31, no. 2, pp. 107–127, 2003.

[70] Manolios, P. and Vroon, D., "Algorithms for ordinal arithmetic," in *19th International Conference on Automated Deduction – CADE-19* (Baader, F., ed.), vol. 2741 of *LNAI*, pp. 243–257, Springer–Verlag, July/August 2003.

[71] Manolios, P. and Vroon, D., "Ordinal arithmetic in ACL2," in Kaufmann and Moore [61]. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/` (08/2007).

[72] Manolios, P. and Vroon, D., "Integrating reasoning about ordinal arithmetic into ACL2," in *Formal Methods in Computer-Aided Design: 5th International Conference – FMCAD-2004*, LNCS, Springer–Verlag, November 2004.

[73] Manolios, P. and Vroon, D., "Integrating static analysis and general-purpose theorem proving for termination analysis," in *ICSE'06, The 28th International Conference on Softwar Engineering, Emerging Results*, ACM, May 2006.

[74] Manolios, P. and Vroon, D., "Ordinal arithmetic: Algorithms and mechanization," *Journal of Automated Reasoning*, pp. 1–37, 2006.

[75] Manolios, P. and Vroon, D., "Termination analysis with calling context graphs," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (Ball, T. and Jones, R. B., eds.), vol. 4144 of *Lecture Notes in Computer Science*, pp. 401–414, Springer, 2006.

[76] Martín-Mateos, F.-J., Alonso, J.-A., Hidalgo, M.-J., and Ruiz-Reina, J.-L., "A formal proof of Dickson's Lemma in ACL2," in *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2003)* (Vardi, M. Y. and Voronkov, A., eds.), vol. 2850 of *LNCS*, pp. 49–58, Springer Verlag, 2003.

[77] Matthews, J., Moore, J. S., Ray, S., and Vroon, D., "Verification Condition Generation Via Theorem Proving," in *Proceedings of the* 13*th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)* (Hermann, M. and Voronkov, A., eds.), vol. 4246 of *LNCS*, (Phnom Penh, Cambodia), pp. 362–376, Nov. 2006.

[78] McCune, W., "Solution of the robbins problem," *Journal of Automated Reasoning*, vol. 19, no. 3, pp. 263–276, 1997.

[79] McCune, W. and Shumsky, O., "Ivy: A preprocessor and proof checker for first-order logic," in Kaufmann *et al.* [55], pp. 265–281.

[80] Miller, L. W., "Normal functions and constructive ordinal notations," *Journal of Symbolic Logic*, vol. 41, pp. 439–459, June 1976.

[81] Miné, A., "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, pp. 31–100, 2006. `http://www.di.ens.fr/~mine/publi/-article-mine-HOSC06.pdf` (08/2007).

[82] Moore, J. S., Lynch, T., and Kaufmann, M., "A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm," *IEEE Transactions on Computers*, vol. 47, pp. 913–926, September 1998.

[83] Moore, J. S., Lynch, T., and Kaufmann, M., "A mechanically checked proof of the AMD5$_K$86 floating-point division program," *IEEE Trans. Comp.*, vol. 47, pp. 913–926, September 1998.

[84] Moore, J. S. and Porter, G., "The apprentice challenge," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 3, pp. 193–216, 2002.

[85] Morris, F. and Jones, C., "An early program proof by Alan Turing," *IEEE Annals of the History of Computing*, vol. 6, pp. 139–143, April–June 1984.

[86] Owre, S., Rushby, J., and Shankar, N., "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)* (Kapur, D., ed.), pp. 748–752, Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag, June 1992.

[87] Paulson, L. C., "Set theory for verification: I. From foundations to functions," *Journal of Automated Reasoning*, vol. 11, no. 3, pp. 353–389, 1993.

[88] Paulson, L. C., *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[89] Paulson, L. C., "Set theory for verification: II. Induction and recursion," *Journal of Automated Reasoning*, vol. 15, no. 2, pp. 167–215, 1995.

[90] Paulson, L. C., "The reflection theorem: a study in meta-theoretic reasoning," in *18th International Conf. on Automated Deduction: CADE-18* (Voronkov, A., ed.), no. 2392 in LNAI, pp. 377–391, Springer-Verlag, 2002.

[91] Paulson, L. C., "The relative consistency of the axiom of choice mechanized using isabelle," *LMS Journal of Computation and Mathematics*, vol. 6, pp. 198–248, 2003.

[92] Paulson, L. C. and Grabczewski, K., "Mechanizing set theory: cardinal arithmetic and the axiom of choice," *Journal of Automated Reasoning*, vol. 17, pp. 291–323, 1996.

[93] Podelski, A. and Rybalchenko, A., "A complete method for the synthesis of linear ranking functions.," in *VMCAI*, pp. 239–251, 2004.

[94] Podelski, A. and Rybalchenko, A., "Transition invariants," in *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, (Washington, DC, USA), pp. 32–41, IEEE Computer Society, 2004.

[95] Ray, S. and Sumners, R., "Verification of an in-place quicksort in ACL2," in Kaufmann and Moore [60].

[96] Rogers, Jr., H., *Theory of Recursive Functions and Effective Computability*. MIT Press, 1st paperback ed., 1987.

[97] Rudnicki, P., "An overview of the MIZAR project," in *1992 Workshop on Types for Proofs and Programs*, 1992.

[98] Ruiz-Reina, J.-L., Alonso, J.-A., Hidalgo, M.-J., and Martin, F.-J., "Multiset relations: A tool for proving termination," in Kaufmann and Moore [58].

[99] Ruiz-Reina, J.-L., Jimenez, J. A. A., Hidalgo, M.-J., and Martín-Mateos, F.-J., "Formal reasoning about efficient data structures: A case study in ACL2.," in *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers* (Bruynooghe, M., ed.), vol. 3018 of *Lecture Notes in Computer Science*, pp. 75–91, Springer, 2003.

[100] Ruiz-Reina, J.-L., Martín-Mateos, F.-J., Alonso, J.-A., and Hidalgo, M.-J., "Formal correctness of a quadratic unification algorithm," *J. Autom. Reason.*, vol. 37, no. 1-2, pp. 67–92, 2006.

[101] Russinoff, D., "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998. http://www.onr.com/user/russ/david/k7-div-sqrt.html.

[102] Russinoff, D. M. and Flatau, A., "Rtl verification: A floating-point multiplier," in Kaufmann *et al.* [55], pp. 201–232.

[103] Russinoff, D. M., "A mechanically checked proof of correctness of the AMD5$_K$86 floating-point square root microcode," *Formal Methods in System Design Special Issue on Arithmetic Circuits*, 1997.

[104] Russinoff, D. M., "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998.

[105] Russinoff, D. M., "A mechanically checked proof of correctness of the AMD-K5 floating-point square root microcode," *Formal Methods in System Design*, vol. 14, pp. 75–125, 1999.

[106] Russinoff, D. M. and Flatau, A., "RTL verification: A floating-point multiplier," in Kaufmann *et al.* [55], pp. 201–231.

[107] Sawada, J., "Formal verification of divide and square root algorithms using series calculation," in Kaufmann and Moore [60].

[108] Schütte, K., *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.

[109] Setzer, A., "Ordinal systems," in *Sets and Proofs* (Cooper, B. and Truss, J., eds.), pp. 301–331, Cambridge University Press, 1999.

[110] Sustik, M., "Proof of Dixon's lemma using the ACL2 theorem prover via an explicit ordinal mapping," in Kaufmann and Moore [61]. See URL `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/` (08/2007).

[111] Tiwari, A., "Termination of linear programs," in *Computer-Aided Verification, CAV* (Alur, R. and Peled, D., eds.), vol. 3114 of *LNCS*, pp. 70–82, Springer, July 2004.

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

[113] Turing, A., "On computable numbers, with an application to the entscheidungsproblem," in *Proceedings of the London Mathematical Society*, vol. 42 of *Series 2*, pp. 230–265, 1936.

[114] Turing, A. M., "Systems of logic based on ordinals," *Proceedings of the London Mathematical Society*, vol. 45, no. 2, pp. 161–228, 1939. See URL `http://www.alanturing.net/` (08/2007).

[115] Turing, A. M., "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*, pp. 67–69, Universtity Mathematical Laboratory, Cambridge, June 1949.