On the termination of recursive algorithms in pure first–order functional languages with monomorphic inductive data types

by

Kostas Arkoudas

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author ....[illegible]..
Department of Electrical Engineering and Computer Science
May 4, 1996

Certified by..[illegible]..
David Allen McAllester
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ... [illegible] ....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

On the termination of recursive algorithms in pure first–order functional languages with monomorphic inductive data types

by

Kostas Arkoudas

Submitted to the Department of Electrical Engineering and Computer Science
on May 4, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

We present a new method for verifying the termination of recursively defined functions written in pure first-order functional languages and operating on monomorphic inductive data types. Our method extends previous work done by C. Walther on the same subject. Walther's method does not yield a direct answer as to whether a given function terminates. Instead, it produces a set of *termination hypotheses* whose truth *implies* that the function terminates. But then an external general-purpose induction theorem prover must be invoked to verify the hypotheses. By contrast, our method is a self-contained procedure that always produces a direct Yes/No verdict. Moreover, our method is much faster, and, unlike methods such as the one used in the Boyer-Moore system, it requires no interaction with the user. A programming environment in which one can define inductive data types and functions has been implemented in order to test our method, and the latter was found to be very efficient in practice, having readily proven the termination of many classic algorithms, including QuickSort, SelectionSort, MinSort, and other sorting algorithms, algorithms for computing greatest common divisors and transforming propositional wffs into normal forms, and several others.

Thesis Supervisor: David Allen McAllester
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

First of all I would like to acknowledge my advisor, David McAllester, for his patience, his approachability, his willingness to help his students, and his inspiring enthusiasm for and dedication to the field.

Needless to say, an invaluable resource during the many years of my studies has been the emotional and financial support of my parents (not necessarily in that order). So, as a very small token of appreciation, this thesis is dedicated to them (Mom and Dad: I know you always longed for an efficient syntactic treatment of termination in a recursive framework; long no more).

I would also like to thank my teachers at RPI, from whom I have the fondest memories, for stimulating my interest in Computer Science and Logic, and for genuinely believing me (however mistakenly) capable of contribution to these fields. I would particularly like to single out Selmer Bringsjord, Robert McNaughton, and David Musser, all three of whom have been critical in my career development so far. Last, but not least, I would like to thank Bob Givan for much helpful feedback during the writing of this thesis. Bob might well be the only person (besides David) who has had the misfortune to plow through almost every gory detail in this manuscript.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

An important problem in many verification logics is proving the termination of recursively defined functions. Termination is important not only from an operational standpoint (getting into an infinite loop is not a desirable turn of events at execution time) but also from a verification viewpoint. For instance, proving that QuickSort produces a sorted permutation of any given list of numbers presupposes that the algorithm halts for all appropriate inputs. In this paper we present an efficient and fully automatic method (automatic in that it requires no help from the user whatsoever) for proving that a given algorithm halts.

Two qualifying remarks are in order at this point. First, it is clear that the problem concerning us here is no other than the halting problem, which is well–known to be mechanically undecidable. Therefore, since our method is cast in a formalism that is equivalent in computing power to Turing machines, it is bound to be incomplete, meaning that it will fail to recognize many algorithms that do in fact terminate. The claimed advantage of the method is that it works for a remarkable variety of "commonly encountered" algorithms; and, furthermore, that it does so efficiently, and without requiring any guidance from or interaction with the user.

Secondly, our method is applicable only to pure functional languages featuring *inductive* data types such as the natural numbers, binary trees, etc.. In such languages there is no concept of state and the only looping mechanism available is recursion. An aditional requirement that we impose is that functions cannot be first-class values, meaning that they cannot be passed as arguments to other functions or be returned as results of function calls. From the viewpoint of automated deduction these are not very severe restrictions, since for most verification systems the language of choice is of the kind described above. The idea of proving termination mechanically has also been explored in imperative environments, i.e., for languages allowing assignment statements, while loops, etc. (see [2], [4], or [5]), although the problem is admittedly much more difficult in such frameworks.

A common thread underlying much of the existing work on termination, both on the procedural side and on the functional side, is the use of *termination functions*, also known as *convergence* functions. In pure functional paradigms, which are the focus of this paper, a termination function maps the arguments $s_1, \ldots, s_k$ of a function call $f(s_1, \ldots, s_k)$ to some value in a well-founded domain (W, R). Termination is established by proving that every time a recursive call is made, the value of the termination function strictly decreases in the sense of the well-ordering R.

Researchers have employed this idea in various ways, each with its own set of pros and cons. In the Boyer-Moore prover ([1]) the user must either discover and define the termination function himself, or at least steer the system to it by formulating certain *inductive lemmata* that provide hints as to why the algorithm terminates. The main advantage of this method is its broad applicability, the fact that it can be used on a large class of algorithms. In addition, the well-ordered range of the termination function need not be the natural numbers under the "greater than" relation (a common restriction of other methods, including ours); for instance, lexicographic orderings can be used just as well. Its downside is that it places a large share of the burden of proof on the user, who is expected to have a clear idea of why the algorithm terminates and to guide the system to the desired conclusions by formulating appropriate lemmata—an expectation that naive users often cannot meet.

Our method is based on Walther's work on the subject, as presented in [8]. The termination function used in that work is based on the concept of *size*; roughly, the ultimate goal is to show that every time a recursive call $f(s_1, \ldots, s_k)$ is made, the size of one or more arguments strictly decreases, which will allow us to conclude that f terminates. The gist of Walther's method is a calculus for inferring *weak size inequalities* of the form $s \preceq t$, asserting that the size of s is less than *or equal to* the size of t[1]. The terms s and t denote inductive objects such as lists, trees, numbers, etc., and can be either ground or not. We could, for example, derive the following inequalities from Walther's calculus: $\texttt{Reverse}(l) \preceq l$ (asserting that the size of a reversed list is not greater than the size of the original list), or $\texttt{pred}(\texttt{succ}(0)) \preceq \texttt{succ}(0)$ (making the not-so-exciting claim that $0 \leq 1$).

The main difference between Walther's method and ours is that his does not yield a direct verdict as to whether or not a given algorithm terminates. Instead, it produces a set of *termination hypotheses* whose validity *implies* that the algorithm terminates. Unfortunately, that still leaves us with the task of verifying the termination hypotheses, which can be a non-trivial problem. Walther resolves this by assuming that we have access to "a powerful induction theorem prover" which we can then invoke in order to verify the said hypotheses. The obvious drawback is that, even if we assume one is available, calling up a general-purpose theorem prover to verify hypotheses of essentially arbitrary complexity can be quite time-consuming. In fact Walther's method can take a long time even before it gets to the point where it needs to call the external theorem prover. Generating the termination hypotheses is an expensive procedure in and of itself, as it is based on a very rich calculus with a computationally complex inference problem.

Our approach modifies and extends Walther's idea in two ways. First, we simplify his calculus for weak size inequality in a way that expedites inference without sacrificing a great deal of expressiveness. Secondly, we give a new, equally simple calculus for *strict size inequality* that can be used to infer termination directly, thereby averting the need to resort to a general-purpose induction theorem prover. The result is an efficient and fully automated method that can serve as a conservative "black

[1]For now "size" can be understood intuitively. Think of the size of a list as the number of its elements, the size of a number as its magnitude, the size of a tree as the number of its nodes, and so on. A formal definition will be given in the sequel.

box" for determining whether an arbitrary algorithm (expressed in a language of the sort we desribed earlier) terminates. Of course the gains in simplicity and efficiency are not attained without repercussions. Technically speaking, Walther's method does recognize a larger class of algorithms. However, we make the empirical claim that our method works succefully on a remarkable multitude of algorithms encountered in practice. We substantiate this claim by illustrating our method on a collection of algorithms taken from [8] and elsewhere.

1.2 Preliminaries

1.2.1 Terms

Perhaps the most fundamental concept in our investigation is that of a **Herbrand term**. Given a collection of function symbols $\Sigma = \{f_1, f_2, \ldots\}$[2] in which every f_i has a unique natural number $\#(f_i)$ associated with it known as its **arity**, and a collection of **variables** $V = \{x, y, z, x_1, y_1, \ldots\}$, the set of Herbrand terms (or simply "terms") over Σ and V, notated $\mathcal{T}(\Sigma, V)$, is the smallest set containing V such that $f_i(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, V)$ whenever $t_1 \in \mathcal{T}(\Sigma, V), \ldots, t_n \in \mathcal{T}(\Sigma, V)$ and the arity of f_i is n. For instance, if $\Sigma = \{ \texttt{0}, \texttt{s}, \texttt{Plus}, \texttt{Fact} \}$ with $\#(\texttt{0}) = 0$, $\#(\texttt{s}) = 1$, $\#(\texttt{Fact}) = 1$, $\#(\texttt{Plus}) = 2$, and $V = \{x, y, z, x_1, y_1, \ldots\}$, then the following are all terms over Σ and V: $\texttt{0}, x, \texttt{s(0)}, \texttt{s(Plus(0, s(0)))}, \texttt{Fact(s(}y\texttt{))}, \ldots$ (note that function symbols of arity zero are treated as constants). The set $\mathcal{T}(\Sigma, \emptyset)$, written simply as $\mathcal{T}(\Sigma)$, comprises the so-called **ground terms** over Σ; these are the terms that do not contain any variables. All of the above terms with the exception of x and $\texttt{Fact(s(}y\texttt{))}$ are ground.

For any term $t \equiv f(\cdots) \in \mathcal{T}(\Sigma, V)$ we will write $Top(t)$ to denote f—the "top function symbol" of t. e.g. $Top(\texttt{Plus(s(s(0)),0)} = \texttt{Plus}$, $Top(\texttt{0}) = \texttt{0}$, etc. A term s is said to be a **subterm** of t, written $s \sqsubseteq t$, if either (i) $s \equiv t$, or else (ii) $t \equiv f(t_1, \ldots, t_n)$ and s is a subterm of a t_i. If $s \sqsubseteq t$ and $s \not\equiv t$ we say that s is a **proper substerm** of t and we write $s \sqsubset t$. The inverse of the $\sqsubseteq$ ($\sqsubset$) relation is denoted by

[2] Σ is often called a **signature**.

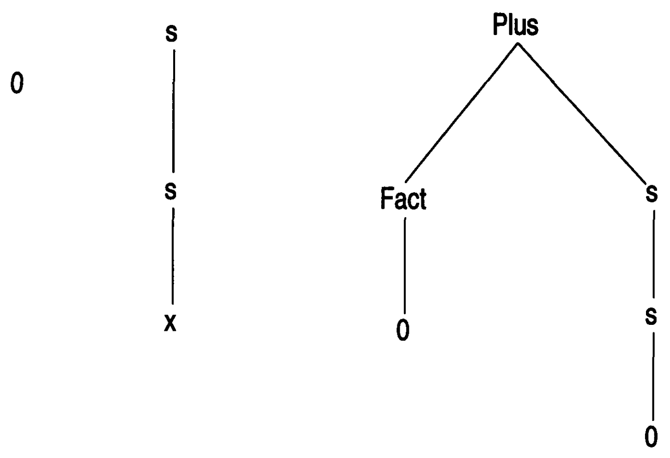

Figure 1-1: Representing terms as trees.

$\sqsupseteq$ ($\sqsupset$), and if $t \sqsupseteq s$ ($t \sqsupset s$) we say that t is a (proper) **superterm** of s. Further, if $t \equiv f(t_1, \ldots, t_n)$ then we say that $t_1, \ldots, t_n$ are the **immediate subterms** of t. We write $\#(t)$ to denote the number of t's immediate subterms, and $t \downarrow i$ to denote the i^{th} such subterm. e.g. for $t \equiv \texttt{Plus(Fact(s(0)),s(s(0)))}$ we have $\#(t) = 2$, $t \downarrow 1 = \texttt{Fact(s(0))}, t \downarrow 2 = \texttt{s(s(0))}, (t \downarrow 2) \downarrow 1 = \texttt{s(0)}$, $\#(t \downarrow 1) = 1$, and so on. In general, the immediate subterms of any term t are $t_1, \ldots, t \downarrow \#(t)$.

A term t is essentially an ordered tree structure and it is natural to depict it as such, with $Top(t)$ at the root and the trees $t \downarrow 1, \ldots, t \downarrow \#(t)$ as the root's children, from left to right. Fig. 1.2.1 shows the terms `0`, `s(s(`x`))`, and `Plus(Fact(0),s(s(0)))` as trees. Note that only variables and constants (function symbols of arity zero) can appear at the leaves; internal nodes must be occupied by function symbols of positive arity. In particular, the number of children of a certain node is equal to the arity of the function symbol labeling that node. Therefore, terms can be represented in two ways, as linear (one-dimensional) strings of characters, and as (two-dimensional) trees[3]. We will use both as we see fit, so at times we will be speaking of a term as if it were a tree (e.g. we might say "consider the leaves of `Plus(s(0),s(0))`"), while at other times we will be viewing terms as strings.

The **size** of a term t is simply the number of its nodes (viewing t as a tree). More formally: the size of a constant or a variable is one, while the size of a term

[3]Note that the subterm/subtree relations are isomorphic in these two representations.

$f(t_1,\ldots,t_n)$ is one more than the sum of the sizes of $t_1,\ldots,t_n$. Hence, the sizes of the terms appearing in Fig. 1.2.1 are 1, 3, and 6, respectively. Term size should not be confused with term **height**, which is simply the height of the tree representation of the term (thus the height of constants and variables is zero, while the height of $f(t_1,\ldots,t_n)$ is one more than the height of the highest term amongst $\{t_1,\ldots,t_n\}$).

Given a signature Σ and a set of variables V, a **substitution** is a function $\theta : V \longrightarrow \mathcal{T}(\Sigma,V)$ that is the identity almost everywhere (i.e. $\theta(v) \neq v$ only for finitely many v). We will usually write out a substitution θ as $[v_1 \mapsto t_1,\ldots,v_n \mapsto t_n]$, the obvious interpretation being that $\theta(v_i) = t_i$ for $i \in \{1,\ldots,n\}$ (with $t_i \neq v_i$), while $\theta(v) = v$ for $v \notin \{v_1,\ldots,v_n\}$. Given a term $t \in \mathcal{T}(\Sigma,V)$ and a substitution $\theta = [v_1 \mapsto t_1,\ldots,v_n \mapsto t_n] : V \longrightarrow \mathcal{T}(\Sigma,V)$, we will write $t\,\theta$ (or $t\,[v_1 \mapsto t_1,\ldots,v_n \mapsto t_n]$ in expanded notation) to denote the term obtained from t by replacing each occurence of v_i by the corresponding t_i. More precisely,

$$t\,\theta = \begin{cases} \theta(t) & \text{if } t \in V \\ f(s_1\,\theta,\ldots,s_k\,\theta) & \text{if } t \equiv f(s_1,\ldots,s_k). \end{cases}$$

For instance, $\texttt{Plus(s(}x\texttt{),Plus(}x\texttt{,s(}y\texttt{)))}[x \mapsto \texttt{s(0)}, y \mapsto \texttt{Plus(}z\texttt{,s(0))}] \equiv$ $\texttt{Plus(s(s(0)),Plus(s(0),s(Plus(}z\texttt{,s(0)))))}$. We will often write $[\overrightarrow{v} \mapsto \overrightarrow{t}]$ as a shorthand for $[v_1 \mapsto t_1,\ldots,v_n \mapsto t_n]$.

For a certain signature Σ and a set of variables V, we define $\mathcal{B}(\Sigma,V)$, the set of **Boolean expressions over Σ and V**, as follows:

- The symbols **True** and **False** are in $\mathcal{B}(\Sigma,V)$.
- For any two terms s and t in $\mathcal{T}(\Sigma,V)$, the expression $(s = t)$ is in $\mathcal{B}(\Sigma,V)$.
- If E_1 and E_2 are in $\mathcal{B}(\Sigma,V)$, then so are $(E_1 \wedge E_2)$, $(E_1 \vee E_2)$, and $(\neg E_1)$. Expressions of the form $\neg\,(s = t)$ will be abbreviated as $s \neq t$.
- Nothing else is an element of $\mathcal{B}(\Sigma,V)$.

When writing such expressions we will omit outer parentheses in accordance with the usual conventions, and, to enhance readability, we will freely use square brackets along with parentheses. Also, when $V = \emptyset$ we will write $\mathcal{B}(\Sigma, V)$ simply as $\mathcal{B}(\Sigma)$; the elements of $\mathcal{B}(\Sigma)$ will be called *ground Boolean expressions* over Σ. Finally, if $E \in \mathcal{B}(\Sigma, V)$ and θ is a substitution from V to $\mathcal{T}(\Sigma, V)$, we will write $E\theta$ to denote the expression obtained from E by carrying out the replacements prescribed by θ. More formally:

$$E\theta = \begin{cases} E & \text{if } E \in \{\textbf{True}, \textbf{False}\} \\ (s\theta = t\theta) & \text{if } E \equiv (s = t) \\ (E_1\theta \; op \; E_2\theta) & \text{if } E \equiv (E_1 \; op \; E_2), \text{ for } op \in \{\wedge, \vee\} \\ (\neg E_1 \, \theta) & \text{if } E \equiv (\neg E_1). \end{cases}$$

1.2.2 Trees

Let T be a tree in which every node has a finite number of children (we shall call T a **finitely branching** tree). We can assign to each node u in T a unique tuple of positive integers called the **position of** u **in** T, and denoted by $Pos_T(u)$, as follows:

- The position of the root of T is $[1]$.
- If a node u has k children $v_1, \ldots, v_k$, from left to right, and $Pos_T(u) = [i_1, \ldots, i_n]$, then $Pos_T(v_j) = [i_1, \ldots, i_n, j]$.

By way of illustration, Fig. 1.2 shows the position of every node in the tree

```
Plus(s(0),Plus(Fact(0),s(s(0)))).
```

The nodes of a tree are partitioned into **levels** as usual: the root is at the first level, while the children of an n^{th}–level node are at the $(n+1)^{st}$ level. Note that the level of a node u in T is equal to the length of the tuple $Pos_T(u)$; i.e. $Pos_T(u) = [i_1, \ldots, i_n]$ iff u is at the n^{th} level of T.

If u and v are nodes in a finitely branching tree T, we shall say that u **precedes** v, or that u **is to the left of** v, written $u <_T v$, iff $Pos_T(u) <' Pos_T(v)$, where $<'$ is the lexicographic extension of the less-than $<$ relation on N to $\cup N^i, i = 1, 2, \ldots$ To

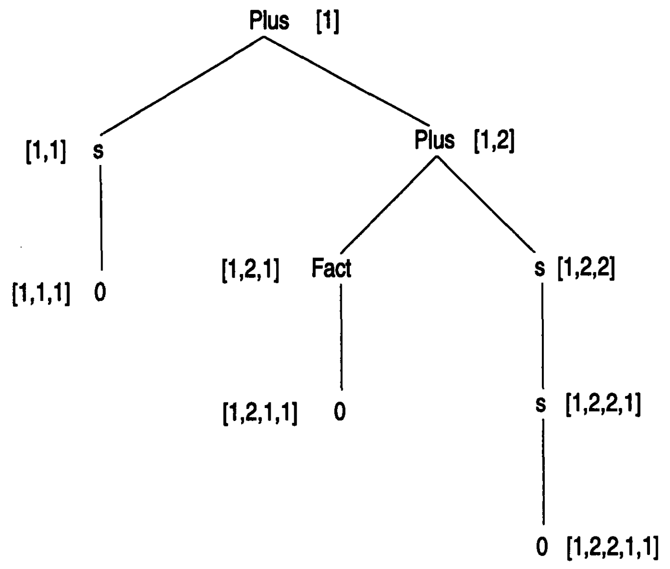

Figure 1-2: The tuple $Pos_T(u)$ uniquely identifies any node u in a finitely branching tree T.

minimize notational clutter, we will drop the subscript $_T$ whenever it is immaterial or can be easily deduced from the context. It is not difficult to see that $u <_T v$ iff u is visited before v in an in-depth traversal of T. Clearly, the relation $<_T$ well-orders any finite-branching tree T, and we shall write $ORD(T)$ to denote the unique ordinal that is isomorphic to $(T, <_T)$; moreover, we will let $\pi_T : ORD(T) \longrightarrow T$ denote the said isomorphism[4]. So, for example, if T is the term `Plus(s(0), Fact(s(s(0))))` then $ORD(T) = 7$, while $\pi(0) = \texttt{Plus}, \pi(1) = \texttt{s}, \pi(2) = \texttt{0}, \pi(3) = \texttt{Fact}$, etc. Notice that for infinite trees $ORD(T)$ could very well be greater than ω, say $\omega + 5$ or $3\omega + 87$. Finally, we shall call $<_T$ the **standard well-ordering** of T.

[4]Notational abuse: in the expression $\pi_T : ORD(T) \longrightarrow T$, we are using T on the right side of the arrow as the set of T's nodes.

Chapter 2

The language IFL

In this chapter we present the syntax and semantics of IFL, a pure first–order functional programming language with monomorphic inductive data types.[1]

2.1 Data types in IFL

In IFL a data type is defined in accordance with the following grammar:

<*Data-Type*> ::= **Type** <*Constructor-Declaration*>$^+$.

<*Constructor-Declaration*> ::= <*Constructor-Name*> **:** <*Type-Name*> |

<*Constructor-Name*> **(** <*Selector-Sequence*> **)** **:** <*Type-Name*>

<*Selector-Sequence*> ::= <*Selector*> | <*Selector*> **,** <*Selector-Sequence*>

<*Selector*> ::= <*Selector-Name*> **:** <*Selector-Type-Name*>

where the notation <*A*>$^+$ simply means "one or more occurences of the non-terminal *A*", and <*Constructor-Name*>, <*Type-Name*>, <*Selector-Name*>, and <*Selector-Type-*

[1]A brief note on terminology: "pure" refers to the total absence of the kind of side-effects usually associated with languages supporting state; "first-order" simply means that, unlike objects like numbers, lists, etc., functions are not first-class values (most notably, they cannot be passed as arguments to other functions or be returned as results of function calls); finally, "monomorphic" is just the opposite of "polymorphic". A polymorphic data structure can contain objects of various distinct types—e.g. a polymorphic list may contain an integer, a string and even an entire queue of, say, real numbers, as individual elements. By contrast, the elements of a monomorphic list must all be of the same type, e.g. they must all be integers.

Name> are identifiers (strings of letters and/or numbers and other symbols, beginning with a letter).

Less formally, a data type is defined by specifying a sequence of one or more function symbols known as **constructors**, the idea being that the elements of the data type will be the ground terms built from these constructor symbols, with constructors of arity zero treated as constants. We will use the term **object** to refer to an element of such a data type.[2] An illustration is provided by the following definition of the data type `NatNum` (natural numbers):

```
Type    zero:NatNum
        succ(pred:NatNum):NatNum.
```

Here the type `NatNum` is "generated" by two constructors: `zero`, a constant, and `succ`, a unary constructor that takes an argument of `NatNum` type and returns ("constructs") another object of that same type. So the objects of this data type, i.e. the natural numbers, are all and only the ground terms built from `zero` and `succ`, namely

{ `zero, succ(zero), succ(succ(zero)), ...` }.

Note that every constructor that has a positive arity n also has n **selectors** $sel_1, \ldots, sel_n$ associated with the argument positions $1, \ldots, n$. The user must specify both the name and the type of each of these selectors, in the format

```
selector-name:selector-type-name
```

when declaring the constructor. In the case of `NatNum` there is only one constructor of positive arity, `succ`, which has one selector named `pred` (for "predecessor"). The selector `pred` has type `NatNum` and is such that `pred(succ(`x`))` = x for all $x \in$ `NatNum`.

Mathematically, selectors are the inverses of constructors. Intuitively, a constructor c of positive arity n "binds" or "pulls together" n objects $w_1, \ldots, w_n$ to produce a new complex object w, which, in a sense, contains the objects $w_1, \ldots, w_n$. By con-

[2]We will be riding roughshod over the sign/reference distinction by conflating a syntactic entity (a term) with the mathematical object it is intended to denote.

trast, sel_i^c, the selector associated with the i^{th} argument position of c, when applied to the object w will return the component object w_i — mathematically speaking, sel_i^c will *project* the object w along the i^{th} co-ordinate. Thus, constructors of positive arity *build* new objects by pulling together one or more objects of various types, while selectors *dismantle* objects so built by retrieving their constituent components.

It is natural to view a constructor c that produces objects of type T as a function $c : T_1 \times \cdots T_n \longrightarrow T$, where $n \geq 0$ is the arity of c. For instance, the signatures of `zero` and `succ` are as follows: `zero:`$\longrightarrow$ `NatNum` and `succ:NatNum` $\longrightarrow$ `NatNum`. We will call T **the type of** c and T_i the i^{th} **component type** of c, for each $i \in \{1, \ldots, n\}$. And, as was already indicated, we will write sel_i^c to denote the selector associated with the i^{th} argument position of c (so that, for instance, $sel_1^{\mathtt{succ}} = \mathtt{pred}$). Therefore, if $w_1, \ldots, w_n$ are objects of types $T_1, \ldots, T_n$, respectively, then for every $i \in \{1, \ldots, n\}$ we have $sel_i^c(c(w_1, \ldots, w_n)) = w_i$. We will view sel_i^c as a function from T to T_i, and we will call T_i **the type** of sel_i^c. In contradistinction, we will call T **the argument type** of sel_i^c. Note that the type of sel_i^c is the i^{th} component type of c, while the argument type of sel_i^c is simply the type of c.

Constructors such as `succ` that take as arguments one or more objects of the same type as the object that they produce are said to be **reflexive**. More precisely, a constructor $c : T_1 \times \ldots \times T_n \longrightarrow T$ is said to be reflexive if at least one of its component types is T, i.e. if there is an $i \in \{1, \ldots, n\}$ such that $T_i = T$. Otherwise c is called **irreflexive**. Trivially, constructors of zero arity are irreflexive. In a similar spirit, a selector is called reflexive or irreflexive according to whether or not its type it the same as its argument type. An example of a reflexive selector is `pred`. It follows that a constructor is reflexive iff at least one of its selectors is reflexive. Note that every data type must have at least one irreflexive constructor in order to generate "base-case" objects, so that production of more complex (reflexive) objects can "get started", so to speak.

Another data type that we will frequently use is `NatList`, lists of natural numbers. In IFL this type can be defined as follows:

```
Type   empty:NatList
```

```
    add(head:NatNum,tail:NatList):NatList.
```

Here we have two constructors, the constant `empty` (denoting the empty list) and the binary operator `add(n,l)`, which forms the list obtained by inserting a number `n` at the front of a list `l`. Note that `add` is reflexive since its second argument is of type `NatList`. The two selectors of `add`, $sel_1^{\mathtt{add}}$ and $sel_2^{\mathtt{add}}$, are, respectively, `head:NatList` $\longrightarrow$ `NatNum` and `tail:NatList` $\longrightarrow$ `NatList`. Unlike `head`, `tail` is a reflexive selector of `add` since it takes a list and returns a list. Using this notation, the lists $[], [3]$, and $[1,2,1]$ are represented, respectively, by the terms

`empty`,

`add(succ(succ(succ(zero))),empty)`, and

`add(succ(zero),add(succ(succ(zero)),add(succ(zero),empty)))`.

The following list provides additional examples of some classic data types and their definitions in IFL:

- Integers:

  ```
  Type    int(sign:Polarity,abs-value:NatNum):Integer.
  ```

  where polarity is represented as follows:

  ```
  Type    positive:Polarity
          negative:Polarity.
  ```

- Positive integers:

  ```
  Type    one:PosInt
          inc(dec:PosInt):PosInt.
  ```

  Note the isomorphism with the natural numbers (map(`one`) = `zero`, map(`inc`(x)) = `succ`(map(x))).

- Rational numbers:

  ```
  Type    rat(sign:Polarity,num:NatNum,denum:PosInt):Rational.
  ```

- Truth values:

  ```
  Type    true:Boolean
          false:Boolean.
  ```

- Formulas of propositional logic:

  ```
  Type   atomic(subscript:NatNum):Wff
         not(neg:Wff):Wff
         and(l-and:Wff,r-and:Wff):Wff
         or(l-or:Wff,r-or:Wff):Wff.
  ```

- LISP s–expressions:

  ```
  Type   nil:S-expr
         atom(number:NatNum):S-expr
         cons(car:S-expr,cdr:S-expr):S-expr.
  ```

- Stacks of integers:

  ```
  Type   empty-stack:IntStack
         push(top:Integer,pop:IntStack):IntStack.
  ```

  With this notation,
  `pop(push(int(negative,succ(zero)),empty-stack)) =`
  `empty-stack`. Note the isomorphism with `NatList`.

- Characters:

  ```
  Type   a:Char
         b:Char
         c:Char
         ⋮
         z:Char.
  ```

- Strings (lower-case, over the English alphabet):

  ```
  Type   empty-string:String
         concat(first:Char,rest:String):String.
  ```

  Note that, as defined, strings, lists, and stacks are essentially one and the same structure.

- Binary trees of numbers:

```
Type   null:BTree
       tree(root-value:NatNum,
            l-branch:BTree,r-branch:BTree):BTree.
```

At this point we need to establish some notational conventions. First, in the interest of brevity we will hereafter use the symbol 0 in place of `zero`, the symbol s in place of `succ`, and $\mathtt{s}^{\mathtt{i}}(\mathtt{0})$, for $\mathtt{i} > 0$, in place of $\overbrace{\mathtt{succ}(\cdots(\mathtt{succ}}^{\text{i times}}(\mathtt{0})))$. Secondly, given a data type T we will write $RCON_T$ ($IRCON_T$) for the set of the reflexive (irreflexive) constructors of T (e.g. $RCON_{\mathtt{NatList}} = \{\mathtt{add}\}$ and $IRCON_{\mathtt{NatList}} = \{\mathtt{empty}\}$). Further, we will write CON_T for $RCON_T \cup IRCON_T$ and SEL_T for the set of all selectors of all constructors of T. For instance,

$$CON_{\mathtt{NatNum}} = \{\mathtt{0}, \mathtt{s}\}$$

and

$$SEL_{\mathtt{BTree}} = \{\ \mathtt{root\text{-}value}, \mathtt{l\text{-}branch}, \mathtt{r\text{-}branch}\ \}.$$

Finally, we will assume that every data type T comes equipped with an infinite supply of variables V_T. As usual, elements of V_T will be used whenever we wish to denote arbitrary objects of type T. The exact contents of V_T are largely a matter of taste and will be made precise only when it is necessary to do so. As an example, we might let $V_{\mathtt{NatNum}} = \{n, m, k, n_1, m_1, k_1, n_2, \ldots\}$ and $V_{\mathtt{Wff}} = \{\phi, \psi, \phi_1, \psi_1, \ldots\}$.[3] Last, we will use $v_1, v_2, \ldots$ as metavariables (i.e. variables ranging over the various V_T).

Notice that our stipulations so far do not settle the question of what happens when we apply a selector of a constructor $c : T_1 \times \ldots \times T_n \longrightarrow T$, say sel_i^c, to an object of type T that is *not* constructed by c, i.e. whose top function symbol is not c. For instance what are the "values" of terms like `pred(0)`, `head(empty)`, or `l-or(and(⋯))`? This is an issue because we wish to regard each selector sel_i^c as a *total* function from T to T_i; hence sel_i^c should return some object of type T_i no matter what object of type T it is given as input. Following Walther, we adopt the convention that sel_i^c, when applied to a term of type T whose top symbol is not c,

[3]Needless to say, we will assume that if T_1 and T_2 are two distinct types, then V_{T_1} and V_{T_2} are disjoint.

should return the **minimal** object of type T_i, where the minimal object of type T_i is either

1. the leftmost constructor of zero arity in the definition of T_i, if one exists, or else
2. the object obtained by applying the leftmost irreflexive constructor of T_i (one must exist) to the minimal elements of its component types.

So, for example, the minimal objects of `NatNum`, `NatList` and `Wff`, are, respectively, `0`, `empty`, and `atomic(0)`. Therefore,

```
pred(0) = head(empty) = 0,
   tail(empty) = empty,
l-or(and(···)) = atomic(0),
```

and so forth.

We end this section by introducing the notion of a well-formed definition. In particular, we say that the definition of a data type T is **well-formed** with respect to the definitions of $n \geq 0$ other data types $T_1, \ldots, T_n$ if

$$(CON_T \cup SEL_T) \cap (CON_{T_i} \cup SEL_{T_i}) = \emptyset \text{ for every } i = 1, \ldots, n$$

i.e. if all the constructor and selector symbols appearing in the definition of T are new with respect to the T_i. Moreover, the same definition is said to be **well-typed** (again with respect to $T_1, \ldots, T_n$) if (1) the type of every selector of T is either T itself (in which case the selector is reflexive), or one of the T_is, and (2) at least one of the constructors of T is irreflexive. Each of the foregoing definitions is well-formed and well-typed with respect to all the definitions that precede it.

2.2 Procedures in IFL

Procedures[4] in IFL are defined by means of the following grammar:

<Procedure> ::= **Procedure** *<Proc-Name>*(*<Parameters>*): *<Return-Type>* ;

Begin *<Body>* **End.**

<Parameters> ::= *<ParamDec>* | *<ParamDec>*;*<Parameters>*

<ParamDec> ::= *<ParamList>* :*<ParamType>*

<ParamList> ::= *<ParamName>* | *<ParamName>* , *<ParamList>*

<Body> ::= **If True Then** *<Term>*; | *<IF-THEN-ELSE>*$^+$ *<Term>*;

<IF-THEN-ELSE> ::= **If** *<Bool-Exp>* **Then** *<Term>* **Else**

where *<Proc-Name>, <Ret-Type>, <ParamName>*,[5] and *<ParamType>* are identifiers, *<Term>* is given by the usual grammar for Herbrand terms, and a *<Bool-Exp>* is either one of the constants {**True, False**}, or an *atomic expression* of the form $(s = t)$ or $(s \neq t)$ for two terms s and t, or a conjunction or disjunction of two Boolean expressions[6]. Note that the body of a procedure is either (1) a single statement `If True Then r;` for some term `r`, or (2) a sequence of `If-Then-Else` statements ending with a mandatory "catch-all" clause `Else r;`. A procedure with body

```
If E₁ Then r₁
Else
    ⋮
      If Eₘ Then rₘ
      Else
         rₘ₊₁;
```

is equivalent to one with body

[4]The term "function" would have been more fitting than "procedure", but we chose the latter to avoid any ambiguity between the mathematical concept of a function and the IFL concept. As we define them here, procedures return values and generally behave like LISP or ML functions (save the obvious differences, of course), not as Pascal procedures.

[5]Although the name of a parameter can be an arbitrary identifier, usually we will assume that it comes from V_T, where T is the parameter's type.

[6]These will be formed with the keywords **`And`** and **`Or`**.

```
If E1 Then r1
Else
      ⋮
        If Em Then rm
        Else
            If True Then rm+1;
```

so we will use the expression $[< \mathtt{E}_1, \mathtt{r}_1 >, \ldots, < \mathtt{E}_\mathtt{m}, \mathtt{r}_\mathtt{m} >, < \mathtt{True}, \mathtt{r}_{\mathtt{m}+1} >]$ as a generic abbreviation for the body of an arbitrary procedure (hence for $\mathtt{m} = 0$ the expression $[< \mathtt{True}, \mathtt{r}_{\mathtt{m}+1} >]$ covers the case in which the body of the procedure comprises the single statement `If True Then r1;`).

Observe that this syntax choice ensures that procedures are **deterministic** and **case-complete**. "Deterministic" means that the various antecedents in the body of the procedure are mutually exclusive, i.e. that at most one of them will be true for any given sequence of arguments. This is waranteed in our formalism because, on account of the `Else` tokens, each antecedent $\mathtt{E}_\mathtt{j}$ is conjoined to the negation of every preceding antecedent. "Case-complete" means that the antecedents are jointly exhaustive, i.e. that at least one of them will be true for any sequence of arguments. This is ensured in our syntax by the requisite trailing "catch-all" clause[7]. Thus determinism *and* case-completeness in tandem entail that for any given sequence of arguments *exactly* one antecedent will be true, and hence, from an evaluation standpoint, that exactly one `If-Then` statement will be "fired".

The following is a collection of some useful primitive recursive functions expressed as IFL procedures operating on the data types we introduced in the previous section.

- Binary addition on natural numbers:

```
Procedure Plus(n,m:NatNum):NatNum;
Begin
If m = 0 Then n
Else
```

[7]The case in which the body contains only one statement is no exception, as in that case the antecedent is required to be the constant **True**.

```
    s(Plus(n,pred(m)));
End.
```

- Subtraction ($n - m$):

```
Procedure Minus(n,m:NatNum):NatNum;
Begin
If m = 0 Then n
Else
    Minus(pred(n),pred(m));
End.
```

- Multiplication:

```
Procedure Times(n,m:NatNum):NatNum;
Begin
If n = 0 Then 0
Else
    Plus(m,Times(pred(n),m));
End.
```

- Exponentiation (n^m):

```
Procedure Exp(n,m:NatNum):NatNum;
Begin
If m = 0 Then s(0)
Else
    Times(n,Exp(n,pred(m)));
End.
```

- The factorial function:

```
Procedure Fact(n:NatNum):NatNum;
Begin
If n = 0 Then s(0)
Else
```

```
    Times(n,Fact(pred(n)));
End.
```

- The Fibonacci sequence:

```
Procedure Fib(n:NatNum):NatNum;
Begin
If n = 0 Then 0
Else
    If n = s(0) Then s(0)
    Else
        Plus(Fib(pred(n),Fib(pred(pred(n)))));
End.
```

- The less-than predicate ($n < m$):

```
Procedure Less(n,m:NatNum):Boolean;
Begin
If (n = 0) And (m = 0) Then false
Else
    If (n = 0) And (m ≠ 0) Then true
    Else
        If (n ≠ 0) And (m = 0) Then false
        Else
            Less(pred(n),pred(m));
End.
```

- The equality predicate (for `NatNum`):

```
Procedure Equal(n,m:NatNum):Boolean;
Begin
If (n = 0) And (m = 0) Then true
Else
    If (n = 0) And (m ≠ 0) Then false
    Else
```

```
        If (n ≠ 0) And (m = 0) Then false
        Else
            Equal(pred(n),pred(m));
End.
```

- The less-than-or-equal-to predicate ($n \leq m$):

```
Procedure Leq(n,m:NatNum):Boolean;
Begin
If Less(n,m) = true Or Equal(n,m) = true Then true
Else
    false;
End.
```

- The length of a list of numbers:

```
Procedure Length(l:NatList):NatNum;
Begin
If l = empty Then 0
Else
    s(Length(tail(l)));
End.
```

- List membership:

```
Procedure Member(n:NatNum;l:NatList):Boolean;
Begin
If l = empty Then false
Else
    If head(l) = n Then true
    Else
        Member(n,tail(l));
End.
```

- Appending two lists:

```
Procedure Append(l1,l2:NatList):NatList;
Begin
If l1 = empty Then l2
Else
    add(head(l1),Append(tail(l1),l2));
End.
```

- Reversing a list:

```
Procedure Reverse(l:NatList):NatList;
Begin
If l = empty Then empty
Else
    Append(Reverse(tail(l)),add(head(l),empty)));
End.
```

- Traversing a binary tree "in order":

```
Procedure InOrder(t:BTree):NatList;
Begin
If t = null Then empty
Else
    Append(InOrder(l-branch(t)),
           add(value(t),InOrder(r-branch(t))));
End.
```

Finally, just as we did for data types, we will say that a definition of a procedure f is well-formed with respect to the definitions of $m \geq 0$ other procedures $f_1, \ldots, f_m$ and $n > 0$ data types $T_1, \ldots, T_n$ if every function symbol appearing in the body of f is either f itself (in which case f is recursive), or one of the f_i, or a constructor or selector symbol introduced by the definition of some T_i. In addition, f is well-typed (again, with respect to $f_1, \ldots, f_m$ and $T_1, \ldots, T_n$) if (i) the return type of f and the types of its parameters are amongst the T_i, and (ii) in every conditional statement **If**

`E Then r` in the body of f, `r` is a term of type T, where T is the return type of f.[8]

2.3 Modules

2.3.1 Basic definitions and notational conventions

A **module** M is defined to be a finite *sequence* of definitions of data types and procedures. Modules are defined as sequences because the idea is that they are developed incrementally, in stages: we usually start from scratch, with the empty module M_0; then we obtain a module M_1 by defining a data type T_1 whose constructors are either constants or have T_1 as a component type; then we *extend* M_1 by defining either a new data type T_2 or a procedure f_1, thereby obtaining a new module M_2; and so forth. Nevertheless, if the order of the relevant definitions is tacitly understood we will occasionally treat a given module as a set, rather than a sequence, of data types and procedures.

If M is a module comprising $n > 0$ types $T_1, \ldots, T_n$ and $m \geq 0$ procedures $f_1, \ldots, f_m$, we write

- CON_M for $\mathit{CON}_{T_1} \cup \cdots \cup \mathit{CON}_{T_n}$.
- SEL_M for $\mathit{SEL}_{T_1} \cup \cdots \cup \mathit{SEL}_{T_n}$.
- PROC_M for $\{f_i \mid 1 \leq i \leq m\}$.
- Σ_M for $\mathit{CON}_M \cup \mathit{SEL}_M \cup \mathit{PROC}_M$. We call Σ_M the **signature** of M; it is simply the set of all function symbols in M.
- V_M for $V_{T_1} \cup \cdots \cup V_{T_n}$.
- TYPES_M for $\{T_1, \ldots, T_n\}$.

For example, if M comprises the two data types **`NatNum`** and **`NatList`**, so that $\mathit{TYPES}_M = \{$`NatNum`, `NatList`$\}$, and the four procedures `Plus`, `Minus`, `Times`, and

[8]A precise definition of "a term of type T" is given in the next section, though an intuitive understanding is sufficient at this point.

`Append`, then

$$CON_M = \{\texttt{0},\ \texttt{s},\ \texttt{empty},\ \texttt{add}\},$$
$$SEL_M = \{\texttt{pred},\ \texttt{head},\ \texttt{tail}\},$$
$$PROC_M = \{\texttt{Plus},\ \texttt{Minus},\ \texttt{Times},\ \texttt{Append}\},$$

and Σ_M is the union of all three of these. Continuing, we define the **terms of type** T, for any $T \in TYPES_M$, as follows:

1. Every constructor symbol of zero arity in CON_T is a term of type T.
2. Every variable in V_T is a term of type T (recall that V_T is the set of variables available for T).
3. If f is a function symbol in Σ_M (constructor, selector, or procedure) with type signature $T_1 \times \cdots \times T_n \longrightarrow T$, where T, $T_i \in TYPES_M$ for $i = 1, \ldots, n$, and $t_1, \ldots, t_n$ are terms of type $T_1, \ldots, T_n$, respectively, then $f(t_1, \ldots, t_n)$ is a term of type T.
4. Nothing else is a term of type T.

For instance, if M is as described in the last example and if

$$V_{\texttt{NatNum}} = \{n, m, n_1, m_1, \ldots\} \quad \text{and} \quad V_{\texttt{NatList}} = \{l, l_1, l_2, \ldots\},$$

then the terms (a)–(f) in the following list are of type `NatNum`, while terms (g)–(j) are of type `NatList`:

(a) `0`

(b) m

(c) `s(s(0))`

(d) `Times(s(`n`),`m`)`

(e) `Plus(s(0) Times(`m_2`,0))`

(f) `Minus(Plus(s(0),s(0)), Plus(0,s(0)))`

(g) `empty`

(h) `add(`n`,`l_7`)`

(i) l

(j) `Append(add(0,empty),empty)`.

As usual, terms that contain no variables will be called **ground terms**. In the preceding example only (a), (c), (f), (g), and (j) are ground terms. Furthermore, we say that a term t built from function symbols in Σ_M and variables in V_M is **well-typed** if t is of type T for some $T \in TYPES_M$. In our current example terms (a)–(j) are all well-typed, whereas terms such as `s(empty)` or `Append(0,`l`)` are obviously not.

With these definitions as background, we write

- $\mathcal{T}(CON_M)$ for the well-typed terms that are built exclusively from the constructor symbols in CON_M. Intuitively, these are ground terms representing the "objects" that exist in the universe of M (see section B). Terms (a), (c), and (g) in the previous list are examples.

- $\mathcal{T}(CON_M^T)$ for objects of type T, i.e. for the set $\{t \in \mathcal{T}(CON_M) \mid t \text{ is of type } T\}$. Thus

$$\mathcal{T}(CON_M) = \bigcup_{T \in TYPES_M} \mathcal{T}(CON_M^T).$$

  We will continue to use the expressions "an object $w \in T$" and "a term $w \in \mathcal{T}(CON_M^T)$" interchangeably.

- $\mathcal{T}(\Sigma_M)$ for the well-typed terms built from function symbols in Σ_M, i.e.,

$$\mathcal{T}(\Sigma_M) = \{\textit{ground} \text{ terms of type } T \mid T \in TYPES_M\}.$$

  All ground terms in the list (a)–(j) are examples.

- $\mathcal{T}(\Sigma_M, V_M)$ for the set of all well-typed terms built from function symbols in Σ_M and variables in V_M, i.e. $\mathcal{T}(\Sigma_M, V_M) = \{ \text{ terms of type } T \mid T \in TYPES_M\}$. Note that $\mathcal{T}(CON_M) \subseteq \mathcal{T}(\Sigma_M) \subseteq \mathcal{T}(\Sigma_M, V_M)$.

2.3.2 Admissibility

Every time that we extend a module by defining a new data type or a new procedure we must check that the new module thus obtained is "admissible" — for instance, we must check that the new definition is well-formed with respect to the previous definitions. The precise requirements for admissibility are laid down as follows. First, we say that a module is **well-formed** if

(i) every data type defined in it is well-formed with respect to the data types previously defined in it, and

(ii) every procedure defined in it is well-formed with respect to the data types and procedures previously defined in it.

Likewise, a module is **well-typed** if every data type and every procedure defined in it is well-typed with respect to the data types and procedures previously defined in it. Finally, we say that a module M is **admissible** if

1. M is well-formed and well-typed, and
2. every procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ defined in M is **terminating,** i.e. it halts for every possible sequence of arguments $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$.

In what follows we will be concerned with the last of the above requirements, i.e. with verifying that f terminates. The other conditions are very easy to check. We will briefly discuss each of them in the remainder of this section.

Checking that a new definition is well-formed is a trivial syntactic test. Checking that a data type definition is well-typed is also trivial. Checking that a procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ with body

$$[<\mathtt{E_1}, \mathtt{r_1}>, \ldots, <\mathtt{E_m}, \mathtt{r_m}>, <\mathtt{True}, \mathtt{r_{m+1}}>]$$

is well-typed involves (a) verifying that all $k+1$ types $T_{par_1}, \ldots, T_{par_k}, T$ have already been defined in M, and (b) verifying that each $\mathtt{r_i}$ is a term of type T, $\mathtt{i} = \mathtt{1}, \ldots, \mathtt{m}+\mathtt{1}$. Part (a) is a trivial look-up. For part (b) we give an algorithm $Type_M$ that will return the type of any given term t *if* t is well-typed in M, and null otherwise. Thus we can also use this algorithm to check whether an arbitrary term t is well-typed in M: simply compute $Type_M(t)$ and verify that the result is not null.

```
FUNCTION $Type_M(t)$ {
1. IF $(\#(t) = 0)$ THEN
2.     Return $(Top(t) \in CON_T$ for some $T \in TYPES_M)?\ T\ :$ null[9];
3. IF $(Top(t) \in \Sigma_M$ and has signature $T_1 \times \cdots \times T_n \longrightarrow T)$ THEN
4.     IF $(\#(t) \neq n)$ THEN
5.         Return null;
6.     ELSE {
7.         FOR $i = 1$ TO $n$ DO
8.             IF $(Type_M(t \downarrow i) \neq T_i)$ THEN
9.                 Return null;
10.        Return $T$; }
12. ELSE
13.     Return null; }
```

From here on we will only be concerned with well-formed and well-typed modules, so the term "module" will be used synonymously with "well-formed and well-typed module".

2.4 The Operational Semantics of IFL

In this section we present a Plotkin–style structured operational semantics (SOS) for IFL. In IFL, as in most functional languages, the imperative concept of executing a program is replaced by that of evaluating an expression. Thus, in what follows the "program" to be "executed" will simply be a term $u \in \mathcal{T}(\Sigma_M)$ such as `Fact(Fib(`s^5`(0)))` or `Reverse(add(0,add(s(0),empty)))`. Evaluation amounts to rewriting u in accordance with the definitions of the various procedures in M and

[9]The expression $(E?\ v_1 : v_2)$ (from C++) returns v_1 if the Boolean expression E is true and v_2 otherwise.

the conventions regarding constructors and selectors. The rewriting continues until we have obtained an object, i.e. a term $w \in \mathcal{T}(CON_M)$, which is then interpreted as "the answer" to the original input term u (so, for the two sample terms given above, the "answers" are the objects $\mathtt{s}^{120}(\mathtt{0})$ and $\mathtt{add}(\mathtt{s}(\mathtt{0}), \mathtt{add}(\mathtt{0}, \mathtt{empty}))$, respectively). Of course the rewriting might continue *ad infinitum* if some procedure(s) in M are not terminating.

More formally, we specify the SOS of IFL as a six-tuple

$$< \mathcal{C}, \Longrightarrow_{\mathcal{T}}, \Longrightarrow_{\mathcal{B}}, \mathcal{F}, \mathcal{I}, \mathcal{O} >$$

where

- $\mathcal{C}$, the set of **configurations** for the abstract SOS machine, is simply the set $\mathcal{T}(\Sigma_M)$.
- $\mathcal{F}$, the set of **final configurations** (a subset of $\mathcal{C}$), is $\mathcal{T}(CON_M)$.
- $\mathcal{I}$, the **input funtion** that maps an input term $u \in \mathcal{T}(\Sigma_M)$ to an **initial configuration** in $\mathcal{C}$, is simply the identity function.
- $\mathcal{O}$, the **output function** that maps a **final configuration** in $\mathcal{F}$ to an "answer" in $\mathcal{T}(CON_M)$, is again the identity function.

Transitions from one configuration $c \in \mathcal{C}$ to another configuration $c' \in \mathcal{C}$ (in our case from one term $u \in \mathcal{T}(\Sigma_M)$ to another term $u' \in \mathcal{T}(\Sigma_M)$) are specified by the **term transition relation** $\Longrightarrow_{\mathcal{T}}$, a binary relation on $\mathcal{T}(\Sigma_M)$. This relation is defined in a mutually recursive manner in terms of the **Boolean expression transition relation** $\Longrightarrow_{\mathcal{B}}$, which is a binary relation on $\mathcal{B}(\Sigma_M)$, the set of all Boolean expressions on Σ_M.

Figures 1.3 and 2-2 list the axioms and rules of inference that define the relations $\Longrightarrow_{\mathcal{T}}$ and $\Longrightarrow_{\mathcal{B}}$, respectively. We use the symbols u, w, f, con, sel, g, and E, with or without subscripts and/or superscripts, as domain variables ranging over $\mathcal{T}(\Sigma_M)$, $\mathcal{T}(CON_M)$, $PROC_M$, CON_M, SEL_M, Σ_M, and $\mathcal{B}(\Sigma_M)$, respectively. In virtue of these variables, every transition shown in the two tables acts as a **transition pattern**, i.e. as a schema that stands for a (usually infinite) class of transitions. For example, the axiom **True** $\vee E \Longrightarrow_{\mathcal{B}}$ **True** specifies an infinite set of pairs $< E_1$,**True**$>$ in which E_1 "matches" the pattern **True** $\vee E$; for instance, E_1 could be

$$\textbf{True} \vee (\texttt{pred(s(0))} \neq \texttt{Fact(s}^8\texttt{(0))}) \text{ , or}$$

$$\textbf{True} \vee (\textbf{False} \wedge [\texttt{pred(pred(0))} = \texttt{head(empty)}]).$$

Also note that several axioms and inference rules have **constraints** that serve to delimit the set of allowable transitions specified by the axiom/rule in question. For instance, in the axiom $(w_1 = w_2) \Longrightarrow_{\mathcal{B}}$ **False** the constraint $w_1 \neq w_2$ enforces the intended meaning by disallowing transitions such as $(\texttt{0} = \texttt{0}) \Longrightarrow_{\mathcal{B}}$ **False**.

The following points are noteworthy:

- None of the axioms/rules defining $\Longrightarrow_{\mathcal{T}}$ apply to objects, i.e. to terms $w \in \mathcal{T}(CON_M)$. Therefore, objects are "minimal" elements of the $\Longrightarrow_{\mathcal{T}}$ relation: for any object w, there is no term $t \in \mathcal{T}(\Sigma_M)$ such that $w \Longrightarrow_{\mathcal{T}} t$. Intuitively, that is because objects are interpreted as answers; once an answer has been obtained, there is "nowhere farther" to go.

- Rule [T1], which we shall call the **term evaluation progress rule**, forces the evaluation of a term u to proceed from left to right, in accordance with the standard well-ordering of u. That is to say, in the process of evaluating u, a transition $g(\cdots t \cdots) \Longrightarrow_{\mathcal{T}} g(\cdots t' \cdots)$ can occur only if *all* proper subterms of $g(\cdots t \cdots)$ to the left of t are objects.

- Evaluation of Boolean expressions also proceeds from left to right, as enforced by the "progress rules" [B1], [B2], and [B7]. Moreover, truth evaluation is *non-strict* (rules [B8] and [B10]), i.e. it stops as early as possible. Hence it is conceivable that a condition in the body of some procedure evaluates to **True** or **False** even though it contains terms that diverge.

$$\frac{u_1 \Longrightarrow_{\mathcal{T}} u'_1}{g(w_1,\ldots,w_n,u_1,u_2,\ldots,u_m) \Longrightarrow_{\mathcal{T}} g(w_1,\ldots,w_n,u'_1,u_2,\ldots,u_m)} \quad [T1]$$

$$sel^i_{con}(con(w_1,\ldots,w_n)) \Longrightarrow_{\mathcal{T}} w_i \quad [T2]$$
for any $i \in \{1,\ldots,n\}$

$$sel^i_{con_1}(con_2(w_1,\ldots,w_n)) \Longrightarrow_{\mathcal{T}} Minimal_M(T_i) \quad [T3]$$
where $con_1 \not\equiv con_2, T_i$ is the i^{th} component type of con_1, $Minimal_M(T_i)$ is the minimal object of that type, and $1 \leq i \leq \#(con_1)$.

$$\frac{\begin{array}{c}\mathtt{E}_1[par_1 \mapsto w_1,\ldots,par_k \mapsto w_k] \Longrightarrow^*_{\mathcal{B}} \textbf{False}\\ \vdots \\ \mathtt{E}_\mathtt{j}[par_1 \mapsto w_1,\ldots,par_k \mapsto w_k] \Longrightarrow^*_{\mathcal{B}} \textbf{False}\\ \mathtt{E}_{\mathtt{j}+1}[par_1 \mapsto w_1,\ldots,par_k \mapsto w_k] \Longrightarrow^*_{\mathcal{B}} \textbf{True}\end{array}}{f(w_1,\ldots,w_k) \Longrightarrow_{\mathcal{T}} \mathtt{r}_{\mathtt{j}+1}[par_1 \mapsto w_1,\ldots,par_k \mapsto w_k]} \quad [T4]$$
where $f \in PROC_M$ has k parameters $par_1,\ldots,par_k$ and body $[<\mathtt{E}_1,\mathtt{r}_1>,\ldots,<\mathtt{E}_\mathtt{m},\mathtt{r}_\mathtt{m}>,<\mathtt{True},\mathtt{r}_{\mathtt{m}+1}>]$, while $0 \leq \mathtt{j} \leq \mathtt{m}$.

Figure 2-1: The definition of the term transition relation $\Longrightarrow_{\mathcal{T}}$.

- We can now formally define a procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ as **terminating** iff for all objects $w_1 \in T_{par_1},\ldots,w_k \in T_{par_k}$ there is an object $w \in T$ such that $f(w_1,\ldots,w_k) \Longrightarrow^*_{\mathcal{T}} w$.

A simple example of the above SOS "in action" is provided in Figure 2-4, which illustrates the evaluation of the term `Plus(s(s(0)),s(s(0)))` (based on the definition of `Plus` given in section 2.2). We can see from steps 7,10,18,22, and 28 that

$$\mathtt{Plus(s(s(0)),s(s(0)))} \Longrightarrow^*_{\mathcal{T}} \mathtt{s(s(s(s(0))))},$$

thus "the answer" to `Plus(s(s(0)),s(s(0)))` is the object `s(s(s(s(0))))`.

Lastly, for any s and t in $\mathcal{T}(\Sigma_M)$ such that $s \Longrightarrow_{\mathcal{T}} t$, we define $C(s,t)$, the **rewrite**

$$\frac{u_1 \Longrightarrow_{\mathcal{T}} u_1'}{(u_1 \; op \; u_2) \Longrightarrow_{\mathcal{B}} (u_1' \; op \; u_2) \quad \text{for } op \in \{=, \neq\}} \qquad [B1]$$

$$\frac{u \Longrightarrow_{\mathcal{T}} u'}{(w \; op \; u) \Longrightarrow_{\mathcal{B}} (w \; op \; u') \quad \text{for } op \in \{=, \neq\}} \qquad [B2]$$

$$(w_1 = w_2) \Longrightarrow_{\mathcal{B}} \textbf{True} \quad \text{for } w_1 \equiv w_2 \qquad [B3]$$

$$(w_1 = w_2) \Longrightarrow_{\mathcal{B}} \textbf{False} \quad \text{for } w_1 \not\equiv w_2 \qquad [B4]$$

$$(w_1 \neq w_2) \Longrightarrow_{\mathcal{B}} \textbf{True} \quad \text{for } w_1 \not\equiv w_2 \qquad [B5]$$

$$(w_1 \neq w_2) \Longrightarrow_{\mathcal{B}} \textbf{False} \quad \text{for } w_1 \equiv w_2 \qquad [B6]$$

$$\frac{E_1 \Longrightarrow_{\mathcal{B}} E_1'}{E_1 \; op \; E_2 \Longrightarrow_{\mathcal{B}} E_1' \; op \; E_2 \quad \text{for } op \in \{\wedge, \vee\}} \qquad [B7]$$

$$\textbf{True} \vee E \Longrightarrow_{\mathcal{B}} \textbf{True} \qquad [B8]$$

$$\textbf{False} \vee E \Longrightarrow_{\mathcal{B}} E \qquad [B9]$$

$$\textbf{False} \wedge E \Longrightarrow_{\mathcal{B}} \textbf{False} \qquad [B10]$$

$$\textbf{True} \wedge E \Longrightarrow_{\mathcal{B}} E \qquad [B11]$$

Figure 2-2: The definition of the Boolean expression transition relation ($\Longrightarrow_{\mathcal{B}}$).

$$\boxed{\begin{array}{ll} E \Longrightarrow_{\mathcal{B}}^{*} E & [\text{B12}] \\ \\ \dfrac{\begin{array}{l} E_1 \Longrightarrow_{\mathcal{B}}^{*} E_2 \\ E_2 \Longrightarrow_{\mathcal{B}} E_3 \end{array}}{E_1 \Longrightarrow_{\mathcal{B}}^{*} E_3} & [\text{B13}] \end{array}}$$

Figure 2-3: The definition of the reflexive and transitive closure of $\Longrightarrow_{\mathcal{B}}$.

cost of the transition $s \Longrightarrow_{\mathcal{T}} t$, as follows:

1. If $s \Longrightarrow_{\mathcal{T}} t$ by virtue of rule [T2] or [T3], then $C(s,t) = 1$.

2. If $s \Longrightarrow_{\mathcal{T}} t$ by virtue of [T1], then $C(s,t) = 1 + C(u_1, u_1')$.

3. Finally, if $s \Longrightarrow_{\mathcal{T}} t$ by virtue of rule [T4], then $s \equiv f(w_1, \ldots, w_k)$ for some $f \in PROC_M$ with k parameters $par_1, \ldots, par_k$ and body

$$[<\mathtt{E_1}, \mathtt{r_1}>, \ldots, <\mathtt{E_m}, \mathtt{r_m}>, <\mathtt{True}, \mathtt{r_{m+1}}>]$$

and, for some $\mathtt{j} \leq \mathtt{m}$, there are $\mathtt{j}+1$ paths of the form

$$\mathtt{E_1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv E_1^1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} E_1^2[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} \cdots$$
$$\cdots \Longrightarrow_{\mathcal{B}} E_1^{n_1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} E_1^{n_1+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \textbf{False}$$
$$\vdots$$
$$\mathtt{E_j}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv E_j^1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} E_j^2[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} \cdots$$
$$\cdots \Longrightarrow_{\mathcal{B}} E_j^{n_j}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} E_j^{n_j+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \textbf{False}$$
$$\mathtt{E_{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv E_{j+1}^1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} E_{j+1}^2[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} \cdots$$
$$\cdots \Longrightarrow_{\mathcal{B}} E_{j+1}^{n_{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}} E_{j+1}^{n_{j+1}+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \textbf{True}$$

In that case we let

$$C(s,t) = 1 + \sum_{i=1}^{j+1} \left[\sum_{\gamma=1}^{n_i} BC(E_i^{\gamma}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}], E_i^{\gamma+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) \right]$$

where, for any two Boolean expressions E and E' over Σ_M such that $E \Longrightarrow_{\mathcal{B}} E'$, we define $BC(E, E')$, **the rewrite cost of the Boolean transition** $E \Longrightarrow_{\mathcal{B}} E'$

#	*Transition*	*Justification*
1	s(s(0)) = 0 $\Longrightarrow^*_{\mathcal{B}}$ s(s(0)) = 0	[B12]
2	s(s(0)) = 0 $\Longrightarrow_{\mathcal{B}}$ **False**	[B4]
3	s(s(0)) = 0 $\Longrightarrow^*_{\mathcal{B}}$ **False**	1,2,[B13]
4	s(s(0)) $\neq$ 0 $\Longrightarrow^*_{\mathcal{B}}$ s(s(0)) $\neq$ 0	[B12]
5	s(s(0)) $\neq$ 0 $\Longrightarrow_{\mathcal{B}}$ **True**	[B5]
6	s(s(0)) $\neq$ 0 $\Longrightarrow^*_{\mathcal{B}}$ **True**	4,5,[B13]
7	Plus(s(s(0)),s(s(0))) $\Longrightarrow_{\mathcal{T}}$ s(Plus(s(s(0)),pred(s(s(0)))))	3,6,[T3], for $j = 1$
8	pred(s(s(0))) $\Longrightarrow_{\mathcal{T}}$ s(0)	[T2]
9	Plus(s(s(0)),pred(s(s(0)))) $\Longrightarrow_{\mathcal{T}}$ Plus(s(s(0)),s(0))	8,[T1]
10	s(Plus(s(s(0)),pred(s(s(0))))) $\Longrightarrow_{\mathcal{T}}$ s(Plus(s(s(0)),s(0)))	9,[T1]
11	s(0) = 0 $\Longrightarrow^*_{\mathcal{B}}$ s(0) = 0	[B12]
12	s(0) = 0 $\Longrightarrow_{\mathcal{B}}$ **False**	[B4]
13	s(0) = 0 $\Longrightarrow^*_{\mathcal{B}}$ **False**	11,12,[B13]
14	s(0) $\neq$ 0 $\Longrightarrow^*_{\mathcal{B}}$ s(0) $\neq$ 0	[B12]
15	s(0) $\neq$ 0 $\Longrightarrow_{\mathcal{B}}$ **True**	[B5]
16	s(0) $\neq$ 0 $\Longrightarrow^*_{\mathcal{B}}$ **True**	14,15,[B13]
17	Plus(s(s(0)),s(0)) $\Longrightarrow_{\mathcal{T}}$ s(Plus(s(s(0)),pred(s(0))))	13,16,[T3], for j=1
18	s(Plus(s(s(0)),s(0))) $\Longrightarrow_{\mathcal{T}}$ s(s(Plus(s(s(0)),pred(s(0)))))	17,[T1]
19	pred(s(0)) $\Longrightarrow_{\mathcal{T}}$ 0	[T2]
20	Plus(s(s(0)),pred(s(0))) $\Longrightarrow_{\mathcal{T}}$ Plus(s(s(0)),0)	19,[T1]
21	s(Plus(s(s(0)),pred(s(0)))) $\Longrightarrow_{\mathcal{T}}$ s(Plus(s(s(0)),0))	20,[T1]
22	s(s(Plus(s(s(0)),pred(s(0))))) $\Longrightarrow_{\mathcal{T}}$ s(s(Plus(s(s(0)),0)))	21,[T1]
23	0 = 0 $\Longrightarrow^*_{\mathcal{B}}$ 0 = 0	[B12]
24	0 = 0 $\Longrightarrow_{\mathcal{B}}$ **True**	[B3]
25	0 = 0 $\Longrightarrow^*_{\mathcal{B}}$ **True**	23,24,[B13]
26	Plus(s(s(0)),0) $\Longrightarrow_{\mathcal{T}}$ s(s(0))	25,[T3], for $j = 0$
27	s(Plus(s(s(0)),0)) $\Longrightarrow_{\mathcal{T}}$ s(s(s(0)))	26,[T1]
28	s(s(Plus(s(s(0)),0))) $\Longrightarrow_{\mathcal{T}}$ s(s(s(s(0))))	27,[T1]

Figure 2-4: A transition sequence for the term `Plus(s(s(0)),s(s(0)))`.

as

- $C(u_1, u_1')$ or $C(u, u')$ if $E \Longrightarrow_{\mathcal{B}} E'$ by virtue of [B1] or [B2], respectively,
- $BC(E_1, E_1')$ if $E \Longrightarrow_{\mathcal{B}} E'$ by virtue of [B7],
- 0 if $E \Longrightarrow_{\mathcal{B}} E'$ by virtue of any other rule.

In other words we stipulate that once all the terms occuring in a Boolean expression E have been reduced to objects in $\mathcal{T}(CON_M)$, computing the truth value of E is "free", i.e. rules [B3]–[B6] and [B8]–[B11] are applied at "zero cost".

A few examples: the transition

$$\texttt{s(Plus(s(s(0)),pred(s(s(0)))))} \Longrightarrow_{\mathcal{T}} \texttt{s(Plus(s(s(0)),s(0)))}$$

(transition #10 in Fig. 2-4) has rewrite cost 3; intuitively, that accounts for the three transitions #8,#9, and #10. Transition #7 in the same figure has cost 1 (as no additional cost is incurred by evaluating the Boolean expressions in the body of `Plus`), while transition #18 has cost 2. Finally, the rewrite cost of a transition path $s_1 \Longrightarrow_{\mathcal{T}} s_2 \Longrightarrow_{\mathcal{T}} \cdots \Longrightarrow_{\mathcal{T}} s_n \Longrightarrow_{\mathcal{T}} s_{n+1}$ is defined as the sum of the costs of the individual transitions comprising the path, namely $\sum_{i=1}^{n} C(s_i, s_{i+1})$. So, for instance, the rewrite cost of the transition path [#7,#10,#18] in Fig 2-4 is $1 + 3 + 2 = 6$.

2.5 An interpreter for IFL

It is a simple exercise to write an interpreter for IFL that works in the typical Read–Eval–Print fashion. The procedure $Eval_M$ given below takes a term $t \in \mathcal{T}(\Sigma_M)$ and attempts to map t to some object in $\mathcal{T}(CON_M)$ by essentially rewriting it in accordance with the transition relations $\Longrightarrow_{\mathcal{T}}$ and $\Longrightarrow_{\mathcal{B}}$ of the SOS of IFL. For instance, if M comprises the data types and procedures given in sections 2.1 and 2.2, then

$$Eval_M(\texttt{Reverse(add(s}^{15}\texttt{(0),add(Fact(s}^{3}\texttt{(0)),empty))))} = \texttt{add(s}^{6}\texttt{(0),add(s}^{15}\texttt{(0),empty))}$$

$Eval_M$(Length(Append(InOrder(tree(s^7(0), tree(s^2(0), null, null), tree(Fibonacci(s^3(0), null, null))), add(s^{18}(0), empty))))) = s^4(0), etc.

Of course in any practical implementation we would precede the call to $Eval_M(t)$ [10] with a call to $Type_M(t)$ in order to ensure that t is well-typed. Or better yet, type checking and evaluation could proceed simultaneously in order to avoid multiple recursive descents on t. At any rate, the procedure we give below assumes that the input t is a well-typed term and performs a three-way switch on the top function symbol of t, which could denote either a constructor, or a selector, or a user-defined procedure.

```
FUNCTION $Eval_M(t : \mathcal{T}(\Sigma_M))$ {
1.  $\sigma \longleftarrow Top(t)$;
2.  IF $\sigma \in CON_M$ THEN      // Case 1: $\sigma \in CON_M$
3.      Return  $(\#(t) = 0)?\ \sigma\ :\ \sigma(Eval_M(t \downarrow 1), \ldots, Eval_M(t \downarrow \#(t)))$;
4.  ELSE      // Case 2: $\sigma \in SEL_M$
5.      IF $\sigma \equiv sel_i^c$ for some $c : T_1 \times \cdots \times T_n \longrightarrow T$ in $CON_M$ THEN {
6.          $sel\_arg \longleftarrow Eval_M(t \downarrow 1)$;
7.          Return  $(Top(sel\_arg) \equiv c)?\ (sel\_arg \downarrow i)\ :\ Minimal_M(T_i)$;   }
8.      ELSE      // Case 3: $\sigma \in PROC_M$
9.          IF  $\sigma \equiv f$ for some $f \in PROC_M$ with $k$ parameters $par_1, \ldots, par_k$
            and body $[<\mathtt{E}_1, \mathtt{r}_1>, \ldots, <\mathtt{E}_m, \mathtt{r}_m>, <\mathtt{True}, \mathtt{r}_{m+1}>]$ THEN {
10.             FOR  $i = 1$ TO  $k$ DO
11.                 $w_i \longleftarrow Eval_M(t \downarrow i)$;
12.             FOR  $\mathtt{j} = 1$ TO $\mathtt{m} + 1$ DO
13.                 IF $(TEval_M(\mathtt{E_j}[par_1 \mapsto w_1, \ldots, par_k \mapsto w_k]))$ THEN
14.                     Return $Eval_M(\mathtt{r_j}[par_1 \mapsto w_1, \ldots, par_k \mapsto w_k])$;  }   }
```

where

(1) the value $TEval_M(E)$, for any $E \in \mathcal{B}(\Sigma_M)$, is defined inductively as follows:

[10]To be read "Evaluate t in M".

- If $E \in \{$ **True,False** $\}$, then $TEval_M(E) = E$.
- If $E \equiv (s = t)$, for any $s, t \in \mathcal{T}(\Sigma_M)$, then

$$TEval_M(E) = \begin{cases} \textbf{True} & \text{if } Eval_M(s) = Eval_M(t) \\ \textbf{False} & \text{otherwise.} \end{cases}$$

  This clause is where the mutual recursiveness of $\Longrightarrow_{\mathcal{T}}$ and $\Longrightarrow_{\mathcal{B}}$ enters the picture.

- If $E \equiv (\neg E_1)$, then $TEval_M(E) =$ **True** if $TEval_M(E_1) =$ **False**, while $TEval_M(E) =$ **False** if $TEval_M(E_1) =$ **True**.
- If $E \equiv (E_1 \ op \ E_2)$, for $op \in \{\wedge, \vee\}$, then

$$TEval_M(E) = TEval_M(E_1) \ op \ TEval_M(E_2).$$

And,
(2) $Minimal_M(T_i)$ returns the minimal object of type T_i (as defined in section 2.1). This is trivial and can be done in time linear in the number of type definitions in M (and independently of the length of those definitions).

Just as they are $\Longrightarrow_{\mathcal{T}}$–minimal elements, objects are also fixed points of $Eval_M$. In particular, the following is easily proved by strong induction on the size of w:

Lemma 2.5.1 *For all objects $w \in \mathcal{T}(\mathrm{CON}_M)$, $Eval_M(w) = w$.*

Another simple observation about the interpreter is that it "preserves type":

Lemma 2.5.2 *For all $t \in \mathcal{T}(\Sigma_M)$, $Type_M(Eval_M(t)) = Type_M(t)$.*

It is clear that running $Eval_M$ on a term t might fail to return an answer if M is not admissible. In particular, $Eval_M$ might get into an infinite loop if some of the procedures in M are not terminating. We will write $Eval_M(t) \uparrow$ to indicate that the computation of $Eval_M(t)$ diverges, and $Eval_M(t) \downarrow$ for the opposite, i.e. to indicate that $Eval_M$ halts on input t. Likewise, we will write $TEval_M(E) \uparrow$ or $TEval_M(E) \downarrow$ for an expression $E \in \mathcal{B}(\Sigma_M)$ to indicate that the evaluation of the truth value of E diverges or converges, respectively. Clearly, $TEval_M(E) \downarrow$ iff $Eval_M(t) \downarrow$ for every term t occuring in E whose evaluation is necessary in computing the truth value of E.

On the other hand, if $Eval_M(t)$ does return an answer, that answer is guaranteed to be an object $w \in \mathcal{T}(CON_M)$. Furthermore, w will be such that $t \Longrightarrow_{\mathcal{T}}^{*} w$. And conversely, if w is an object in $\mathcal{T}(CON_M)$ such that $t \Longrightarrow_{\mathcal{T}}^{*} w$, then we will have $Eval_M(t) = w$:

Theorem 2.5.1 (Interpreter completeness) *For any module M and term $t \in \mathcal{T}(\Sigma_M)$, $Eval_M(t)$ returns a term w iff w is an object in $\mathcal{T}(\mathrm{CON}_M)$ such that $t \Longrightarrow_{\mathcal{T}}^{*} w$.*

A proof of this can be found in Appendix A.

We can now easily prove the analogous completeness result for $TEval_M$:

Theorem 2.5.2 *For all $E \in \mathcal{B}(\Sigma_M)$, $TEval_M(E)$ returns a value TV iff $TV \in$ {**True**,**False**} and $E \Longrightarrow_{\mathcal{B}}^{*} TV$.*

The following corollary is immediate:

Corollary 2.5.1 *For any module M and term $t \in \mathcal{T}(\Sigma_M)$, $Eval_M(t) \downarrow$ iff there is an object $w \in \mathcal{T}(\mathrm{CON}_M)$ such that $t \Longrightarrow_{\mathcal{T}}^{*} w = Eval_M(t)$.*

The above corollary provides us with an alternative (and perhaps more intuitive) characterization of termination that we will be using in the sequel:

Proposition 2.5.1 *A procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ in a module M is terminating iff $Eval_M(f(w_1, \ldots, w_k)) \downarrow$ for all objects $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$.*

Proof. Suppose f is terminating, so that for all $w_1 \in T_{par_1}, \ldots, w_{par_k} \in T_k$, there is a $w \in T$ such that $f(w_1, \ldots, w_k) \Longrightarrow_{\mathcal{T}}^{*} w$; then, by the preceding corollary, we also have $Eval_M(f(w_1, \ldots, w_k)) \downarrow$. Conversely, suppose that for all $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$, we have $Eval_M(f(w_1, \ldots, w_k)) \downarrow$. Then, again by corollary 2.5.1, for any $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$, there will be an object w such that $f(w_1, \ldots, w_k) \Longrightarrow_{\mathcal{T}}^{*} w$, hence f is terminating. □

It follows that if M is an admissible module, $Eval_M$ ($TEval_M$) computes a total function from $\mathcal{T}(\Sigma_M)$ ($\mathcal{B}(\Sigma_M)$) to $\mathcal{T}(CON_M)$ ({**True,False**}). We state this in the following theorem, which is easily proved by induction.

Theorem 2.5.3 *If M is an admissible module then for all $t \in \mathcal{T}(\Sigma_M)$, $Eval_M(t)$ halts and returns an object w such that $t \Longrightarrow_{\mathcal{T}}^{*} w$. Similarly, for all $E \in \mathcal{B}(\Sigma_M)$, $TEval_M(E)$ halts and returns a truth value TV such that $E \Longrightarrow_{\mathcal{B}}^{*} TV$. Thus, for admissible modules M, $Eval_M$ and $TEval_M$ compute total functions from $\mathcal{T}(\Sigma_M)$ and $\mathcal{B}(\Sigma_M)$ to $\mathcal{T}(\mathrm{CON}_M)$ and* {**True,False**}*, respectively.*

Since the interpreter $Eval_M$ will be used extensively in the remainder of this paper, it will be expedient at this point to analyze its behavior a little further and derive a few more key conclusions for future reference. We begin by observing that $Eval_M$ attempts to work its way down its argument in accordance with the standard well-ordering of terms, as the latter is embodied in the progress rules of the SOS of IFL. Thus the call $Eval_M(t)$ will initiate a depth-first traversal of t, during which $Eval_M$ will be recursively invoked for every "visited" node $r \sqsubset t$; the result of each such intermediate invocation will be used subsequently in evaluating the parent term of r.

We say that $Eval_M$ "attempts" to perform a recursive descent on its argument because it might in fact not complete the traversal. For instance, if $Eval_M(t \downarrow 1) \uparrow$ (say, because $Top(t \downarrow 1)$ stands for a nowhere-terminating procedure), then $Eval_M$ will never "reach" $t \downarrow 2$. Let us pursue this idea a little further. Let us write $Eval_M(s) \Longrightarrow^{+} Eval_M(t)$ to indicate that calling $Eval_M$ on s will eventually (after a finite time period) result in a recursive invocation of $Eval_M$ on t; in other words, $Eval_M(s) \Longrightarrow^{+} Eval_M(t)$ says that the call $Eval_M(s)$ must eventually spawn the call $Eval_M(t)$. For example, for any non-leaf term $t \in \mathcal{T}(\Sigma_M)$ we always have $Eval_M(t) \Longrightarrow^{+} Eval_M(t \downarrow 1)$, as the very first thing that $Eval_M$ does after it has classified $Top(t)$ (as a constructor, selector, or procedure) is to recursively call itself on $t \downarrow 1$. However, as we indicated above, we do not necessarily have $Eval_M(t) \Longrightarrow^{+} Eval_M(r)$ for every term $r \sqsubset t$, as the interpreter might never get to certain subterms of t if it gets stuck at some prior point[11]. To reiterate the last example, if $Eval_M(t_1) \uparrow$ then $Eval_M(f(t_1, t_2)) \not\Longrightarrow^+ Eval_M(t_2)$, as the divergence of t_1 will prevent $Eval_M$ from ever getting to $t \downarrow 2$.

Nevertheless, we are guaranteed to have $Eval_M(t) \Longrightarrow^+ Eval_M(r)$ for some $r \sqsubset t$ *if* $Eval_M$ halts for all proper subterms of t to the left of r. This is more accurately stated and proved by way of lemma 2.5.3, but before we get there it might be instructive to spell out formally what we mean when we write $Eval_M(t_1) \Longrightarrow^+ Eval_M(t_2)$, as the description that was offered earlier hinged on non-extensional terms such as "results in", "must", etc.

Let us begin by defining a binary relation $\mathcal{S}$ [12] on $\mathcal{T}(\Sigma_M)$ as follows:

- For any term $t \equiv g(t_1, \ldots, t_n), g \in \Sigma_M$, and any $j \in \{1, \ldots, n\}$, if $Eval_M(t_i) \downarrow$ for every $i = 1, \ldots, j-1$, then $t \; \mathcal{S} \; t_j$ (so that we always vacuously have $g(t_1, \ldots, t_n) \; \mathcal{S} \; t_1$).

- For any term $t \equiv f(t_1, \ldots, t_k)$, where f is a procedure in M with k parameters $par_1, \ldots, par_k$ and body

$$[<\mathtt{E}_1, \mathtt{r}_1>, \ldots, <\mathtt{E}_\mathtt{m}, \mathtt{r}_\mathtt{m}>, <\mathtt{True}, \mathtt{r}_{\mathtt{m}+1}>],$$

  if there are k objects $w_1, \ldots, w_k$ such that

$$f(t_1, \ldots, t_k) \Longrightarrow_{\mathcal{T}}^* f(w_1, \ldots, w_k)$$

  and an integer $\mathtt{j} \in \{\mathtt{0}, \ldots, \mathtt{m}\}$ such that

$$f(w_1, \ldots, w_k) \Longrightarrow_{\mathcal{T}} \mathtt{r}_{\mathtt{j}+1}[\overrightarrow{par} \mapsto \overrightarrow{w}]$$

  then $t \; \mathcal{S} \; \mathtt{r}_{\mathtt{j}+1}[par_1 \mapsto Eval_M(t_1), \ldots, par_k \mapsto Eval_M(t_k)]$.

We can now formally interpret the expression $Eval_M(t_1) \Longrightarrow^+ Eval_M(t_2)$ as asserting

[11] "Prior" in the sense of the standard well-ordering of terms.

[12] " $\mathcal{S}$ " stands for "spawns", so $t_1 \; \mathcal{S} \; t_2$ should be read as "the evaluation of t_1 spawns the evaluation of t_2".

that $t_1\ \mathcal{S}^+\ t_2$, where $\mathcal{S}^+$ is the transitive closure of $\mathcal{S}$. In a similar fashion, we will write $Eval_M(t_1) \Longrightarrow^* Eval_M(t_2)$ to mean $t_1\ \mathcal{S}^*\ t_2$, where $\mathcal{S}^*$ is the reflexive closure of $\mathcal{S}^+$.

We are now in a position to state and prove the following useful result:

Lemma 2.5.3 *Let M be any module (not necessarily admissible), let $t \in \mathcal{T}(\Sigma_M)$, and let s be a proper subterm of t. If $Eval_M(r) \downarrow$ for every term $r \sqsubset t$ to the left of s that is not a superterm of s, then $Eval_M(t) \Longrightarrow^* Eval_M(s)$.*

Proof. By strong induction on the size of t. Let $t_1, \ldots, t_n$ be the $n > 0$ immediate subterms of t, and let t_j be the unique child that is a superterm of s. Further, assume that $Eval_M(r) \downarrow$ for every term $r \sqsubset t$ that precedes s, and is not a superterm of s. From this assumption it follows that $Eval_M(r) \downarrow$ for all terms $r \sqsubset t_j$ that are to the left of s and do not contain it (since $r \sqsubset t_j$ implies $r \sqsubset t$). Hence, by the inductive hypothesis, we conclude that $Eval_M(t_j) \Longrightarrow^* Eval_M(s)$ (i). Now, by assumption, we have $Eval_M(t_i) \downarrow$ for every $i \in \{1, \ldots, j-1\}$ (since each such t_i precedes s and does not contain it). But then, by the definition of $\mathcal{S}$, we see that $Eval_M(t) \Longrightarrow^* Eval_M(t_j)$ (ii). Finally, (i) and (ii) together with the transitivity of $\Longrightarrow^*$ establish that $Eval_M(t) \Longrightarrow^* Eval_M(s)$, and the argument is complete. □

We end this section with two additional lemmas, both of which are straightforward and stated without proof.

Lemma 2.5.4 *Let M' be an extension of a module M, so that $\Sigma_{M'} \supset \Sigma_M$. Then,*
(a) For all terms $t \in \mathcal{T}(\Sigma_M)$, $Eval_{M'}(t) = Eval_M(t)$.
(b) Let $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ be a procedure in $PROC_{M'} - \mathrm{PROC}_M$ and let `If` $\mathtt{E_j}$ `Then` $\mathtt{r_j}$ *be a statement in the body of f such that $\mathtt{E_j} \in \mathcal{B}(\Sigma_M, V_M)$. Then, for any k objects $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$ and $TV \in \{\textbf{True,False}\}$,*

$$TEval_{M'}(\mathtt{E_j}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = TV \quad \textit{iff} \quad TEval_M(\mathtt{E_j}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = TV.$$

Lemma 2.5.5 *For any module M and term $f(t_1, \ldots, t_k) \in \mathcal{T}(\Sigma_M)$, if $Eval_M(t_i) \downarrow$ for every $i = 1, \ldots, k$ then*

$$Eval_M(f(t_1, \ldots, t_k)) \downarrow \quad \textit{iff} \quad Eval_M(f(Eval_M(t_1), \ldots, Eval_M(t_k))) \downarrow.$$

Chapter 3

Size orderings and the main termination theorem

3.1 The size of an object

Every object has a certain *size*. How we "measure" that size depends on the type of the object. For example, for objects of type `NatList` a natural measure of size is the length of the list. Thus we say that the empty list has size 0, the list $[3,1]$ has size two, and so on. Or, if we agree to let $SZ_T(w)$ denote the size of an object w of type T, we can write $SZ_{\texttt{NatList}}(\texttt{empty}) = 0$, $SZ_{\texttt{NatList}}(\texttt{add}(\texttt{s}^5(\texttt{0}), \texttt{add}(\texttt{s}(\texttt{0}), \texttt{empty}))) = 2$, etc. More formally, we have

$$SZ_{\texttt{NatList}}(l) = \begin{cases} 0 & \text{if } l \equiv \texttt{empty} \\ 1 + SZ_{\texttt{NatList}}(\texttt{tail(l)}) & \text{if } l \not\equiv \texttt{empty}. \end{cases} \tag{3.1}$$

In a similar spirit, we take the size of a `BTree` object to be the number of its nodes, so that

$$SZ_{\texttt{BTree}}(\texttt{null}) = 0,$$

$$SZ_{\texttt{BTree}}(\texttt{tree}(\texttt{s}^7(\texttt{0}), \texttt{tree}(\texttt{s}^5(\texttt{0}), \texttt{null}, \texttt{null}), \texttt{null})) = 2,$$

and so on. More precisely:

$$SZ_{\texttt{BTree}}(t) = \begin{cases} 0 & \text{if } t \equiv \texttt{null} \\ 1 + SZ_{\texttt{BTree}}(\texttt{l-branch(t)}) + & \\ \quad SZ_{\texttt{BTree}}(\texttt{r-branch(t)}) & \text{if } t \not\equiv \texttt{null}. \end{cases} \tag{3.2}$$

As a last example, we define the size of a propositional wff to be the number of propositional operators in it. Thus

$$SZ_{\texttt{Wff}}(\texttt{atomic}(\texttt{s}^{12}(0))) = 0,$$

$$SZ_{\texttt{Wff}}(\texttt{and}(\texttt{atomic}(\texttt{s}^{38}(0)), \texttt{or}(\texttt{atomic}(\texttt{s}(0)), \texttt{atomic}(\texttt{s}^{7}(0))))) = 2,$$

etc. In other words,

$$SZ_{\texttt{Wff}}(\phi) = \begin{cases} 0 & \text{if } \phi \equiv \texttt{atomic}(\cdots) \\ 1 + SZ_{\texttt{Wff}}(\texttt{neg}(\phi)) & \text{if } \phi \equiv \texttt{not}(\cdots) \\ 1 + SZ_{\texttt{Wff}}(\texttt{l-and}(\phi)) + & \\ \quad SZ_{\texttt{Wff}}(\texttt{r-and}(\phi)) & \text{if } \phi \equiv \texttt{and}(\cdots) \\ 1 + SZ_{\texttt{Wff}}(\texttt{l-or}(\phi)) + & \\ \quad SZ_{\texttt{Wff}}(\texttt{r-or}(\phi)) & \text{if } \phi \equiv \texttt{or}(\cdots) \end{cases} \tag{3.3}$$

Each of the three size measures given above applies only to objects of its own particular type, but nevertheless they all use the same essential idea: if an object is built from an *irreflexive* constructor, let its size be zero (the first clauses of definitions 3.1—3.3); if it is built from a *reflexive* constructor, let its size be *one more* than the sum of the sizes of its components (tail clauses of 3.1—3.3). This simply reflects the intuition that the irreflexive constructors of a type T produce "simple" ("atomic") objects — primitive building blocks that have no internal structure as far as T is concerned. Hence their size should be zero. By contrast, a reflexive constructor of T takes $n > 0$ objects $s_1, \ldots, s_n$ and bundles them into a new "complex" object s. The T-size of s should therefore be one plus the sum of the T-sizes of the s_i, the "one plus" signifying that we have gone "one level higher" by applying the reflexive constructor in question, and the summing of the T-sizes of the s_i signifying that each s_i is now a part of s and thus contributes to the latter's size. For instance, by joining together two `BTrees` `l` and `r` at a new root node of value `v`, we produce the more complex object `tree(v,l,r)`, which, intuitively, is one unit larger than the sizes of `l` and `r` summed together.

We abstract the preceding observations into a general definition of size for objects of a type T as follows:

$$SZ_T(s) = \begin{cases} 0 & \text{if } s \equiv ircons(\cdots) \\ 1 + \sum_{j=1}^{m} SZ_T(s \downarrow i_j) & \text{if } s \equiv rcons(\cdots). \end{cases} \tag{3.4}$$

where $ircons$ ($rcons$) is any irreflexive (reflexive) constructor of type T, and $i_1, \ldots, i_m$ are the reflexive positions of $rcons$ (so that the i_j^{th} component type of $rcons$ is T, for $j \in \{1, \ldots, m\}$).

If the type at hand is implicitly understood or can be easily deduced from the context, we may drop the subscript $_T$ and simply write $SZ(s)$.

3.2 A well-founded partial ordering of objects based on size

Now let s and t be two terms of the same type T in some admissible module M, and let $v_1, \ldots, v_k$ be an enumeration of all and only the variables that occur in s and t, where each $v_i \in V_{T_i}$. Then we will write $s \preceq t$ to indicate that for *all* objects $w_1 \in T_1, \ldots, w_k \in T_k$,

$$SZ_T(Eval_M(s[\overrightarrow{v} \mapsto \overrightarrow{w}])) \leq SZ_T(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}])). \tag{3.5}$$

Of course if s and t are ground terms then there are no variables to be replaced and $s \preceq t$ simply asserts that $SZ_T(Eval_M(s)) \leq SZ_T(Eval_M(t))$. Likewise, we will write $s \prec t$ if (3.5) holds but with strict numeric inequality $<$ in place of $\leq$. For instance, we have $l \prec$ `add`(n, l) because no matter what objects we substitute for n and l, the size of l will always be strictly less (one less, to be precise) than the size of `add`(n, l). The following list provides some additional examples:

$$\text{(i) } \mathtt{0} \prec \mathtt{s}(n)$$
$$\text{(ii) } \mathtt{pred}(\mathtt{s}(n)) \preceq n$$
$$\text{(iii) } l_2 \preceq \mathtt{Append}(l_1, l_2)$$
$$\text{(iv) } \mathtt{Reverse}(l) \preceq l$$
$$\text{(v) } \mathtt{Minus}(n_1, n_2) \preceq n_1$$
$$\text{(vi) } n \preceq \mathtt{Plus}(n, m)$$
$$\text{(vii) } n \prec \mathtt{Plus}(n, \mathtt{s}(m))$$
$$\text{(viii) } \mathtt{l\text{-}branch}(t) \preceq t$$

3.3 The GTZ predicate

If s is a term of type T containing $k > 0$ variables $v_1 \in V_{T_1}, \ldots, v_k \in V_{T_k}$, we will write $GTZ(s)$ to mean that for *all* objects $w_1 \in T_1, \ldots, w_k \in T_k$, we have

$$SZ_T(Eval_M(s[v_1 \mapsto w_1, \ldots, v_k \mapsto w_k])) > 0. \tag{3.6}$$

Again, if s is a ground term then (3.6) simply says that $SZ_T(Eval_M(s)) > 0$. For example, we have $GTZ(\mathtt{s}(\mathtt{0}))$, since $SZ(Eval_M(\mathtt{s}(\mathtt{0}))) = 1 > 0$, $GTZ(\mathtt{Fact}(\mathtt{s}(\mathtt{0})))$, $GTZ(\mathtt{Reverse}(\mathtt{add}(\mathtt{0}, \mathtt{empty})))$, $\neg GTZ(\mathtt{atomic}(\mathtt{s}(\mathtt{0})))$,
$\neg GTZ(\mathtt{null})$, and so on. If s does have variables then $GTZ(s)$ says that no matter what objects you substitute for the variables, you will always get an object with size greater than zero. For instance,

$$GTZ(\mathtt{s}(n))$$

$$\neg GTZ(\mathtt{Minus}(n, m))$$

$$\neg GTZ(\mathtt{Plus}(n, m))$$

$$GTZ(\texttt{Plus}(n, \texttt{Fact}(m))).$$

Note that if the top function symbol of s is a constructor then deciding $GTZ(s)$ is trivial and can be done in $O(1)$ time — simply check whether $Top(s)$ is reflexive. For, we always have $GTZ(rcons(\cdots))$ and $\neg GTZ(ircons(\cdots))$ for every reflexive (irreflexive) constructor $rcons$ ($ircons$). However, in general $GTZ(s)$ is mechanically undecidable, i.e. there is no algorithm that will always yield the correct answer. Intuitively, that is because in order to decide $GTZ(s)$ in general we need *semantic* information about the functions computed by the various algorithms whose names appear in s. For example, how can we tell whether or not $GZT(\texttt{f(Times(n,s}^2\texttt{(0))))}$, for some $f : \texttt{NatNum} \longrightarrow \texttt{NatNum}$, without knowing whether or not $f(n) > 0$ for all even numbers n? Of course there are sound algorithms that serve as conservative approximations to a decision procedure (indeed, we will subsequently offer such an algorithm), but these are all bound to be incomplete. Nevertheless, it might be of interest to note that GTZ *is* a decidable predicate if we restrict its domain to $\mathcal{T}(\Sigma_M)$, i.e. to ground terms only. For, by the admissibility of M, all procedures in $PROC_M$ terminate; therefore, to decide $GTZ(s)$ for $s \in \mathcal{T}(\Sigma_M)$, simply compute $Eval_M(s)$ and check to see whether the top function symbol of the result is a reflexive constructor.

3.4 Argument-bounded procedures

A key role in our investigation will be played by the concept of *argument-bounded procedures*. A procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ is said to be i-**bounded** for some $i \in \{1, \ldots, k\}$ if

$$T_{par_i} \equiv T \text{ and } f(v_1, \ldots, v_k) \preceq v_i \tag{3.7}$$

where $v_i \in V_{T_{par_i}}$. Likewise, f is **strictly** i-bounded if (3.7) holds with $\prec$ in place of $\preceq$. We call f argument-bounded if it is i-bounded (weakly or strictly) for at least one of its k argument positions. For example, **Minus** is an 1-bounded procedure, since $\texttt{Minus}(n_1, n_2) \preceq n_1$ for all natural numbers n_1 and n_2 (or, less formally, since $n_1 - n_2 \leq n_1$). Intuitively, that is because the value $\texttt{Minus}(n_1, n_2)$ is always obtained by "removing" something from the first argument (by "reducing" n_1). Note that `Minus` is not 2-bounded, as we do not in general have $n_1 - n_2 \leq n_2$ (e.g. $8-3=5>3$).

3.5 The main termination theorem

In this paper we will only be concerned with termination of **normal** algorithms. In IFL, a procedure f is called normal if every conditional statement `If` $\mathtt{E_j}$ `Then` $\mathtt{r_j}$ in the body of f is such that

(i) the expression $\mathtt{E_j}$ contains no recursive calls $f(\cdots)$, and

(ii) the term $\mathtt{r_j}$ contains no *nested* recursive calls $f(\cdots f(\cdots)\cdots)$.

This is not a severe restriction since most algorithms encountered in practice meet these requirements. Non-normal procedures such as McCarthy's 91–function([5]) or Takeuchi's function for reversing a list ([5]), although mathematically interesting, tend to be rather pathological cases from a practical standpoint. For a treatment of such procedures consult [6].

Following an idea introduced by Floyd ([2]), most methods for proving termination, including ours, are based on the concept of **well–founded sets**. Formally, a well-founded set can be represented as an ordered pair (W, R), where W is a set (any set) and R is a *well-founded binary relation on* W. Recall that R is well-founded iff every non-empty subset of W contains at least one "R–minimal" element; or, equivalently, iff there are no *infinite* chains $x_1, x_2, x_3, \ldots$ such that $x_1 R x_2, x_2 R x_3, x_3 R x_4, \ldots$. R usually induces a partial order on W and for our purposes it is expedient to read xRy as "x is larger than y", in some abstract sense of "large". Under this interpretation, R being well-founded amounts to the preclusion of infinite sequences $x_1, x_2, \ldots$ in which each x_{i+1} is "smaller" than x_i. Thus, simplistically put, no element of W can decrease indefinitely; the descent must eventually stop after finitely many steps. The classic example of a well-founded set is $(N, >)$, the natural numbers under the "greater than" relation. Our method is based on the following fundamental result:

Theorem 3.5.1 (Main termination theorem) *Let M be an admissible module and let M' be an extension of M obtained by defining a new normal procedure f :*

$T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ with k parameters $par_1, \ldots, par_k$ and body

$$[< \mathtt{E_1}, \mathtt{r_1} >, \ldots, < \mathtt{E_m}, \mathtt{r_m} >, < \mathtt{True}, \mathtt{r_{m+1}} >].$$

Then f is terminating in the new module M' iff there is a well-founded set (W, R) and a function $Q : \mathcal{T}(\mathrm{CON}_M^{T_{par_1}}) \times \cdots \times \mathcal{T}(CON_M^{T_{par_k}}) \longrightarrow W$ such that for each recursive call $f(t_1, \ldots, t_k)$ in each $\mathtt{r_j}$, $\mathtt{j} = 1, \ldots, \mathtt{m} + 1$, the following holds for any k objects $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$:

If $TEval_M(\mathtt{E_j}[par_1 \mapsto w_1, \ldots, par_k \mapsto w_k]) = \mathbf{True}$ then

$$Q(w_1, \ldots, w_k) \; R \; Q(Eval_M(t_1[\overrightarrow{par} \mapsto \overrightarrow{w}]), \ldots, Eval_M(t_k[\overrightarrow{par} \mapsto \overrightarrow{w}])).$$

Note that the assumption that f is normal is critical, otherwise the above condition would be nonsensical. For, if some t_i contained an additional recursive call $f(t'_1, \ldots, t'_k)$ then the expression $Eval_M(t_i[\overrightarrow{par} \mapsto \overrightarrow{w}])$ would be undefined because Σ_M does not contain f and thus we cannot use the interpreter $Eval_M$ to compute $t_i[\overrightarrow{par} \mapsto \overrightarrow{w}]$ (since that computation would eventually generate the invalid call $Eval_M(f(t'_1, \ldots, t'_k))$). Moreover, if f were not normal we would not even be able to obtain the truth value $TEval_M(\mathtt{E_j}[\overrightarrow{par} \mapsto \overrightarrow{w}])$, since that involves calling $Eval_M$ on the terms flanking the equality symbols appearing in $\mathtt{E_j}$; if these terms contain recursive calls $f(\cdots)$, we cannot use $Eval_M$ for their evaluation.

We now proceed with the proof of theorem 3.5.1, which will provide us with the opportunity to introduce some ideas that might help to sharpen the reader's intuitions regarding termination. First, for any normal procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ that has k parameters $par_1, \ldots, par_k$ and is well-formed and well-typed wrt to an admissible module M, and for any k objects $w_1 \in Tpar_1, \ldots, w_k \in T_{par_k}$, we define the **recursion tree of the term** $s \equiv f(w_1, \ldots, w_k)$, notated RT_s, as follows:

- The root of RT_s is s.

- The children of a node $f(s_1, \ldots, s_k)$ are specified as follows. Let `If` $\mathtt{E_{j+1}}$ `Then`

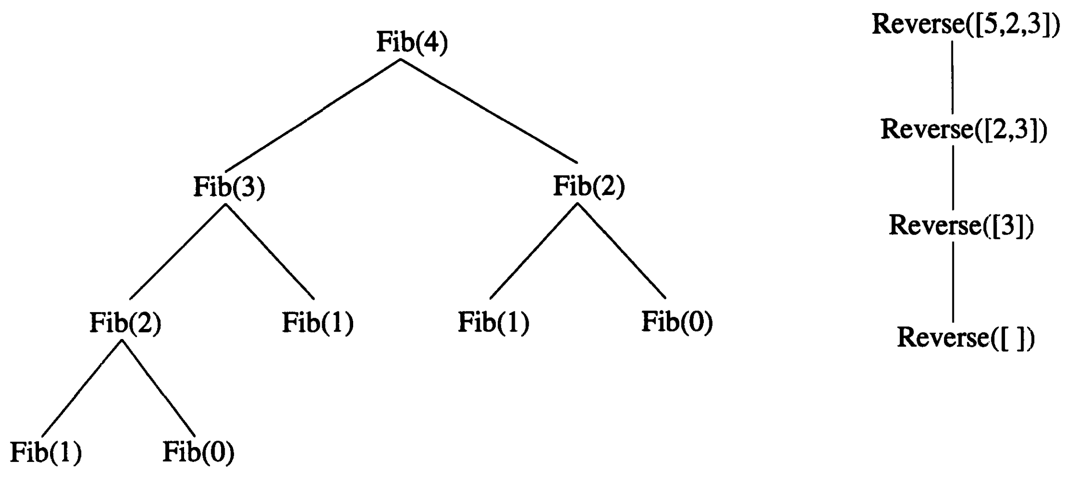

Figure 3-1: The recursion trees of the terms `Fib(4)` and `Reverse([5,2,3])`.

$\mathtt{r}_{\mathtt{j+1}}$ be the unique statement[1] in the body of f such that

$$TEval_M(\mathtt{E}_{\mathtt{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{s}]) = \mathbf{True}$$

and let $f(t_1^1, \ldots, t_k^1)$, ..., $f(t_1^n, \ldots, t_k^n)$ be the $n \geq 0$ "recursive calls" contained in $\mathtt{r}_{\mathtt{j+1}}$, ordered from left to right according to the standard well-ordering of terms. Then the children of $f(s_1, \ldots, s_k)$, from left to right, are the n terms

$$f(Eval_M(t_1^1[\overrightarrow{par} \mapsto \overrightarrow{s}]), \ldots, Eval_M(t_k^1[\overrightarrow{par} \mapsto \overrightarrow{s}]))$$

$$\vdots$$

$$f(Eval_M(t_1^n[\overrightarrow{par} \mapsto \overrightarrow{s}]), \ldots, Eval_M(t_k^n[\overrightarrow{par} \mapsto \overrightarrow{s}])).$$

Thus, if $\mathtt{r}_{\mathtt{j+1}}$ does not contain any recursive calls (i.e. if $n = 0$), then the node $f(s_1, \ldots, s_k)$ is a leaf of RT_s. Also note that if no $\mathtt{r}_{\mathtt{j}}$ in the body of f contains more than one recursive call, then the recursion tree of any term $f(w)$ is essentially a linked list of terms $f(w), f(w'), f(w''), \ldots$. By way of illustration, fig. 3.5 depicts the recursion trees of the terms `Fib(4)` and `Reverse([5,2,3])`.

[1]That there is such a statement follows from the case-completeness and determinism of f in conjunction with the assumptions that M is admissible and f is normal.

Our next goal is to derive yet another intuitive characterization of termination by proving that when we augment an admissible module M with a new normal procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$, the new procedure terminates in the new module M' iff for all objects $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$, the recursion tree of $f(w_1, \ldots, w_k)$ is finite. After we prove this result theorem 3.5.1 will follow naturally, since it is not difficult to see that the conditions of the said theorem preclude infinite recursion trees for terms of the form $f(w_1, \ldots, w_k)$. We begin with the "if" part of the result:

Proposition 3.5.1 *Let M' be the module obtained by extending an admissible module M with a normal procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$, and let $w_1, \ldots, w_k$ be any k objects of type $T_{par_1}, \ldots, T_{par_k}$, respectively. If $f(w_1, \ldots, w_k)$ has a finite recursion tree, then $Eval_{M'}(f(w_1, \ldots, w_k)) \downarrow$.*

In the converse direction, we prove that the call $Eval_M(f(w_1, \ldots, w_k))$ will initiate an in-depth traversal of the recursion tree of $f(w_1, \ldots, w_k)$. It will follow that if the said tree is infinite, the evaluation of $f(w_1, \ldots, w_k)$ will diverge. Proofs for both of these propositions can be found in Appendix A.

Proposition 3.5.2 *Let M' be the module obtained by augmenting an admissible module M with a normal procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$, and let $w_1, \ldots, w_k$ be any k objects of type $T_{par_1}, \ldots, T_{par_k}$ respectively. Then for every finite ordinal $\beta < ORD(RT_{f(w_1,\ldots,w_k)})$ there are terms $u_1, \ldots, u_k$ in $\mathcal{T}(\Sigma_M)$ such that*

$$Eval_{M'}(f(w_1, \ldots, w_k)) \Longrightarrow^* Eval_{M'}(f(u_1, \ldots, u_k))$$

where $Eval_{M'}(u_i) = (\pi_{RT_{f(w_1,\ldots,w_k)}}(\beta)) \downarrow i$ for every $i = 1, \ldots, k$.

As was remarked earlier, it is a consequence of the above proposition that if $RT_{f(w_1,\ldots,w_k)}$ is infinite, the evaluation of $f(w_1, \ldots, w_k)$ will diverge. For, according to what we just proved, we will have

$$Eval_{M'}(f(w_1, \ldots, w_k)) \Longrightarrow^* Eval_{M'}(f(u_1, \ldots, u_k)) \qquad (3.8)$$

for some terms $u_1, \ldots, u_k$ such that $Eval_{M'}(u_i) = \pi_{RT_{f(w_1,\ldots,w_k)}}(\beta) \downarrow i$ for *every* finite ordinal $\beta < ORD(RT_{f(w_1,\ldots,w_k)})$. But if $RT_{f(w_1,\ldots,w_k)}$ is infinite, $ORD(RT_{f(w_1,\ldots,w_k)}) \geq \omega$, hence there are infinitely many finite ordinals $\beta < ORD(RT_{f(w_1,\ldots,w_k)})$, and thus, according to (3.8), the call $Eval_{M'}(f(w_1, \ldots, w_k))$ will recursively invoke the interpreter $Eval_{M'}$ infinitely many times. In fact proposition 3.5.2 conveys more information than the mere prediction that $Eval_{M'}(f(w_1, \ldots, w_k))$ will loop forever if $f(w_1, \ldots, w_k)$ has an infinite recursion tree. It also tells us exactly where the interpreter will diverge; namely, along the *leftmost* inifinite path in $RT_{f(w_1,\ldots,w_k)}$.

Perhaps the following is a more intuitive argument. If $RT_{f(w_1,\ldots,w_k)}$ is infinite then, by König's lemma ([3]), it must have at least one infinite path. Let π_{ω_1} be the leftmost such infinite path (so that for any two nodes u and v, if v is on π_{ω_1} and u is to the left of v in $RT_{f(w_1,\ldots,w_k)}$ but not itself on π_{ω_1}, then the tree rooted at u must be finite). Then, by employing the same argument that we used in the proof of proposition 3.5.2, we can show that for every node u_1 in π_{ω_1} we have $Eval_{M'}(u_1) \Longrightarrow^+ Eval_{M'}(u_2)$, where u_2 is the successor (child) of u_1 in π_{ω_1}. Thus there is an infinite sequence of terms $f(w_1, \ldots, w_k) \equiv u_1, u_2, u_2, \ldots$ such that $Eval_{M'}(u_i) \Longrightarrow^+ Eval_{M'}(u_{i+1})$ for all $i \in \omega$, and thus $Eval_{M'}(f(w_1, \ldots, w_k))$ diverges. We summarize as follows:

Theorem 3.5.2 *Let M' be the module obtained by augmenting an admissible module M with a new normal procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$. Then f is terminating iff for all objects $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$, the term $f(w_1, \ldots, w_k)$ has a finite recursion tree.*

Proving theorem 3.5.1 is now straight-forward: the existence of the function Q—which is said to be a **termination function** for f—guarantees that the term $f(w_1, \ldots, w_k)$ has a finite recursion tree. For, if $f(s_1, \ldots, s_k)$ is any node in $RT_{f(w_1,\ldots,w_k)}$ and $f(s'_1, \ldots, s'_k)$ is any one of its children, we will have $Q(s_1, \ldots, s_k)\ R\ Q(s'_1, \ldots, s'_k)$. But then, because (W, R) is well-ordered, it follows that there are no infinite paths in $RT_{f(w_1,\ldots,w_k)}$, and hence that the latter is finite. Therefore, by the preceding theorem, $Eval_{M'}(f(w_1, \ldots, w_k)) \downarrow$.

The termination function Q should be seen as mapping each k–tuple of objects $w_1 \in T_{par_1}, \ldots, w_k \in T_{par_k}$ to some "quantity" in W. In our method (W, R) will always be $(N, >)$, so the values of Q will be numeric. Furthermore, Q will be based on the size function SZ. In particular, if f is unary ($k = 1$) then Q will actually be SZ itself, i.e. we will have $Q(w) = SZ_{T_{par_1}}(w)$ for all objects $w \in T_{par_1}$. If f has more than one parameter then the termination function will be

$$Q(w_1, \ldots, w_k) = SZ(w_{i_1}) + \cdots + SZ(w_{i_n}) \tag{3.9}$$

where the set $\{i_1, \ldots, i_n\}$ is the so-called **measured subset** of f (in the terminology of [1]). The measured subset of f, which will be written as MS_f, is the set of all argument positions $i \in \{1, \ldots, k\}$ such that for *every* recursive call $f(t_1, \ldots, t_k)$ anywhere in the body of f, we have $t_i \preceq par_i$. Our method, in a nutshell, will compute MS_f and then it will attempt to prove that in each recursive call $f(t_1, \ldots, t_k)$ in the body of f, we have $t_j \prec par_j$ *for at least one* $j \in MS_f$. If successfull, the termination function defined in (3.9) will satisfy the requisite properties of theorem 3.5.1, allowing us to conclude that f terminates for all appropriate inputs.

In order to compute MS_f and to be able to test whether $t_j \prec par_j$ for some $j \in MS_f$, we need an algorithm for computing the two predicates $\preceq$ and $\prec$. Although these are in general undecidable (see section 3.2), they can be "conservatively approximated" to a degree that proves satisfactory for our purposes. In particular, we will present two logic programs (collections of rules) R_w and R_s defining the two predicates, and we will use classic Prolog–style backwards chaining to determine whether or not the goal $s \preceq t$ ($s \prec t$), for two arbitrary terms s and t, can be derived from the "database" $\mathcal{R} = R_w \cup R_s$, i.e. whether $\mathcal{R} \vdash s \preceq t$ ($\mathcal{R} \vdash s \prec t$). An important result that we will prove is that *in the worst case*, using R_w and R_s we can determine whether or not $s \preceq t$ or $s \prec t$ in time linear and quadratic in the size of s, respectively. In fact in most cases arising in practice we are able to compute $\mathcal{R} \vdash s \preceq t$ ($\mathcal{R} \vdash s \prec t$) in time logarithmic in the size of s (i.e. linear in the *height* of s). All of the rules in R_w and R_s will be sound, so if we manage to conclude that, say, $\mathcal{R} \vdash s \preceq t$, then we can validly infer that $s \preceq t$. However, owing to the aforementioned undecidability problem, $\mathcal{R}$ is necessarily incomplete. Therefore, results of the form $\mathcal{R} \vdash s \not\preceq t$ or

$\mathcal{R} \vdash \not\prec t$ do not warantee the respective conclusions $s \not\preceq t$, $s \not\prec t$. The practical import of this is that if the system tells the user that the procedure they have just defined does not terminate, the user is not entitled to infer that the said procedure does not in fact compute a total function. It only means that the user must re-write the procedure in a way that will "help" the system prove its termination, while preserving the intended semantics. We will see that it is easy to develop a procedure-writing style (what is called "Walther style" in [6]) that is conducive to proving termination by our method.

Chapter 4

Recognizing termination

4.1 The weak size inequality calculus R_w

Recall that modules are developed in an incremental fashion, whereby at each step the user extends the "current" module by defining a new data type or a new procedure. Our method for testing termination requires us to keep track of which of the procedures defined so far are argument-bounded. At any given point the set $B_i \subset PROC_M$ will comprise those procedures in the current module M that have been proven to be i-bounded. Note that B_1 will include all the reflexive selectors in SEL_M by default, since all such selectors are 1-bounded. The intended modus operandi is as follows. Immediately after the user has defined a new procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$, the system tries to prove that f is terminating. If it fails, the procedure is rejected as a non-halter and the user must either provide an alternative definition of f or move on to something else. If f is proven to terminate, then the next step is to determine whether it is i-bounded for every $i = 1, \ldots, k$ such that $T_{par_i} = T$. If we succeed in proving that f is i-bounded for such an i, we set $B_i \longleftarrow B_i \cup \{f\}$. Exactly how we go about deciding whether f is i-bounded is discussed in section 4.4.

We are now in a position to present R_w, the set of rules that make up the calculus for weak size inequality. It consists of the Horn clauses [R1]–[R6] given below. We use the symbols s and t with or without subscripts as variables ranging over $\mathcal{T}(\Sigma_M, V_M)$:

$$\boxed{\mathit{True} \implies t \preceq t \quad \text{[R1]}}$$

$$\boxed{\begin{array}{c} \mathit{True} \implies \mathit{ircons}(s_1, \ldots, s_n) \preceq t \quad \text{[R2]} \\ \text{for every irreflexive constructor } \mathit{ircons} \in \mathit{CON}_M. \end{array}}$$

$$\boxed{\begin{array}{c} s_i \preceq t \implies g(s_1, \ldots, s_n) \preceq t \quad \text{[R3]} \\ \text{for every } g \in B_i. \end{array}}$$

$$\boxed{\begin{array}{c} \mathit{GTZ}(t), s_{i_1} \preceq \mathit{sel}_{i_1}^{\mathit{rcons}}(t), \ldots, s_{i_m} \preceq \mathit{sel}_{i_m}^{\mathit{rcons}}(t) \implies \mathit{rcons}(s_1, \ldots, s_n) \preceq t \quad \text{[R4]} \\ \text{for every reflexive constructor } \mathit{rcons} \in \mathit{CON}_M \text{ with} \\ \text{reflexive argument positions } i_1, \ldots, i_m. \end{array}}$$

The GTZ predicate is conservatively approximated by the following rules:

$$\boxed{\begin{array}{c} t = \mathit{rcons}(t_1, \ldots, t_n) \implies \mathit{GTZ}(t) \quad \text{[R5]} \\ \text{for every reflexive constructor } \mathit{rcons} \in \mathit{CON}_M. \end{array}}$$

$$\boxed{\begin{array}{c} t \neq \mathit{ircons}_1(t_1^1, \ldots, t_{n_1}^1), \ldots, t \neq \mathit{ircons}_m(t_1^m, \ldots, t_{n_m}^m) \implies \mathit{GTZ}(t) \quad \text{[R6]} \\ \text{where } \mathit{ircons}_1, \ldots, \mathit{ircons}_m \text{ are all and only the irreflexive} \\ \text{constructors of type } T\text{, for every } T \in \mathit{TYPES}_M. \end{array}}$$

Strictly speaking, each of the above rules is a *schema* that can have many different instances depending on the particular contents of M. To take a simple example, if M comprises the types `NatNum` and `SExpr` and the procedure `Minus`, then, supposing that `Minus` has been proven to be 1-bounded, $\mathcal{R}$ will contain the following instances of [R3]:

$$s \preceq t \implies \texttt{pred}(s) \preceq t$$

$$s \preceq t \implies \texttt{car}(s) \preceq t$$

$$s \preceq t \implies \texttt{cdr}(s) \preceq t$$

$$s_1 \preceq t \implies \texttt{Minus}(s_1, s_2) \preceq t$$

and the following instances of [R6]:

$$t \neq 0 \implies GTZ(t)$$

$$t \neq \texttt{nil}, t \neq \texttt{atom}(t_1) \implies GTZ(t).$$

Soundness

Proving the soundness of the above rules is straightforward, with the possible exception of [R4], for which we will now give a formal proof. Suppose that the antecedent of [R4] is true for two terms $rcons(s_1, \ldots, s_n)$ and t in $\mathcal{T}(\Sigma_M, V_M)$. Let $v_1 \in V_{T_1}, \ldots, v_p \in V_{T_p}$ be the variables occuring in these terms, and let $w_1, \ldots, w_p$ be any p objects of type $T_1, \ldots, T_p$, respectively. To prove that $rcons(s_1, \ldots, s_n) \preceq t$ we must show that $Q_1 \leq Q_2$, where Q_1 is the quantity $SZ(Eval_M(rcons(s_1, \ldots, s_n)[\overrightarrow{v} \mapsto \overrightarrow{w}]))$ and Q_2 is $SZ(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}]))$. With regard to the first quantity we have, by definition of the SZ function,

$$\begin{aligned} Q_1 &= SZ(Eval_M(rcons(s_1, \ldots, s_n)[\overrightarrow{v} \mapsto \overrightarrow{w}])) \\ &= SZ(rcons(Eval_M(s_1[\overrightarrow{v} \mapsto \overrightarrow{w}]), \ldots, Eval_M(s_n[\overrightarrow{v} \mapsto \overrightarrow{w}]))) \\ &= 1 + \sum_{j=1}^{m} SZ(Eval_M(s_{i_j}[\overrightarrow{v} \mapsto \overrightarrow{w}])). \end{aligned} \tag{4.1}$$

As regards Q_2, there are two cases: either $Top(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}])) \equiv rcons$ or not. In the former case we have $Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}]) = rcons(t'_1, \ldots, t'_n)$ for some objects $t'_1, \ldots, t'_n$, and thus

$$Q_2 = 1 + \sum_{j=1}^{m} SZ(t'_{i_j}). \tag{4.2}$$

Now since we have assumed the antecedent of [R4] to be true, we have, for all $j \in \{1, \ldots, m\}$,

$$SZ(Eval_M(s_{i_j}[\overrightarrow{v} \mapsto \overrightarrow{w}])) \ \leq \ SZ(Eval_M(sel_{i_j}^{rcons}(t[\overrightarrow{v} \mapsto \overrightarrow{w}])))$$

$$\begin{aligned} &= SZ(sel^{rcons}_{i_j}(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}]))) \\ &= SZ(sel^{rcons}_{i_j}(rcons(t'_1, \ldots, t'_n))) \\ &= SZ(t'_{i_j}) \end{aligned}$$

and thus it follows from (4.1) and (4.2) that $Q_1 \leq Q_2$.

On the other hand, if $Top(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}])) \not\equiv rcons$ then

$$SZ(sel^{rcons}_{i_j}(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}]))) = 0$$

for every $j = 1, \ldots, m$, since for every such j the term $sel^{rcons}_{i_j}(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}]))$ will be a minimal object built from irreflexive constructors. Now we know from the antecedent that for all $j \in \{1, \ldots, m\}$

$$\begin{aligned} SZ(Eval_M(s_{i_j}(\overrightarrow{v} \mapsto \overrightarrow{w}])) &\leq SZ(Eval_M(sel^{rcons}_{i_j}(t[\overrightarrow{v} \mapsto \overrightarrow{w}]))) \\ &= SZ(sel^{rcons}_{i_j}(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}]))) \\ &= 0 \end{aligned}$$

hence for every such j we must have $SZ(Eval_M(s_{i_j}[\overrightarrow{v} \mapsto \overrightarrow{w}])) = 0$. Therefore, by (4.1), we conclude that $Q_1 = 1$. But now note that, because we have assumed $GTZ(t)$, we have $Q_2 = SZ(Eval_M(t[\overrightarrow{v} \mapsto \overrightarrow{w}])) > 0$, hence $Q_1 \leq Q_2$ and we are done.

Some examples

Here are a few simple derivations in R_w:

$\mathcal{R}_w \vdash \texttt{pred}(\texttt{pred}(n)) \preceq n$:

1.	$n \preceq n$	[R1]
2.	$\texttt{pred}(n) \preceq n$	[R3], since $\texttt{pred} \in B_1$
3.	$\texttt{pred}(\texttt{pred}(n)) \preceq n$	[R3] again.

$\mathcal{R}_w \vdash \texttt{add}(n, \texttt{tail}(l)) \preceq l$, on the assumption that $(l \neq \texttt{empty})$:

1.	$l \preceq l$	[R1]
2.	$\texttt{tail}(l) \preceq l$	[R3]
3.	$GTZ(l)$	[R6], by the assumption $(l \neq \texttt{empty})$
4.	$\texttt{add}(n, \texttt{tail}(l)) \preceq l$	[R4], 2,3.

$\mathcal{R}_w \vdash \texttt{s(Minus(Log(pred}(n)), m)) \preceq n$, assuming that $(n \neq \texttt{0})$ and $\texttt{Minus,Log}^1 \in B_1$:

1.	$\texttt{pred}(n) \preceq \texttt{pred}(n)$	[R1]
2.	$\texttt{Log(pred}(n)) \preceq \texttt{pred}(n)$	[R3], 1
3.	$\texttt{Minus(Log(pred}(n)), m) \preceq \texttt{pred}(n)$	[R3], 2
4.	$GTZ(n)$	[R6], given that $(n \neq \texttt{0})$
5.	$\texttt{s(Minus(Log(pred}(n)), \texttt{m})) \preceq n$	[R4], 3,4.

Deducing atomic expressions from If-Then antecedents

Given that the equality and inequality symbols $=$ and $\neq$ do not appear in the consequent of any of the rules in $\mathcal{R}_w$, one might wonder how will conclusions of the form $(t_1 = t_2)$ or $(t_1 \neq t_2)$ be established. The answer has to do with how termination proofs, as well as proofs of argument-boundedness, are carried out in practice. Such proofs require us to loop through the body $[< \texttt{E}_1, \texttt{r}_1 >, \ldots, < \texttt{E}_\texttt{m}, \texttt{r}_\texttt{m} >]$ of a procedure, examining each statement `If` $\texttt{E}_\texttt{i}$ `Then` $\texttt{r}_\texttt{i}$ in turn, for $\texttt{i} = 1, \ldots, m$. Conclusions of the form $(t_1 = t_2)$ and $(t_1 \neq t_2)$, i.e. atomic expressions, will be derived by analyzing the Boolean expression $\texttt{E}_\texttt{i}$ and will be temporarily added to the global database $\mathcal{R}$ as Prolog facts, so that they can be used by rules [R5] and [R6] to establish GTZ conclusions. For instance, if we were working on the first statement of `Minus` (section 2.2), we would add to $\mathcal{R}$ the assertion $(\texttt{m} = \texttt{0})$.

Such assertions are added to $\mathcal{R}$ only temporarily because they are obviously valid only as long as we are considering that particular statement `If` $\texttt{E}_\texttt{i}$ `Then` $\texttt{r}_\texttt{i}$ from

[1]See section X for a definition of `Log`.

whose antecedent `Ei` the assertions were deduced. When we move to the next statement `If Ei+1 Then ri+1` then all atomic expressions that were derived on the basis of `Ei` must be retracted from $\mathcal{R}$ and replaced by new ones derived from `Ei+1`. In the `Minus` example, when we proceed to the second `If-Then` statement we need to delete the assertion $(\mathtt{m} = \mathtt{0})$ and replace it by $(\mathtt{m} \neq \mathtt{0})$. This process will be made more clear later when we present some concrete examples of termination proofs.

Inferring atomic expressions from antecedents is accomplished by the algorithm $AtEx$ given below, which takes an expression $E \in \mathcal{B}(\Sigma_M, V_M)$ and returns a set of assertions of the form $(t_1 = t_2)$ and $(t_1 \neq t_2)$, each of which is logically implied by E:

```
Function  $AtEx(E : \mathcal{B}(\Sigma_M, V_M))$ {
If  $(E \equiv (t_1 = t_2)$  Or  $E \equiv (t_1 \neq t_2))$  return  $\{E\}$;
Else
    If  $(E \equiv (E_1 \wedge E_2))$  return  $AtEx(E_1) \cup AtEx(E_2)$;
    Else
        If  $(E \equiv (E_1 \vee E_2))$  return  $AtEx(E_1) \cap AtEx(E_2)$;
        Else
            return  $\emptyset$;  }
```

The computational complexity of the R_w calculus as a Prolog program

We end this section by proving that, using the rules of $\mathcal{R}_w$ as a logic program, $\mathcal{R}_w \vdash s \preceq t$ can be decided in no more than $O(\text{SIZE}(s))$ time:

Theorem 4.1.1 *In the worst case, a Prolog interpreter can determine whether or not $\mathcal{R}_w \vdash s \preceq t$, for any $s, t \in \mathcal{T}(\Sigma_M, V_M)$, using $O(n)$ resolution steps, where n is the size of s.*

Proof. By induction on n, the size of s. Supposing that $s \equiv g(s_1, \ldots, s_m)$ and that $s \not\equiv t$[2], there are three mutually exclusive and exhaustive cases: either

[2]If $s \equiv t$ then $s \preceq t$ would follow immediately from [R1].

1. $g \in B_i$ for at least one i, or

2. g is a reflexive constructor, or

3. neither of the above.

In the first scenario the worst case arises when $g \in B_i$ for *every* $i = 1, \ldots, m$, so that $\mathcal{R}$ contains m instances of [R3] with the head $g(s_1, \ldots, s_m) \preceq t$, *and* $\mathcal{R}_w \not\vdash s_j \preceq t$ for all $j \in \{1, \ldots, m-1\}$. Accordingly, in the worst case the interpreter will first try the rule

$$s_1 \preceq t \Longrightarrow g(s_1, \ldots, s_m) \preceq t$$

but will fail to prove $s_1 \preceq t$, then it will try the rule

$$s_2 \preceq t \Longrightarrow g(s_1, \ldots, s_m) \preceq t$$

and will fail to prove $s_2 \preceq t$, and so forth, up to the subgoal $s_m \preceq t$, for which it may either fail or succeed. Hence the total effort in such a case will be

$$T = T_1 + T_2 + \cdots + T_m$$

where T_i is the maximum possible effort (in terms of resolution steps) expended in order to decide $s_i \preceq t$. But by the inductive hypothesis we have $T_i = O(n_i)$, where n_i is the size of s_i, hence

$$T = O(n_1) + \cdots + O(n_m) = O(n_1 + \cdots + n_m) = O(n).$$

As regards (2), the worst case arises when g is reflexive in *all* of its argument positions $1, \ldots, m$ and $\mathcal{R}_w \vdash s_j \preceq sel_j^g(t)$ for all $j \in \{1, \ldots, m-1\}$, as well as $\mathcal{R}_w \vdash GTZ(t)$. In that case the maximum work will be

$$T = T_{GTZ(t)} + T_1 + T_2 + \cdots + T_m$$

where $T_{GTZ(t)}$ is the maximum possible effort for deciding $GTZ(t)$ and T_j is the maximum possible effort for deciding $s_j \preceq sel_j^g(t)$. Again by the inductive hypothesis we get $T = T_{GTZ(t)} + O(n)$, and although the quantity $T_{GTZ(t)}$ is not directly expressible in terms of n, it is easy to see that it is negligible compared to $T_1 + \cdots T_m$; hence it is sensible to conclude that $T = O(n)^3$. Finally, in the third case, given that we have assumed $s \not\equiv t$, there are only two possible scenarios: either $Top(s)$ is an irreflexive constructor or not. In the former case $s \preceq t$ follows immediately from [R2], while in the latter case $\mathcal{R}_w \not\vdash s \preceq t$ also follows immediately since the goal $s \preceq t$ does not unify with the head of any rule in $\mathcal{R}_w$. In either case the search tree of the goal $s \preceq t$ is at most one level deep. □

4.2 The strict inequality calculus R_s

To prove termination we must be able to deduce strict size inequalities of the form $s \prec t$. The calculus $\mathcal{R}_s$ given below is a collection of sound rules that can be used to make such inferences. Perhaps the most important rule in $\mathcal{R}_s$ is [R8]. The rationale behind this rule derives from the simple empirical observation that, sometimes, whether or not an i-bounded procedure f returns an object of *strictly smaller* size than its i^{th} argument depends on whether or not some of its arguments have positive size. For instance, the 1-bounded procedure **Minus** is also strictly 1-bounded whenever both of

[3]Intuitively it is easy to see that deciding $GTZ(t)$ is "easy" by observing that the only predicate symbols appearing in the bodies of [R5] and [R6] are $=$ and $\neq$, which do not appear in the head of any rule in $\mathcal{R}_w$. Therefore, the largest search tree that a goal of the form $GTZ(\cdots)$ can have will only be one level deep: the only resolution step would be that of going from $GTZ(t)$ to $t = rcons(\cdots)$ or $t \neq ircons_1(\cdots), \ldots, t \neq ircons_m(\cdots)$. Of course the tree might be wide (bushy), but it will still be only one level deep. In fact even the width will not be that large at all, as it is linear in the number of constructors in CON_M. More precisely, $GTZ(t)$ can be decided, in the worst case, using $O(\mid CON_M \mid)$ resolution steps, which is inexpensive since CON_M will usually have no more than a few dozen elements.

But we can do even better if we avoid resolving $GTZ(t)$ with an instance of [R5] or [R6] that pertains to a type other than $Type_M(t)$ (e.g. if we avoid matching $GTZ(n)$, for $n \in V_{\texttt{NatNum}}$, with the head of a rule like $t \neq \texttt{empty} \Longrightarrow GTZ(t)$). This can be achieved if we make GTZ a binary predicate by tagging t with its type, giving rise to rules like $t \neq 0 \Longrightarrow GTZ(t, \texttt{NatNum})$. Then deciding $GTZ(t,T)$ would be trivial (even in the worst-case scenario), since it would require no more than $\mid CON_T \mid$ matchings.

its arguments are non-zero. In symbols,

$$GTZ(n,m)^4 \Longrightarrow \texttt{Minus}(n,m) \prec n.$$

Or, as another example, the selector `tail` (which is 1-bounded just in virtue of being reflexive), is also strictly 1-bounded whenever its argument is a non-empty list, i.e.

$$GTZ(l) \Longrightarrow \texttt{tail}(l) \prec l.$$

In fact *all* reflexive selectors strictly reduce their arguments if the latter have positive size. In general, if g is an i-bounded procedure, or selector, of $n > 0$ arguments, and $\{a_1, \ldots, a_m\}$ is a subset of $\{1, \ldots, n\}$, we will call g **strictly i-bounded in positions** $\{a_1, \ldots, a_m\}$ iff

$$GTZ(s_{a_1}, \ldots, s_{a_m}) \Longrightarrow f(s_1, \ldots, s_n) \prec s_i.$$

Thus all reflexive selectors are strictly 1-bounded in the first argument position. In section 4.4 we will discuss in detail how we go about proving that an i-bounded procedure is strictly i-bounded in a certain subset of its argument positions. For this section the mere definition of the concept will suffice. It should be said, however, that for every i-bounded procedure f, $\mathcal{R}_s$ will contain at most one instance of [R8]; thus there cannot be two *distinct* subsets $\{a^1_1, \ldots, a^1_{m_1}\}$ and $\{a^2_1, \ldots, a^2_{m_2}\}$ such that both

$$GTZ(s_{a^1_1}, \ldots, s_{a^1_{m_1}}), s_i \preceq t \;\Longrightarrow\; g(s_1, \ldots, s_n) \prec t$$

and

$$GTZ(s_{a^2_1}, \ldots, s_{a^2_{m_2}}), s_i \preceq t \;\Longrightarrow\; g(s_1, \ldots, s_n) \prec t$$

are in $\mathcal{R}_s$. The rule schemata of $\mathcal{R}_s$ are the following:

$$\boxed{\begin{array}{c} GTZ(t) \;\Longrightarrow\; ircons(s_1, \ldots, s_n) \prec t \quad \text{[R7]} \\ \text{for every irreflexive constructor } ircons \in CON_M. \end{array}}$$

[4]We write $GTZ(t_1, \ldots, t_n)$ as an abbreviation for $GTZ(t_1), \ldots, GTZ(t_n)$.

$$\boxed{\begin{array}{c} GTZ(s_{a_1},\ldots,s_{a_m}),\ s_i \preceq t \implies g(s_1,\ldots,s_n) \prec t \quad \text{[R8]} \\ \text{for every } i\text{-bounded procedure or selector } g \text{ that is} \\ \text{strictly } i\text{-bounded in positions } \{a_1,\ldots,a_m\}. \end{array}}$$

$$\boxed{\begin{array}{c} s_i \prec t \implies g(s_1,\ldots,s_n) \prec t \quad \text{[R9]} \\ \text{for every } i\text{-bounded procedure or reflexive selector } g. \end{array}}$$

$$\boxed{\begin{array}{c} s_{i_1} \prec sel_{i_1}^{rcons}(t),\ldots,s_{i_m} \prec sel_{i_m}^{rcons}(t) \implies rcons(s_1,\ldots,s_n) \prec t \quad \text{[R10]} \\ \text{for every reflexive constructor } rcons \in CON_M \text{ with} \\ \text{reflexive argument positions } i_1,\ldots,i_m. \end{array}}$$

An inductive argument similar to the one used in the case of $\mathcal{R}_w$ can establish the following:

Theorem 4.2.1 *In the worst case, a Prolog interpreter can decide whether or not* $\mathcal{R}_s \vdash s \prec t$, *for any* $s, t \in \mathcal{T}(\Sigma_M, V_M)$, *using* $O(n^2)$ *resolution steps, where* n *is the size of* s.

4.3 Deciding termination

The termination algorithm we are about to present uses an auxiliary function $Holds_f$ that takes a goal $s \preceq t$ (or $s \prec t$) and an expression $\mathtt{E_j}$ from the body

$$[<\mathtt{E_1},\mathtt{r_1}>,\ldots,<\mathtt{E_m},\mathtt{r_m}>,<\mathtt{True},\mathtt{r_{m+1}}>]$$

of a procedure f, computes

$$AtEx(\mathtt{E_j}) \bigcup_{\mathtt{i<j}} AtEx(\neg\mathtt{E_i}), \tag{4.3}$$

adds the results to $\mathcal{R}$, and then, in the fashion of a Prolog interpreter, tries to establish the goal $s \preceq t$ ($s \prec t$). When a verdict is reached the atomic expressions of (4.3) that were earlier added to $\mathcal{R}$ are now deleted from it and an answer of either **True** or **False** is returned depending on whether or not the given goal was proved.

Now suppose that an admissible module M is augmented with a new procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ with body

$$[< \mathtt{E_1}, \mathtt{r_1} >, \ldots, < \mathtt{E_m}, \mathtt{r_m} >, < \mathtt{True}, \mathtt{r_{m+1}}]$$

and that our task is to determine whether f is terminating in the new module. The first step is to compute MS_f, the measured subset of f. This can be easily done by looping through the integers $i = 1, \ldots, k$ and checking, for each such i, that for every recursive call $f(t_1, \ldots, t_k)$ contained in a term $\mathtt{r_j}$, $\mathtt{j} \in \{1, \ldots, m+1\}$, we have $Holds_f(t_i \preceq par_i, \mathtt{E_j})$. We abstract this process into a separate function *ComputeMS* as follows:

```
**Function** $ComputeMS(f)$ {
$res \longleftarrow \emptyset$;
**For** $(i = 1;\ i \leq k;\ ++i)$ {
    **For** $(j = 1;\ j \leq m+1;\ ++j)$
        **For** every subterm of $\mathtt{r_j}$ of the form $f(t_1, \ldots, t_k)$
            **If** $\neg Holds_f(t_i \preceq par_i, \mathtt{E_j})$ **Then** {
                $j \longleftarrow m+2; break;$ }
    **If** $(j == m+1)$ **Then**
        $res \longleftarrow res \cup \{i\};$ }
**return** $res$; }
```

Now to verify termination we simply check that for each recursive call $f(t_1, \ldots, t_k)$ in the body of f there is at least one $i \in MS_f$ such that $t_i \prec par_i$:

```
**Function** $Terminates(f)$ {
$MS_f \longleftarrow ComputeMS(f)$;
**For** $j = 1$ **To** $m+1$ **Do**
    **For** every recursive call $f(t_1, \ldots, t_k)$ in $\mathtt{r_j}$ **Do**
        **If** there is no $i \in MS_f$ such that $Holds_f(t_i \prec par_i, \mathtt{E_j})$
            **return False**;
**return True**; }
```

The soundness of this algorithm follows from the main termination theorem if we just define the termination function Q as

$$Q(w_1, \ldots, w_k) = \sum_{i \in MS_f} SZ(w_i).$$

Then let $f(t_1, \ldots, t_k)$ be any recursive call in any term $\mathtt{r_j}$, let $w_1, \ldots, w_k$ be any k objects of type $T_{par_1}, \ldots, T_{par_k}$, respectively, and suppose that $TEval_M(\mathtt{E_j}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = \mathbf{True}$. By definition of MS_f (and the soundness of *ComputeMS*), we have $t_i \preceq par_i$ for all $i \in MS_f$, hence

$$SZ(Eval_M(par_i[\overrightarrow{par} \mapsto \overrightarrow{w}])) \geq SZ(Eval_M(t_i[\overrightarrow{par} \mapsto \overrightarrow{w}]))$$

or

$$SZ(Eval_M(w_i)) \geq SZ(Eval_M(t_i[\overrightarrow{par} \mapsto \overrightarrow{w}]))$$

or

$$SZ(w_i) \geq SZ(Eval_M(t_i[\overrightarrow{par} \mapsto \overrightarrow{w}]))$$

for every such i. Summing over MS_f we obtain

$$Q_1 = \sum_{i \in MS_f} SZ(w_i) \geq Q_2 = \sum_{i \in MS_f} SZ(Eval_M(t_i[\overrightarrow{par} \mapsto \overrightarrow{w}])). \tag{4.4}$$

But by definition of $Terminates$ there exists an $i_0 \in MS_f$ such that

$$\mathcal{R} \bigcup AtEx(\mathtt{E_j}) \bigcup_{\mathtt{i<j}} AtEx(\neg \mathtt{E_i}) \vdash t_{i_0} \prec par_{i_0}.$$

Because $\mathcal{R}$ and $AtEx$ are sound and because the ground boolean expressions $\mathtt{E_j}[\overrightarrow{par} \mapsto \overrightarrow{w}], \neg \mathtt{E_i}[\overrightarrow{par} \mapsto \overrightarrow{w}], 1 \leq \mathtt{i} \leq m+1, \mathtt{i} \neq \mathtt{j}$, are assumed to be true, we conclude that

$$SZ(w_{i_0}) > SZ(Eval_M(t_{i_0}[\overrightarrow{par} \mapsto \overrightarrow{w}]))$$

thereby deducing from (4.4) that $Q_1 > Q_2$. The argument is thus concluded in virtue of the main termination theorem since Q_1 and Q_2 are, respectively, the values of the termination function Q on $w_1, \ldots, w_k$ and $Eval_M(t_1[\overrightarrow{par} \mapsto \overrightarrow{w}]), \ldots, Eval_M(t_k[\overrightarrow{par} \mapsto \overrightarrow{w}])$.

We end this section by demonstrating our method on a simple example, the procedure `Plus` defined in section 2.2. The call $ComputeMS_f(\texttt{Plus})$ will quickly return the set $\{1, 2\}$ since, in the only recursive call in the body of `Plus`, namely `Plus(n,pred(m))`, we have both $\mathcal{R} \vdash \texttt{n} \preceq \texttt{n}$ and $\mathcal{R} \vdash \texttt{pred(m)} \preceq \texttt{m}$. Then by adding the assertion $\texttt{m} \neq \texttt{0}$ to $\mathcal{R}$ we find that $\mathcal{R} \vdash \texttt{pred(m)} \prec \texttt{m}$ through the following instance of [R8]:

$$GTZ(s),\ s \preceq t \implies \texttt{pred}(s) \prec t$$

and thus $Terminates(\texttt{Plus}) = \textbf{True}$. All other procedures given in section 2.2 have equally simple termination proofs.

4.4 Deciding argument–boundedness

Once a newly defined procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ with body

$$[< \texttt{E}_1, \texttt{r}_1 >, \ldots, < \texttt{E}_\texttt{m}, \texttt{r}_\texttt{m} >, < \texttt{True}, \texttt{r}_{\texttt{m}+1}]$$

has been proven to terminate, the next order of business is to attempt to prove that f is i–bounded for every $i \in \{1, \ldots, k\}$ such that $T_i \equiv T$. If we succeed in proving that f is i-bounded for such an i, we add to $\mathcal{R}$ the following instances of [R3] and [R9]:

$$s_i \preceq t \implies f(s_1, \ldots, s_k) \preceq t$$

$$s_i \prec t \implies f(s_1, \ldots, s_k) \prec t.$$

To prove that f is i–bounded we try to prove that $\texttt{r}_\texttt{j} \preceq par_i$ for every $\texttt{j} = \texttt{1}, \ldots, \texttt{m} + \texttt{1}$; or, more precisely, that $Holds_f(\texttt{r}_\texttt{j} \preceq par_i, \texttt{E}_\texttt{j})$ for every such $\texttt{j}$. This is clearly a sound method—if it succeeds, we can validly infer that f is i-bounded. For, let $w_1, \ldots, w_k$ be aby objects of type $T_{par_1}, \ldots, T_{par_k}$, respectively, and let `If E`$_\texttt{j}$ `Then r`$_\texttt{j}$ be the unique statement in the body of f such that $TEval_M(\texttt{E}_\texttt{j}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = \textbf{True}$. Presumably we have already established that $Holds_f(\mathtt{r_j} \preceq par_i, \mathtt{E_j})$, i.e. that

$$\mathcal{R} \bigcup AtEx(\mathtt{E_j}) \bigcup_{\mathtt{i<j}} AtEx(\neg\mathtt{E_i}) \vdash \mathtt{r_j} \preceq par_i,$$

therefore, because $\mathcal{R}$ is sound and because we are assuming the expressions in $AtEx(\mathtt{E_j}[\overrightarrow{par} \mapsto \overrightarrow{w}])$ and in

$$\bigcup_{\mathtt{i<j}} AtEx(\neg\mathtt{E_i}[\overrightarrow{par} \mapsto \overrightarrow{w}])$$

to be true, we can conclude that

$$SZ(Eval_M(\mathtt{r_j}[\overrightarrow{par} \mapsto \overrightarrow{w}])) \leq SZ(w_i)$$

or, equivalently, that

$$SZ(Eval_M(f(w_1, \ldots, w_k))) \leq SZ(w_i).$$

Since this argument is valid for *any* $w_1, \ldots, w_k$, we are entitled to conclude that f is i-bounded.

The method is based on an inductive argument on the number of recursive invocations of $Eval_M(f(\cdots))$ spawned by a call $Eval_M(f(s_1, \ldots, s_k))$ (we know that there is such a finite number because f has been proven to terminate). We illustrate the method on procedure `Minus`, whose text we reproduce here for convenience:

```
Procedure Minus(n,m:NatNum):NatNum
Begin
If (m = 0) Then n
Else
    Minus(pred(n),pred(m));
End.
```

Letting `n` and `m` be any two natural numbers, we begin with the first statement, `If (m = 0) Then n`, and we easily prove that $\mathtt{r_1} \equiv \mathtt{n} \preceq \mathtt{n}$. The next statement contains the recursive call `Minus(pred(n),pred(m))`, so it is here that we utilize the aforementioned inductive hypothesis (since the evaluation of `Minus(pred(n),pred(m))` will spawn fewer—one less, to be precise—recursive invocations of `Minus` than the evaluation of `Minus(n,m)`); thus we postulate the assertion

$$\mathtt{Minus(pred(n),pred(m))} \preceq \mathtt{pred(n)}.$$

To incorporate transitivity, we cast the above assertion as follows:

$$\mathtt{pred}(s_1) \preceq t \implies \mathtt{Minus}(\mathtt{pred}(s_1),\mathtt{pred}(s_2)) \preceq t. \tag{4.5}$$

Next, we add to $\mathcal{R}$ the expression $(\mathtt{m} \neq \mathtt{0})$ and we try to prove the goal

$$\mathtt{Minus(pred(n),pred(m))} \preceq \mathtt{n}.$$

This matches the head of the inductive hypothesis with the bindings $\{t \mapsto \mathtt{n}, s_1 \mapsto \mathtt{n}, s_2 \mapsto \mathtt{m}\}$, so we are left to prove the subgoal $\mathtt{pred(n)} \preceq \mathtt{n}$, which is easily done through [R3]. Since there are no more statements to consider the proof is complete, and we can now record the knowledge that `Minus` is 1-bounded by adding to $\mathcal{R}$ the two following lemmas:

$$s_1 \preceq t \implies \mathtt{Minus}(s_1,s_2) \preceq t$$

and

$$s_1 \prec t \implies \mathtt{Minus}(s_1,s_2) \prec t.$$

The following algorithm is a generalization of the the inductive process illustrated above for an arbitrary procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ with body

$$[<\mathtt{E_1},\mathtt{r_1}>,\ldots,<\mathtt{E_m},\mathtt{r_m}>,<\mathtt{True},\mathtt{r_{m+1}}>]:$$

```
**Function** $IsBounded(f,i)$ {
**For** $j = 1$ **To** $m+1$ {
    **For** every recursive call $f(s_1,\ldots,s_k)$ in $\mathtt{r_j}$ add to $\mathcal{R}$ the rule
```

```
            $s_i \preceq t \Longrightarrow f(s_1, \ldots, s_k) \preceq t$    // the ind. hyp.
    $res \longleftarrow Holds_f(\mathtt{r_j} \preceq par_i, \mathtt{E_j})$;
    Delete from $\mathcal{R}$ all inductive hypotheses previously added to it.
    **If** $\neg res$ **Then**
        **return False**; }
**return True**; }
```

4.5 Deriving strict inequality lemmas for argument-bounded procedures

After a procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ has been proven to be i-bounded for some $i \in \{1, \ldots, k\}$, the next step is to try to obtain a strict inequality lemma for f and i in the form of rule [R8]. We do this by iterating through the power-set of $\{1, \ldots, k\}$ and trying, for each subset $\{a_1, \ldots, a_n\} \subseteq \{1, \ldots, k\}$, to prove the assertion

$$GTZ(par_{a_1}, \ldots, par_{a_n}) \Longrightarrow f(par_1, \ldots, par_k) \prec par_i$$

or, more generally,

$$GTZ(par_{a_1}, \ldots, par_{a_n}),\ par_i \preceq t \Longrightarrow f(par_1, \ldots, par_k) \prec t.$$

This is done by an inductive argument on the "running time" of $f(par_1, \ldots, par_k)$[5] similar to that used in verifying that f is i-bounded. In particular, we loop through the body of f identifying those statements `If` $\mathtt{E_j}$ **Then** $\mathtt{r_j}$ for which the assertion $GTZ(par_{a_1}, \ldots, par_{a_n})$ is consistent with $\mathtt{E_j}$, and then we try to prove, for each such statement, that $Holds_f(\mathtt{r_j} \preceq par_i, \mathtt{E_j})$—after adding to $\mathcal{R}$ the assertions $GTZ(t_1)$, $\ldots$, $GTZ(t_n)$ as well as the inductive hypothesis

$$GTZ(s_{a_1}, \ldots, s_{a_n}),\ s_i \preceq t \Longrightarrow f(s_1, \ldots, s_k) \prec t$$

[5]Recall that f has been proven to terminate.

for every recursive call $f(s_1, \ldots, s_k)$ occuring in $\mathtt{r_j}$.

To determine whether an antecedent $\mathtt{E_j}$ is "consistent" with the assertion

$$GTZ(par_{a_1}, \ldots, par_{a_n})$$

we use the algorithm *Consistent* given below, which takes an expression $E \in \mathcal{B}(\Sigma_M, V_M)$ and a set of terms $\{t_1, \ldots, t_n\}$ and returns **False** if $E \Longrightarrow \neg GTZ(t_1, \ldots, t_n)$. Thus, if $Consistent(E, \{t_1, \ldots, t_n\}) =$ **False**, then E logically implies $\neg GTZ(t_i)$ for at least one $i \in \{1, \ldots, m\}$:

```
Function  $Consistent(E, \{t_1, \ldots, t_n\})$ {
If  $E \equiv (t = ircons(\cdots))$  return  $(t \neq t_1 \wedge \cdots \wedge t \neq t_n)$;
Else
    If  $E \equiv (E_1 \wedge E_2)$  Then return  $Consistent(E_1, \{t_1, \ldots, t_n\}) \wedge$
                                  $Consistent(E_2, \{t_1, \ldots, t_n\})$;
    Else
        If  $E \equiv (E_1 \vee E_2)$  Then return  $Consistent(E_1, \{t_1, \ldots, t_n\}) \vee$
                                      $Consistent(E_2, \{t_1, \ldots, t_n\})$;
        Else
            return True; }
```

Note that, in general, this is an undecidable problem. The above algorithm is a simple conservative approximation to the ideal "black box", which would return **False** *if and only if* E logically implied $\neg GTZ(t_1, \ldots, t_n)$. That is clearly not the case with *Consistent*, since, for example, the expression $E_0 \equiv (\mathtt{Minus(n,0)} = \mathtt{0})$ logically implies $\neg GTZ(\mathtt{n})$ and yet

$$Consistent(E_0, \{\mathtt{n}\}) = \textbf{True}.$$

However, this will be seen to be acceptable for our purposes.

We now present the method we outlined in the first paragraph in the more precise form of an algorithm:

Input: A procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ with body $[<\mathtt{E_1}, \mathtt{r_1}>, \ldots,$ $<\mathtt{E_m}, \mathtt{r_m}>, <\mathtt{True}, \mathtt{r_{m+1}}>]$ and an integer $i \in \{1, \ldots, k\}$. It is assumed that f has been proven to be i-bounded.

Output: A subset $\{a_1, \ldots, a_n\}$ of $\{1, \ldots, k\}$. If the subset is non-empty, then the following holds and should be added to $\mathcal{R}$:

$$GTZ(s_{a_1}, \ldots, s_{a_n}),\ s_i \preceq t \implies f(s_1, \ldots, s_k) \prec t.$$

```
For every non-empty subset $\{a_1, \ldots, a_n\}$ of $\{1, \ldots, k\}$ Do {
  For $j = 1$ To $m + 1$ Do
    If $Consistent_f(\mathtt{E_j}, \{par_{a_1}, \ldots, par_{a_n}\})$ Then {
      Add to $\mathcal{R}$ the assertions $GTZ(par_{a_1}), \ldots, GTZ(par_{a_n})$;
      For every recursive call $f(t_1, \ldots, t_k)$ in $\mathtt{r_j}$ add to $\mathcal{R}$ the
      inductive hypothesis
        $GTZ(t_{a_1}, \ldots, t_{a_n}), t_i \preceq t \implies f(t_1, \ldots, t_k) \prec t$;
      $res \longleftarrow Holds(\mathtt{r_j} \prec par_i, \mathtt{E_j})$;
      Delete from $\mathcal{R}$ everything that was added to it in the
      previous three steps;
      If $(\neg res)$ Then
        $j \longleftarrow m + 2$; }
  If $(j == m + 1)$ Then
    return $\{a_1, \ldots, a_n\}$; }
return $\emptyset$ ;
```

where $Consistent_f(\mathtt{E_j}, T)$ is a shorthand for

$$Consistent(\mathtt{E_j}, T) \bigwedge_{\mathtt{i<j}} Consistent(\neg\mathtt{E_i}, T).$$

A few remarks are in order here. First, note that the algorithm stops immediately after discovering the first subset $\{a_1, \ldots, a_n\} \subseteq \{1, \ldots, k\}$ for which it can prove the desired result. Therefore, as was pointed out earlier, the database $\mathcal{R}$ will contain *at* *most* one instance of [R8] for every procedure $f \in B_i$; otherwise we could conceivably derive up to $2^k - 1$ such lemmas for f, most of which would be superfluous. Secondly, the subsets of $\{1, \ldots, k\}$ should be generated in order of increasing cardinality, starting with the one-element subsets $\{a_1\}$, continuing with the two-element subsets $\{a_1, a_2\}$, and so forth. This will ensure that the subset returned by the algorithm has *minimal cardinality*, i.e. that there is no smaller subset of $\{1, \ldots, k\}$ for which the same lemma could be proved. This is desirable because the smaller the subset is, the fewer the GTZ assertions in the antecedent of the derived lemma, which means that the lemma has the weakest possible antecedent. Thus if and when we come to use the said lemma in a future proof, we will have to prove as little as possible in order to draw the desired conclusion. Finally, it should be said that in any reasonable implementation the above algorithm should execute fast. The fact that the algorithm takes $O(2^k)$ time is inconsequential since in practice k ranges from 1 or 2 in most cases to no more than 5 or 6 in rare cases.

We illustrate the method on the procedure Minus. We assume it has been proved that $\texttt{Minus} \in B_1$, so that $\mathcal{R}$ already contains the two following rules:

$$s_1 \preceq t \implies \texttt{Minus}(s_1, s_2) \preceq t \tag{4.6}$$

$$s_1 \prec t \implies \texttt{Minus}(s_1, s_2) \prec t. \tag{4.7}$$

Now, beginning with the subset $\{\texttt{n}\} \subset \{\texttt{n}, \texttt{m}\}$, we find that

$$Consistent(\texttt{m} = \texttt{0}, \{\texttt{n}\}) = \textbf{True}.$$

We add to $\mathcal{R}$ the assertion $GTZ(\texttt{n})$ but we fail to prove $\texttt{n} \prec \texttt{n}$, so we continue with the next one-element subset of $\{\texttt{n}, \texttt{m}\}$, namely $\{\texttt{m}\}$. We find that $Consistent(\texttt{m} = \texttt{0}, \{\texttt{m}\}) =$ **False**, so we move on to the second statement, determining that

$$Consistent(\texttt{m} \neq \texttt{0}, \{\texttt{m}\}) = \textbf{True}.$$

We add to $\mathcal{R}$ the assertion $GTZ(\mathtt{m})$ and the inductive hypothesis

$$GTZ(\mathtt{pred(m)}),\ \mathtt{pred(n)} \preceq t \implies \mathtt{Minus(pred(n),pred(m))} \prec t \tag{4.8}$$

and we proceed to prove that

$$\mathtt{Minus(pred(n),pred(m))} \prec \mathtt{m}.$$

We first try rule (4.7), but we fail because $\mathcal{R} \not\vdash \mathtt{pred(n)} \prec \mathtt{m}$. Then we try rule (4.8), the inductive hypothesis, and fail again since $\mathcal{R} \not\vdash GTZ(\mathtt{pred(m)})$. Since these two are the only rules in $\mathcal{R}$ whose heads match the goal at hand, we find that $Holds_{\mathtt{Minus}}(\mathtt{Minus(pred(n),pred(m))} \prec \mathtt{m}, \mathtt{m} \neq \mathtt{0}) = \mathbf{False}$, and we continue with the next subset, namely $\{\mathtt{n},\mathtt{m}\}$. The first antecedent ($\mathtt{m} = \mathtt{0}$) is provably inconsistent with $GTZ(\mathtt{n},\mathtt{m})$ so we take up the second statement, whose antecedent is not inconsistent with that assertion. We add to $\mathcal{R}$ the assertions $GTZ(\mathtt{n})$ and $GTZ(\mathtt{m})$ as well as the same inductive hypothesis we had added earlier. We are now left to prove that $\mathtt{Minus(pred(n),pred(m))} \prec \mathtt{n}$, which, through (4.7), backchains to $\mathtt{pred(n)} \prec \mathtt{n}$. This, in turn, is easily established by the rule[6]

$$GTZ(s), s \preceq t \implies \mathtt{pred}(s) \prec t$$

since we have assumed $GTZ(\mathtt{n})$. Hence the algorithm returns the set $\{\mathtt{n},\mathtt{m}\}$ and $\mathcal{R}$ is augmented with the following rule:

$$GTZ(s_1, s_2),\ s_1 \preceq t \implies \mathtt{Minus}(s_1, s_2) \prec t.$$

[6]Presumably $\mathcal{R}$ already contains that rule. In general, rules of the form $GTZ(s), s \preceq t \implies rsel(s) \prec t$ are entered into $\mathcal{R}$ immediately after the user has defined the data type introducing the reflexive selector *rsel*.

4.6 Examples, strengths, and shortcomings

In this section we demonstrate the pros and cons of our method on several well-known algorithms. We begin with algorithms for computing greatest common divisors. One such algorithm can be written in IFL as follows:

```
Procedure Gcd(n,m:NatNum):NatNum;
Begin
If n = 0 Or m = 0 Then Max(n,m)
Else
    If Leq(n,m) = True Then Gcd(n,Minus(m,n))
    Else
       Gcd(Minus(n,m),m);
End.
```

Owing to the fact that Minus $\in B_1$, we easily find the measured subset of Gcd to be $\{1,2\}$: regarding the first argument position we have $\mathcal{R} \vdash \mathtt{n} \preceq \mathtt{n}$ in the first recursive call, and $\mathcal{R} \vdash \mathtt{Minus(n,m)} \preceq \mathtt{n}$ in the second; likewise, for the second position we have $\mathcal{R} \vdash \mathtt{Minus(m,n)} \preceq \mathtt{m}$ in the first recursion, and $\mathcal{R} \vdash \mathtt{m} \preceq \mathtt{m}$ in the second one. Hence $MS_{\mathtt{Gcd}} = \{1,2\}$. Termination follows easily since both recursive calls in the body of Gcd are under the assumption that both $\mathtt{n} \neq \mathtt{0}$ and $\mathtt{m} \neq \mathtt{0}$. Thus in the first recursive call we find that the second argument decreases (i.e. $\mathtt{Minus(m,n)} \prec \mathtt{m}$), while in the second recursion we find that the first argument decreases (i.e. $\mathtt{Minus(n,m)} \prec \mathtt{n}$). Thus Gcd is proven to terminate.

Another, perhaps more intuitive way of computing gcds is the following:

```
Procedure Gcd1(n,m:NatNum):NatNum;
Begin
If n = m Then n
Else
    If n = 0 Then m
    Else
```

```
        If m = 0 Then n
        Else
           Gcd1(Mod(n,m),Mod(m,n));
End.
```

Our method cannot verify the termination of this algorithm, and it is instructive to understand why. Suppose we defined the remainder procedure Mod as follows:

```
Procedure Mod(n,m:NatNum):NatNum;
Begin
If n = 0 Then 0
Else
    If m = 0 Or Less(n,m) = True Then n
    Else
       Mod(Minus(n,m),m);
End.
```

The termination of Mod is easy to verify: first its measured subset is found to be $\{1,2\}$, and then in the only recursive call there is we find that $\texttt{Minus(n,m)} \prec \texttt{n}$ (since in that case we have $\texttt{n} \neq \texttt{0}, \texttt{m} \neq \texttt{0}$). Mod is also easily proved to be 1–bounded: in the first If-Then statement we have $\texttt{0} \preceq \texttt{n}$, in the second we have $\texttt{n} \preceq \texttt{n}$, and in the third, under the inductive hypothesis

$$\texttt{Minus(n,m)} \preceq t \implies \texttt{Mod(Minus(n,m),m)} \preceq t$$

we conclude that $\texttt{Mod(Minus(n,m),m)} \preceq \texttt{n}$. Subsequently the system would try to prove that Mod is 2–bounded, which would obviously fail since Mod is not in fact 2–bounded (e.g. Mod(5,0) = 5).

That Mod is not 2–bounded may sound curious from a mathematical viewpoint, since, by definition, one thinks of the result of the remainder operation as strictly smaller than the divisor (the second argument). However, the customary mathematical definition of the remainder function does not permit the second argument to be zero; whereas, as an IFL procedure, Mod must return some value for all possible combination of inputs, even when the second argument is zero. This is the source of the peculiarity here[7]. In fact the only way `Mod` could be 2–bounded is if it returned zero when the second argument is zero. Even then, however, we would not be able to *prove* that `Mod` is 2–bounded as long as the latter contained the statement `If Less(n,m) = True Then n`. For then we would still have to prove $\mathtt{n} \preceq \mathtt{m}$, which is not derivable in our system from the antecedent `Less(n,m) = True`—even though it logically follows from it. This point highlights the difference between our method and Walter's approach of using a theorem prover: in order to deduce $\mathtt{n} \preceq \mathtt{m}$ from `Less(n,m) = True` one needs to have semantic knowledge about the *meaning* of procedures such as `Less`. Our method, being almost entirely syntactic, does not have such knowledge and consequently fails to infer many size inequalities which may be crucial to termination and/or argument–boundedness. In contrast, by making the derivation of such inequalities contingent on the verification of hypotheses such as `Less(m,n) = True`, which are to be subsequently established by a general-purpose theorem prover, Walther's method will ultimately fare better in some cases because it will take advantage of the power afforded by such a prover.

After failing to prove that `Mod` is 2–bounded, and having proven that it is 1–bounded, our method would try to derive one of the three following rules, in the given order:

$$GTZ(s_1), s_1 \preceq t \Longrightarrow \mathtt{Mod}(s_1, s_2) \prec t \tag{4.9}$$

$$GTZ(s_2), s_1 \preceq t \Longrightarrow \mathtt{Mod}(s_1, s_2) \prec t \tag{4.10}$$

$$GTZ(s_1, s_2), s_1 \preceq t \Longrightarrow \mathtt{Mod}(s_1, s_2) \prec t. \tag{4.11}$$

Since none of these in fact hold and our method is sound, it goes without saying that we would not be able to derive any of them.

Returning to the termination of `Gcd1`, we compute $MS_{\mathtt{Gcd1}} = \{1, 2\}$. But now, to prove termination, we must show either that $\mathtt{Mod(n,m)} \prec \mathtt{n}$ or that $\mathtt{Mod(m,n)} \prec \mathtt{m}$. Neither conclusion follows in our system from the premises $\mathtt{n} \neq \mathtt{0}, \mathtt{m} \neq \mathtt{0}, \mathtt{n} \neq \mathtt{m}$ (in

[7]The same goes for the IFL implementations of other mathematical functions that are undefined for some arguments, e.g. for division.

fact the first conclusion *should* not follow from these premises; consider `Mod(2,5) = 2`). The second conlusion does follow, but our method cannot prove this, as, in the absence of rules such as (4.9)–(4.11), the only rule available whose head matches the goal `Mod(m,n)` $\prec$ `m` is

$$s_1 \prec t \Longrightarrow \texttt{Mod}(s_1, s_2) \prec t$$

(added to $\mathcal{R}$ immediately after `Mod` was discovered to be 1–bounded), which generates the impossible subgoal `m` $\prec$ `m`.

Next we consider algorithms for finding the minimum element in a given collection of numbers. First we would like a procedure that returns the smallest of two given numbers. A natural implementation is the following:

```
Procedure Min(n,m:NatNum):NatNum;
Begin
If Less(n,m) = True Then n
Else
    m;
End.
```

It is noteworthy that although the termination of `Min` is trivally verified by our method (vacuously, since there are no recursive calls) we are unable to prove that the procedure is argument–bounded (of course it is both 1–bounded and 2–bounded). Doing so would require us to prove at some point either that `n` $\preceq$ `m` or that `m` $\preceq$ `n`, neither of which can be achieved in our system since both require additional knowledge about the semantics of `Less`. Assuming that an inductive theorem prover were available, Walther's method would succeed here in proving that `Min` is both 1–bounded and 2–bounded, as such a prover would presumably be able to deduce from the condition $\neg$`Less(n,m)` (in the second statement of `Min`) that `m` $\preceq$ `n`; and from `Less(n,m)` (in the first statement) that `n` $\preceq$ `m`.

Our method cannot do that, but there is another straightforward way to write the "Min" procedure that will allow us to infer argument–boundedness on both positions:

```
Procedure Min1(n,m:NatNum):NatNum;
```

```
Begin
If (n = 0) Or (m = 0) Then 0
Else
    s(Min1(pred(n),pred(m)));
End.
```

The termination of Min1 is easy to verify since $\mathcal{R} \vdash$ pred(n) $\preceq$ n, $\mathcal{R} \vdash$ pred(m) $\preceq$ m, and, on the assumption that n $\neq$ 0 and m $\neq$ 0, we have both pred(n) $\prec$ n and pred(m) $\prec$ m. But now we can also prove that Min1 is 1–bounded. In the first case we trivially have 0 $\preceq$ n (by the appropriate instance of [R2]). For the second case we temporarily add to $\mathcal{R}$ the inductive hypothesis

$$\texttt{pred(n)} \preceq t \Longrightarrow \texttt{Min1(pred(n), pred(m))} \preceq t$$

and the expressions in $AtEx(\neg \mathrm{E}_1)$, namely, n $\neq$ 0 and m $\neq$ 0, and we proceed to establish s(Min1(pred(n),pred(m))) $\preceq$ n. By [R4], this backchains to GTZ(n) and Min1(pred(n),pred(m)) $\preceq$ pred(n). GTZ(n) is easy: it follows directly from n $\neq$ 0 (rule [R6]). For the subgoal

$$\texttt{Min1(pred(n), pred(m))} \preceq \texttt{pred(n)}$$

we resort to the inductive hypothesis, which yields the trivial subgoal pred(n) $\preceq$ pred(n). This proves Min1 to be 1–bounded, so at this point the system would permanently add to $\mathcal{R}$ the two following rules:

$$s_1 \preceq t \Longrightarrow \texttt{Min1}(s_1, s_2) \preceq t$$

and

$$s_1 \prec t \Longrightarrow \texttt{Min1}(s_1, s_2) \prec t.$$

A similar argument will show Min1 to be 2–bounded as well, so $\mathcal{R}$ would be augmented with two more rules:

$$s_2 \preceq t \Longrightarrow \texttt{Min1}(s_1, s_2) \preceq t$$

$$s_2 \prec t \Longrightarrow \texttt{Min1}(s_1, s_2) \prec t.$$

Subsequently our method would follow the procedure of section 4.5 in an attempt to derive the following strict inequality lemmas, in the given order:

$$GTZ(s_1), s_1 \preceq t \Longrightarrow \texttt{Min1}(s_1, s_2) \prec t$$

$$GTZ(s_2), s_1 \preceq t \Longrightarrow \texttt{Min1}(s_1, s_2) \prec t$$

$$GTZ(s_1, s_2), s_1 \preceq t \Longrightarrow \texttt{Min1}(s_1, s_2) \prec t$$

Since none of these are true, none will be derived. The system would then try to derive the corresponding lemmas for the second argument position, and would fail for the same reason.

The following procedure returns the smallest element in a given list of numbers:

```
Procedure Minimum(l:NatList):NatNum;
Begin
If l = empty Then 0
Else
    If tail(l) = empty Then head(l)
    Else
       Minimum(add(Min1(head(l),head(tail(l))),tail(tail(l)));
End.
```

The termination of `Minimum` is readily verified. First, the measured subset is found to be $\{1\}$, since, in the only recursion there is, we discover that

$$\texttt{add(Min1(head(l),head(tail(l))),tail(tail(l)))} \preceq \texttt{l} \qquad (4.12)$$

through [R4]. Specifically, applying [R4] to (4.12) yields the subgoals $GTZ(\texttt{l})$ and $\texttt{tail(tail(l))} \preceq \texttt{tail(l)}$. The former follows from the contents of $AtEx(\neg \mathrm{E}_1)$, while the latter follows from [R3] and [R1]. The next step is to prove that

$$Holds_{\texttt{Minimum}}(\texttt{add(Min1(head(l),head(tail(l))),tail(tail(l)))} \prec \texttt{l}, \texttt{True}).$$

First we add to $\mathcal{R}$ the expressions in

$$AtEx(\neg(\mathtt{l} = \mathtt{empty})) \cup AtEx(\neg(\mathtt{tail(l)} = \mathtt{empty})) =$$

$$\{\mathtt{l} \neq \mathtt{empty}, \mathtt{tail(l)} \neq \mathtt{empty}\}.$$

Then we apply the only NatList–instance of [R10], generating the subgoals

$$\mathtt{tail(tail(l))} \prec \mathtt{tail(l)}.$$

This matches the head of [R8] (with $g \mapsto \mathtt{tail}$), which splits further into the two subgoals $GTZ(\mathtt{tail(l)})$ and $\mathtt{tail(l)} \preceq \mathtt{tail(l)}$, both of which are readily established. This concludes the proof that Minimum terminates. Argument–boundedness and the associated strict inequality lemmas of section 4.5 are not an issue here since the return type of Minimum and the type of its argument are distinct.

Finally we present a procedure that merges two sorted lists of numbers:

```
Procedure Merge($l_1$,$l_2$:NatList):NatList;
Begin
If $l_1$ = empty Then $l_2$
Else
    If $l_2$ = empty Then $l_1$
    Else
       If Less(head($l_1$),head($l_2$)) = True Then
          add(head($l_1$),Merge(tail($l_1$),$l_2$))
       Else
          add(head($l_2$),Merge($l_1$,tail($l_2$)));
End.
```

In proving the termination of Merge we first compute $MS_{\mathtt{Merge}}$, which is found to be $\{1,2\}$ (as $\mathcal{R} \vdash \mathtt{tail(l_1)} \preceq \mathtt{l_1}$, $\mathcal{R} \vdash \mathtt{l_1} \preceq \mathtt{l_1}$, $\mathcal{R} \vdash \mathtt{l_2} \preceq \mathtt{l_2}$, and $\mathcal{R} \vdash \mathtt{tail(l_2)} \preceq \mathtt{l_2}$). Then, in the first recursion, we find that $\mathtt{tail(l_1)} \prec \mathtt{l_1}$ (from [R8], since $\mathcal{R} \vdash \mathtt{l_1} \preceq \mathtt{l_1}$ by [R1] and $\mathcal{R} \vdash GTZ(\mathtt{l_1})$ by $AtEx(\neg \mathtt{E_1})$). In the second recursion we cannot prove that the first argument decreases, but we can do so for the second one, i.e. for `tail(l₂)`, using essentially the same argument we used in the preceding case.

Our next example is a procedure **DM** that takes a **Wff** and recursively applies DeMorgan's laws to it until all the negation signs have been pushed inwards to atomic propositions. **DM** also eliminates double negations along the way. Hence, the "postcondition" for the result DM(ϕ) is that it contains no subformulas of the form $\neg(\phi_1 \lor \phi_2)$ or $\neg(\phi_1 \land \phi_2)$ or $\neg\neg\phi_1$. In standard notation **DM** can be defined as follows:

$$DM(\phi) = \begin{cases} DM(\neg\phi_1) \lor DM(\neg\phi_2) & \text{if } \phi \equiv \neg(\phi_1 \land \phi_2) \\ DM(\neg\phi_1) \land DM(\neg\phi_2) & \text{if } \phi \equiv \neg(\phi_1 \lor \phi_2) \\ DM(\phi_1) & \text{if } \phi \equiv \neg\neg\phi_1 \\ DM(\phi_1)\ op\ DM(\phi_2) & \text{if } \phi \equiv \phi_1\ op\ \phi_2,\ op \in \{\lor, \land\} \\ \phi & \text{otherwise.} \end{cases}$$

For instance, $DM(\neg(p \land \neg q) \lor r) = (\neg p \lor q) \lor r$, $DM(\neg(\neg\neg p \land (q \lor \neg r))) = \neg p \lor (\neg q \land r)$, and so on.

In IFL we can straightforwardly implement DM as follows:

```
Procedure DM(φ:Wff):Wff;
Begin
If φ = and(l-and(φ),r-and(φ)) Then
   and(DM(l-and(φ)),DM(r-and(φ)))
Else
    If φ = or(l-or(φ),r-or(φ)) Then
       or(DM(l-or(φ)),DM(r-or(φ)))
    Else
       If φ = not(neg(φ)) and neg(φ) = not(neg(neg(φ))) Then
          DM(neg(neg(φ)))
       Else
          If φ = not(neg(φ)) And
          neg(φ) = and(l-and(neg(φ)),r-and(neg(φ))) Then
             or(DM(not(l-and(neg(φ)))),DM(not(r-and(neg(φ)))))
    Else
       If $\phi$ = not(neg($\phi$)) And
       neg($\phi$) = or(l-or(neg($\phi$)),r-or(neg($\phi$))) Then
          and(DM(not(l-or(neg($\phi$)))),DM(not(r-or(neg($\phi$)))))
       Else
          $\phi$;
End.
```

Presumably $\mathcal{R}$ will already contain the following rule instances (amongst others, obtained right after the definition of the type `Wff`):

$$s \preceq t \Longrightarrow \texttt{l-and}(s) \preceq t \tag{4.13}$$

$$s \preceq t \Longrightarrow \texttt{r-and}(s) \preceq t \tag{4.14}$$

$$GTZ(t), s \preceq \texttt{neg}(t) \Longrightarrow \texttt{not}(s) \preceq t \tag{4.15}$$

$$t = \texttt{not}(s) \Longrightarrow GTZ(t) \tag{4.16}$$

$$GTZ(s), s \preceq t \Longrightarrow \texttt{l-and}(s) \prec t \tag{4.17}$$

$$s \prec \texttt{neg}(t) \Longrightarrow \texttt{not}(s) \prec t. \tag{4.18}$$

In computing $MS_{\texttt{DM}}$ we easily see that $\psi \preceq \phi$ for every recursive call $\texttt{DM}(\psi)$ in the first three statements, since in those cases ψ is made up exclusively of selectors. The only slightly tricky cases are the fourth and fifth statements, i.e. those in which ϕ is of the form $\neg(\phi_1 \wedge \phi_2)$ and $\neg(\phi_1 \vee \phi_2)$, respectively. In the former case we need to prove that

$$\texttt{not(l-and(neg(}\phi\texttt{)))} \preceq \phi \tag{4.19}$$

and

$$\texttt{not(r-and(neg(}\phi\texttt{)))} \preceq \phi. \tag{4.20}$$

Now, (4.19) can be derived from rule (4.15), if we can only prove $GTZ(\phi)$ and $\texttt{l-and(neg(}\phi\texttt{))} \preceq \texttt{neg}(\phi)$. The first is immediate from (4.16) since we are given that

ϕ = `not(···)`, while the second follows directly from (4.13) since $\mathtt{neg}(\phi) \preceq \mathtt{neg}(\phi)$. This proves (4.19). A symmetrical argument using rule (4.14) in place of (4.13) will establish (4.20). Finally, similar reasoning will show that $\mathtt{not(l\text{-}or(neg(}\phi\mathtt{)))} \preceq \phi$ in the case where $\phi \equiv \neg(\phi_1 \vee \phi_2)$. Thus we conclude that $MS_{\mathtt{DM}} = \{1\}$.

Now to prove termination we need to show that $Holds_{\mathtt{DM}}(\psi \prec \phi, \mathtt{E_j})$ for every recursive call $\mathtt{DM}(\psi)$ in each $\mathtt{r_j}$. Again, the only cases that are not straightforward are those in which $\phi \equiv \neg(\phi_1 \wedge \phi_2)$ and $\phi \equiv \neg(\phi_1 \vee \phi_2)$. In the first case we must show that $\mathtt{not(l\text{-}and(neg(}\phi\mathtt{)))} \prec \phi$. By rule (4.18), this backchains to $\mathtt{l\text{-}and(neg(}\phi\mathtt{))} \prec \mathtt{neg}(\phi)$, which, in turn, by (4.17), backchains to $GTZ(\mathtt{neg}(\phi))$ and $\mathtt{neg}(\phi) \preceq \mathtt{neg}(\phi)$, both of which are easily established. A symmetrical argument will verify that $\mathtt{not(r\text{-}and(neg(}\phi\mathtt{)))} \prec \phi$ in the case in which $\phi \equiv \neg(\phi_1 \wedge \phi_2)$, while similar reasoning can handle the case $\phi \equiv \neg(\phi_1 \vee \phi_2)$.

Having established termination, we ask whether DM is 1–bounded. The answer of course turns out negative. Consider, for instance,

$$\mathtt{DM}(\neg(\mathtt{p} \wedge \mathtt{q})) = \neg\mathtt{p} \vee \neg\mathtt{q}.$$

Here the size of $\neg(\mathtt{p} \wedge \mathtt{q})$ is two, while the size of the result $\neg\mathtt{p} \vee \neg\mathtt{q}$ is three. Consequently, knowing that our method is sound, we may *a priori* conclude that it will not prove DM to be 1–bounded. Nevertheless, it is instructive to hand-simulate the attempt and see exactly what happens. In particular, it is worth noting that the "proof" goes through fine for the first three statements. For the first one, for example,

```
If φ = and(l-and(φ),r-and(φ)) Then
   and(DM(l-and(φ)),DM(r-and(φ)))
```

we add to $\mathcal{R}$ the two inductive hypotheses

$$\mathtt{l\text{-}and}(\phi) \preceq t \Longrightarrow \mathtt{DM(l\text{-}and(}\phi\mathtt{))} \preceq t$$

$$\mathtt{r\text{-}and}(\phi) \preceq t \Longrightarrow \mathtt{DM(r\text{-}and(}\phi\mathtt{))} \preceq t$$

and we proceed to verify that

$$\texttt{and(DM(l-and(}\phi\texttt{)),DM(r-and(}\phi\texttt{)))} \preceq \phi. \tag{4.21}$$

Presumably $\mathcal{R}$ will already contain the rule

$$GTZ(t), s_1 \preceq \texttt{l-and}(t), s_2 \preceq \texttt{r-and}(t) \Longrightarrow \texttt{and}(s_1, s_2) \preceq t$$

by virtue of which the goal (4.21) backchains to

$$GTZ(\phi) \quad \text{(i)}$$
$$\texttt{DM}(\texttt{l-and}(\phi)) \preceq \texttt{l-and}(\phi) \quad \text{(ii), and}$$
$$\texttt{DM}(\texttt{r-and}(\phi)) \preceq \texttt{r-and}(\phi) \quad \text{(iii).}$$

The subgoal (i) is easily verified through the rule

$$t = \texttt{and}(s_1, s_2) \Longrightarrow GTZ(t)$$

while (ii) and (iii) are readily established owing to the inductive hypotheses. Our method will also breeze through the second case using analogous reasoning; the third case is trivial with the appropriate inductive hypothesis at hand, since $\mathcal{R} \vdash \texttt{neg}(\texttt{neg}(\phi)) \preceq \phi$.

The foregoing argument is an elegant machine proof that the transformation

$$\texttt{DM}(\phi_1 \wedge \phi_2) \Longrightarrow \texttt{DM}(\phi_1) \wedge \texttt{DM}(\phi_2)$$

is size-preserving—on the critical inductive assumptions, of course, that no matter what ϕ_1 and ϕ_2 are, we will have $\texttt{DM}(\phi_1) \preceq \phi_1$ and $\texttt{DM}(\phi_2) \preceq \phi_2$. If the proof were to go through for all six possible cases, these assumptions would be justified and the conclusion that DM is 1–bounded would follow validly by induction. But of course the proof does not go all the way through; it fails on the fourth case, which is exactly where it *should* fail: in the non-size-preserving transformation

$$\mathtt{DM}(\neg(\phi_1 \land \phi_2)) \longrightarrow \mathtt{DM}(\neg\phi_1) \lor \mathtt{DM}(\neg\phi_2).$$

The reader may wish to verify this by carrying out the procedure in detail and discovering exactly where things go wrong.

We end this section with some sorting examples from [8]. We begin with selection sort, which uses the following auxilliary procedure:

```
Procedure Replace(n,m:NatNum;l:NatList):NatList;
Begin
If l = empty Then empty
Else
    If head(l) = n Then add(m,tail(l))
    Else
       add(head(l),Replace(n,m,tail(l)));
End.
```

which replaces the leftmost occurence of n in l with m. Replace is first proven to terminate (straightforward) and then to be 3–bounded, since, with the aid of the inductive hypothesis

$$\mathtt{tail(l)} \preceq t \Longrightarrow \mathtt{Replace(n,m,tail(l))} \preceq t$$

we find that

$$Holds_{\mathtt{Replace}}(\mathtt{add(head(l),Replace(n,m,tail(l)))} \preceq \mathtt{l}, \mathtt{True}) \tag{4.22}$$

by virtue of [R4]. In particular, the goal in (4.22) backchains by [R4] to $GTZ(\mathtt{l})$ and

$$\mathtt{Replace(n,m,tail(l))} \preceq \mathtt{tail(l)}. \tag{4.23}$$

$GTZ(\mathtt{l})$ follows directly from $AtEx(\neg\mathtt{E}_1)$, while (4.23) follows from the inductive hypothesis and [R1]. Accordingly, we augment $\mathcal{R}$ with the following rules:

$$s_3 \preceq t \Longrightarrow \texttt{Replace}(s_1, s_2, s_3) \preceq t \tag{4.24}$$

$$s_3 \prec t \Longrightarrow \texttt{Replace}(s_1, s_2, s_3) \prec t. \tag{4.25}$$

Finally, we attempt to find a subset $A \subseteq \{s_1, s_2, s_3\}$ such that

$$GTZ(A) \Longrightarrow \texttt{Replace}(s_1, s_2, s_3) \prec s_3$$

but do not succeed since there is no such subset.

The selection sort is written as follows:

```
Procedure SelectSort(l:NatList):NatList;
Begin
If l = empty Then empty
Else
   If head(l) = Minimum(l) Then add(head(l),SelectSort(tail(l)))
   Else
     add(Minimum(l),SelectSort(Replace(Minimum(l),head(l),tail(l))));
End.
```

In computing $MS_{\texttt{SelectSort}}$ the first recursion is easy; we readily get $\texttt{tail(l)} \preceq \texttt{l}$. In the second recursion we must prove that

$$\texttt{Replace(Minimum(l),head(l),tail(l))} \preceq \texttt{l},$$

which, by rule (4.24), reduces to the straightforward subgoal $\texttt{tail(l)} \preceq \texttt{l}$. Hence $MS_{\texttt{SelectSort}} = \{1\}$. The termination proof is similar: in the first recursion we get $\texttt{tail(l)} \prec \texttt{l}$ (by [R8]), and in the second one we match the goal

$$\texttt{Replace(Minimum(l),head(l),tail(l))} \prec \texttt{l}$$

with the head of rule (4.25), obtaining the subgoal $\texttt{tail(l)} \prec \texttt{l}$, which is again established by [R8], since in both cases we have $\mathtt{l} \neq \mathtt{empty}$.

Interestingly enough, our method also succeeds in proving `SelectSort` to be 1–bounded:

First statement. Trivially, $\mathcal{R} \vdash \mathtt{empty} \preceq \mathtt{l}$.

Second statement. First we add to $\mathcal{R}$ the inductive hypothesis

$$\mathtt{tail(l)} \preceq t \Longrightarrow \mathtt{SelectSort(tail(l))} \preceq t.$$

Then to derive $\mathtt{add(head(l), SelectSort(tail(l)))} \preceq \mathtt{l}$ we use [R4], producing the subgoals $GTZ(\mathtt{l})$ (which is immediate by $AtEx(\neg \mathtt{E_1})$) and

$$\mathtt{SelectSort(tail(l))} \preceq \mathtt{tail(l)}$$

which, by the inductive hypothesis, yields the readily verifiable

$$\mathtt{tail(l)} \preceq \mathtt{tail(l)}.$$

Third statement This is more interesting. First we postulate the hypothesis

$$\mathtt{Replace(Minimum(l), head(l), tail(l))} \preceq t \Longrightarrow$$
$$\mathtt{SelectSort(Replace(Minimum(l), head(l), tail(l)))} \preceq t. \qquad (4.26)$$

Then, using [R4], the goal

$$\mathtt{add(Mimimum(l), SelectSort(Replace(Minimum(l), head(l), tail(l))))} \preceq \mathtt{l}$$

backchains to

$$\mathtt{SelectSort(Replace(Minimum(l), head(l), tail(l)))} \preceq \mathtt{tail(l)}$$

and $GTZ(\mathtt{l})$. The latter is immediate from $\mathtt{l} \neq \mathtt{empty}$. For the former we use

(4.26) to get

$$\texttt{Replace(Minimum(l),head(l),tail(l))} \preceq \texttt{tail(l)}.$$

To this we can apply rule (4.24), which will finally yield the trivial `tail(l)` $\preceq$ `tail(l)`.

Our method will go on and attempt to prove

$$GTZ(s), s \preceq t \Longrightarrow \texttt{SelectSort}(s) \prec t.$$

This, of course, is not true, so it cannot be proved.

Our next example is `MinSort`, which uses the following auxilliary procedure:

```
Procedure DeleteMin(l:NatList):NatList;
Begin
If (l = empty) Or (tail(l) = empty) Then empty
Else
   If Leq(head(l),head(tail(l))) = True Then
      add(head(tail(l)),DeleteMin(add(head(l),tail(tail(l)))))
   Else
      add(head(l),DeleteMin(tail(l)));
End.
```

`DeleteMin(l)` yields a permutation of `l` that contains one less occurence of `Minimum(l)`. Termination is verified easily. The procedure is also found to be 1–bounded, so the following rules are added to $\mathcal{R}$:

$$s \preceq t \Longrightarrow \texttt{DeleteMin}(s) \preceq t \tag{4.27}$$

$$s \prec t \Longrightarrow \texttt{DeleteMin}(s) \prec t. \tag{4.28}$$

Attempting to derive a strict inequality lemma in the manner of section 4.5, we first add to $\mathcal{R}$ the assertion $GTZ(\texttt{l})$ and then we begin by examining the first case. The antecedent of that is inconsistent with the hypothesis $GTZ(\texttt{l})$, so we move to the next case. We add to $\mathcal{R}$ the inductive hypothesis

$$GTZ(\texttt{add(head(l),tail(tail(l)))}),\ \texttt{add(head(l),tail(tail(l)))} \preceq t \Longrightarrow \\ \texttt{DeleteMin(add(head(l),tail(tail(l))))} \prec t \tag{4.29}$$

and we try to derive

$$\texttt{add(head(tail(l)),DeleteMin(add(head(l),tail(tail(l)))))} \prec \texttt{l}.$$

This is done through [R4], yielding the subgoal

$$\texttt{DeleteMin(add(head(l),tail(tail(l))))} \prec \texttt{tail(l)}. \tag{4.30}$$

This subgoal unifies both with the head of (4.28) and with the head of the inductive hypothesis. Which one will be tried first depends on their respective positions in $\mathcal{R}$. Assuming that $\mathcal{R}$ grows at the end (i.e. that new rules are appended to it), rule (4.28) would be tried first, generating the subgoal

$$\texttt{add(head(l),tail(tail(l)))} \prec \texttt{tail(l)}$$

which, trhough [R10], would yield the unsatisfiable

$$\texttt{tail(tail(l))} \prec \texttt{tail(tail(l))}.$$

At this point the interpreter would backtrack and try to establish (4.30) through the inductive hypothesis, producing the subgoals

$$GTZ(\texttt{add(head(l),tail(tail(l)))})$$

and

$$\texttt{add(head(l),tail(tail(l)))} \preceq \texttt{tail(l)}.$$

The first of these is immediate, while the second matches [R4] and will produce $GTZ(\texttt{tail(l)})$ and

$$\texttt{tail(tail(l))} \preceq \texttt{tail(tail(l))}.$$

The former follows from $AtEx(\neg\texttt{E}_1)$ and the latter from [R1], thus the proof is complete. A similar argument will work for the last case, which will establish the lemma and cause the system to extend $\mathcal{R}$ with the following rule:

$$GTZ(s), s \preceq t \Longrightarrow \texttt{DeleteMin}(s) \prec t. \tag{4.31}$$

`MinSort` is implemented as follows:

```
Procedure MinSort(l:NatList):NatList;
Begin
If l = empty Then empty
Else
    add(Minimum(l),MinSort(DeleteMin(l)));
End.
```

Since $\texttt{DeleteMin} \in B_1$, we get $\texttt{DeleteMin(l)} \preceq \texttt{l}$ and hence $MS_{\texttt{MinSort}} = \{1\}$. Termination is established through lemma (4.31), as in the second case of `MinSort` we have $\texttt{l} \neq \texttt{empty}$ and hence $\mathcal{R} \vdash GTZ(\texttt{l})$.

Next we consider `QuickSort`. Walther's implementation of `QuickSort` in [8] uses the following procedures:

```
Procedure RemoveSmaller(n:NatNum;l:NatList):NatList;
Begin
If l = empty Then empty
Else
    If Leq(head(l),n) = True Then RemoveSmaller(n,tail(l))
    Else
       add(head(l),RemoveSmaller(n,tail(l)));
End.
```

```
Procedure RemoveLarger(n:NatNum;l:NatList):  NatList;
Begin
If l = empty Then empty
Else
    If Less(n,head(l)) = True Then RemoveLarger(n,tail(l))
    Else
        add(head(l),RemoveLarger(n,tail(l)));
End.
```

`RemoveLarger(n,l)` returns a copy of `l` with all elements greater than `n` removed; `RemoveSmaller` does the same thing for all elements that are smaller than or equal to `n`. Both procedures are proven to terminate and to be 2–bounded, so the following rules are added to $\mathcal{R}$:

$$s_2 \preceq t \Longrightarrow \texttt{RemoveSmaller}(s_1, s_2) \preceq t \tag{4.32}$$

$$s_2 \prec t \Longrightarrow \texttt{RemoveSmaller}(s_1, s_2) \prec t \tag{4.33}$$

$$s_2 \preceq t \Longrightarrow \texttt{RemoveLarger}(s_1, s_2) \preceq t \tag{4.34}$$

$$s_2 \prec t \Longrightarrow \texttt{RemoveLarger}(s_1, s_2) \prec t. \tag{4.35}$$

`QuickSort` can now be written as follows:

```
Procedure QuickSort(l:NatList):NatList;
Begin
If l = empty Then empty
Else
    Append(QuickSort(RemoveLarger(head(l),tail(l))),
            add(head(l),QuickSort(RemoveSmaller(head(l),tail(l)))));
End.
```

The termination of `QuickSort` is easily verified with the help of rules (4.32)– (4.35). Since `Append` is not an argument–bounded procedure, 2-boundedness cannot be established for `QuickSort`.

Our next example is MergeSort, using the auxilliary procedures RemoveEven and RemoveOdd, which remove all the elements in the even (odd) positions of a given list:

```
Procedure RemoveEven(l:NatList):NatList;
Begin
If l = empty then empty
Else
    If tail(l) = empty Then l
    Else
        add(head(l),RemoveEven(tail(tail(l)));
End.


Procedure RemoveOdd(l:NatList):NatList;
Begin
If (l = empty) Or (tail(l) = empty) Then empty
Else
    add(head(tail(l)),RemoveOdd(tail(tail(l))));
End.
```

Note that

$$(\forall \mathtt{l} \in \mathtt{NatList})\, \mathtt{RemoveOdd}(\mathtt{RemoveEven}(\mathtt{l})) = \mathtt{empty}.$$

Both procedures are proven to be terminating and 1–bounded, thus the following rules are added to $\mathcal{R}$:

$$s_1 \preceq t \Longrightarrow \mathtt{RemoveEven}(s_1) \preceq t \tag{4.36}$$

$$s_1 \prec t \Longrightarrow \mathtt{RemoveEven}(s_1) \prec t \tag{4.37}$$

$$s_1 \preceq t \Longrightarrow \mathtt{RemoveOdd}(s_1) \preceq t \tag{4.38}$$

$$s_1 \prec t \Longrightarrow \mathtt{RemoveOdd}(s_1) \prec t. \tag{4.39}$$

In addition, we manage to prove the following lemma for RemoveOdd:

$$GTZ(s_1), s_1 \preceq t \Longrightarrow \mathtt{RemoveOdd}(s_1) \prec t. \tag{4.40}$$

Unfortunately a similar assertion cannot be proved for **RemoveEven** because it is not true that **RemoveEven(l)** $\prec$ l whenever l $\neq$ **empty**. In particular, this is false for all one-element lists, as **RemoveEven** is the identity on such lists. Observe, however, that the one-element lists are the *only* counter-examples to the statement in question. For all longer lists l we do have **RemoveEven(l)** $\prec$ l, or equivalently,

$$GTZ(\mathtt{tail}(s_1)), s_1 \preceq t \Longrightarrow \mathtt{RemoveEven}(s_1) \prec t. \tag{4.41}$$

We will return to this observation shortly.

Let us now consider the termination of **MergeSort**:

```
Procedure MergeSort(l:NatList):NatList;
Begin
If l = empty Then empty
Else
    If tail(l) = empty Then l
    Else
       Merge(MergeSort(RemoveEven(l)),MergeSort(RemoveOdd(l)));
End.
```

As the first step we note that since both **RemoveEven** and **RemoveOdd** are 1–bounded, we will easily compute the measured subset of **MergeSort** to be $\{1\}$. Next, however, we must prove that, on the assumptions l $\neq$ **empty** and tail(l) $\neq$ **empty**, we have **RemoveEven(l)** $\prec$ l and **RemoveOdd(l)** $\prec$ l. The second is easily handled by (4.40), but the first is a problem. The only available rule we can apply to it is (4.37), which will not work since it backchains to l $\prec$ l. Thus the proof does not go through and termination cannot be established. The missing piece of knowledge responsible for this failure is that the condition tail(l) $\neq$ **empty** entails **RemoveEven(l)** $\prec$ l, i.e. rule (4.41). If (4.41) was in $\mathcal{R}$ the goal **RemoveEven(l)** $\prec$ l would be reduced to the trivial subgoals GTZ(tail(l)) and l $\preceq$ l, and termination would then follow.

This type of difficulty is not peculiar to **RemoveEven**. For many i–bounded procedures $f : T_{par_1} \longrightarrow T$, what determines whether or not $f(w) \prec w$ is not whether

$GTZ(w)$ but rather whether $GTZ(sel(w))$ or $GTZ(sel(sel'(w)))$, etc., where sel, sel', etc., are selectors of T_{par_1}. For instance, in **RemoveEven**, what determines whether or not **RemoveEven(l)** $\prec$ l is not whether $GTZ(l)$ but whether $GTZ(\texttt{tail(l)})$. In general, for any i–bounded procedure f of k parameters, the problem is that our method will only try to derive lemmas of the form

$$GTZ(s_{a_1}, \ldots, s_{a_n}) \Longrightarrow f(s_1, \ldots, s_k) \prec s_i.$$

We can remedy this as follows. Let us say that an atomic expression $s = t$ or $s \neq t$ is a **size expression** iff $Top(t)$ is a constructor symbol, either reflexive or irreflexive. Thus $\texttt{l} \neq \texttt{empty}, \texttt{pred(n)} = \texttt{0}$, and $\texttt{m} = \texttt{succ(pred(m))}$ are all size expressions. Now let $SZ\text{–}SET_f$, the **size set** of f, be the set of all terms s such that $s = t$ or $s \neq t$ is some size expression in the body of f. For instance, the size set of **RemoveEven** is $\{\texttt{l}, \texttt{tail(l)}\}$. Then we modify our algorithm for deriving strict inequality lemmas (section 4.5) so that we iterate through the power-set of $\{par_1, \ldots, par_k\} \cup SZ\text{–}SET_f$, not just the power-set of $\{par_1, \ldots, par_k\}$. Just as before, for every subset A of this set we visit each statement `If Ej Then rj` in the body of f for which $Consistent_f(\texttt{E}_\texttt{j}, A) = \textbf{True}$, we add to $\mathcal{R}$ the assertions $GTZ(A)$ along with the appropriate inductive hypotheses, and then we try to prove that $Holds_f(\texttt{r}_\texttt{j} \prec par_i, \texttt{E}_\texttt{j})$. If we succeed in proving this for all statements $< \texttt{E}_\texttt{j}, \texttt{r}_\texttt{j} >$ for which $\texttt{E}_\texttt{j}$ is consistent with $GTZ(A)$, then we halt and augment $\mathcal{R}$ with the rule

$$GTZ(A), s_i \preceq t \Longrightarrow f(s_1, \ldots, s_k) \prec t.$$

To illustrate, for **RemoveEven** we would consider the power-set of $\{\texttt{l}, \texttt{tail(l)}\}$: as before, we would fail to prove that

$$GTZ(\texttt{l}) \Longrightarrow \texttt{RemoveEven(l)} \prec \texttt{l}$$

but we would succeed in proving

$$GTZ(\texttt{tail(l)}) \Longrightarrow \texttt{RemoveEven(l)} \prec \texttt{l}.$$

As has already been explained, this will enable our method to prove the termination of `MergeSort`.

Our final example is `BubbleSort`. The following procedure returns a copy of the input list in which the smallest element has been "bubbled up" to the right end of the list:

```
Procedure Bubble(l:NatList):NatList;
Begin
If l = empty Or tail(l) = empty Then l
Else
    If Leq(head(l),head(tail(l))) = True Then
       add(head(tail(l)),Bubble(add(head(l),tail(tail(l)))))
    Else
       add(head(l),Bubble(tail(l)));
End.
```

`Bubble` is proven to terminate and to be 1–bounded, so the following rules are added to $\mathcal{R}$:

$$s \preceq t \Longrightarrow \texttt{Bubble}(s) \preceq t \tag{4.42}$$

$$s \prec t \Longrightarrow \texttt{Bubble}(s) \prec t. \tag{4.43}$$

The next procedure removes the last element of the given list:

```
Procedure RemoveLast(l:NatList):NatList;
Begin
If l = empty Or tail(l) = empty Then empty
Else
    add(head(l),RemoveLast(tail(l)));
End.
```

RemoveLast is easily proven to be terminating, 1–bounded, and strictly 1–bounded whenever the input list is non–empty, hence the following rules are added to $\mathcal{R}$:

$$s \preceq t \Longrightarrow \texttt{RemoveLast}(s) \preceq t \tag{4.44}$$

$$s \prec t \Longrightarrow \texttt{RemoveLast}(s) \prec t \tag{4.45}$$

$$GTZ(s), s \preceq t \Longrightarrow \texttt{RemoveLast}(s) \prec t. \tag{4.46}$$

BubbleSort can now be implemented as follows:

```
Procedure BubbleSort(l:NatList):NatList;
Begin
If l = empty Then empty
Else
    add(Last(Bubble(l)),BubbleSort(RemoveLast(Bubble(l))));
End.
```

where Last is a procedure that returns the last element of a given list.

Since RemoveLast is 1–bounded, we readily compute $MS_{\texttt{BubbleSort}} = \{1\}$. Next, to prove RemoveLast(Bubble(l)) $\prec$ l, we can either apply rule (4.45) or rule (4.46). Unfortunately, neither will work. Rule (4.45) will require us to prove Bubble(l) $\prec$ l, which is not even true, while rule (4.46) will require the assertion GTZ(Bubble(l)), which is true (on the assumption l $\neq$ empty) but cannot be derived.

We can get around this difficulty by devising a slightly more sophisticated method for deducing GTZ conclusions. In particular, we want to be able to infer conclusions of the form $GTZ(f(\cdots))$, for $f \in PROC_M$, even in the absence of size equations such as $f(\cdots) = rcons(\cdots)$ or $f(\cdots) \neq ircons(\cdots)$. One way to do this is to employ the same technique we use for deriving strict inequality lemmas. Specifically, we will examine various subsets $A \subseteq \{s_1, \ldots, s_k\} \cup SZ\text{–}SET_f$ in an attempt to derive a lemma of the form

$$GTZ(A) \Longrightarrow GTZ(f(s_1, \ldots, s_k)).$$

More concretely, immediately after f has been defined and proven to terminate, we carry out the following algorithm:

```
For every non-empty subset $A \subseteq \{par_1, \ldots, par_k\} \cup SZ\text{-}SET_f$ Do {
    For $j = 1$ To $m + 1$ Do
        If $Consistent_f(\mathtt{E_j}, A)$ Then {
            Add to $\mathcal{R}$ the assertions $GTZ(A)$;
            For every recursive call $f(t_1, \ldots, t_k)$ in $\mathtt{r_j}$ add to $\mathcal{R}$ the
            inductive hypothesis
                $GTZ(A[par_i \mapsto t_i]) \Longrightarrow GTZ(f(t_1, \ldots, t_k))$;
            $res \longleftarrow Holds_f(GTZ(\mathtt{r_j}), \mathtt{E_j})$;
            Delete from $\mathcal{R}$ everything that was added to it in the
            previous three steps;
            If $(\neg res)$ Then
                $j \longleftarrow m + 2$; }
    If $(j == m + 1)$ Then
        return $A$; }
return $\emptyset$ ;
```

If a non-empty set A is returned by this algorithm, we add to $\mathcal{R}$ the rule

$$GTZ(A) \Longrightarrow GTZ(f(par_1, \ldots, par_k)).$$

We illustrate the method on procedure `Plus`:

```
Procedure Plus(n,m:NatNum):NatNum;
Begin
If m = 0 Then n
Else
    s(Plus(n,pred(m)));
End.
```

Here we have to consider the power-set of $\{\mathtt{n},\mathtt{m}\}$. For the subset $\{\mathtt{n}\}$, we examine the first statement and we find its antecedent $\mathtt{m} = 0$ to be consistent with $GTZ(\mathtt{n})$, so we add to $\mathcal{R}$ the assertion $GTZ(\mathtt{n})$ and then we trivially derive $GTZ(\mathtt{r_1} \equiv \mathtt{n})$.

The second statement is also consistent with $GTZ(\mathtt{n})$, so we add to $\mathcal{R}$ the assertion $GTZ(\mathtt{n})$ along with the inductive hypothesis

$$GTZ(\mathtt{n}) \Longrightarrow GTZ(\mathtt{Plus(n, pred(m))})$$

and we proceed to prove

$$GTZ(\mathtt{s(Plus(n, pred(m)))})$$

which is immediate. At this point the algorithm will successfully return the set $\{\mathtt{n}\}$ and $\mathcal{R}$ will be augmented with the rule

$$GTZ(s_1) \Longrightarrow GTZ(\mathtt{Plus}(s_1, s_2)). \tag{4.47}$$

As a second example, consider the multiplication procedure:

```
Procedure Times(n,m:NatNum):NatNum;
Begin
If n = 0 Then 0
Else
    Plus(m,Times(pred(n),m));
End.
```

Here, beginning with the subset $\{\mathtt{n}\}$, we find the first case to be inconsistent with $GTZ(\mathtt{n})$, so we move to the next statement, whose antecedent is consistent with the said assertion. We thus add to $\mathcal{R}$ the fact $GTZ(\mathtt{n})$ and the hypothesis

$$GTZ(\mathtt{pred(n)}) \Longrightarrow GTZ(\mathtt{Times(pred(n), m)})$$

and we try to derive

$$GTZ(\mathtt{Plus(m, Times(pred(n), m))}).$$

By rule (4.47) this backchains to $GTZ(\mathtt{m})$, which is unprovable. Next we consider the subset $\{\mathtt{m}\}$, which fails from the beginning as it is consistent with the first case

($\texttt{n = 0}$) but we cannot derive $GTZ(\texttt{0})$. Finally, for the subset $\{\texttt{n},\texttt{m}\}$ we reject the first statement as inconsistent with $GTZ(\texttt{n},\texttt{m})$, while, in the second case, rule (4.47) yields the satisfiable $GTZ(\texttt{m})$, so at this point the method has successfully inferred the lemma $n > 0, m > 0 \Longrightarrow n \cdot m > 0$. Upon conclusion the set $\{\texttt{n},\texttt{m}\}$ is returned and $\mathcal{R}$ is extended with the rule

$$GTZ(s_1, s_2) \Longrightarrow GTZ(\texttt{Times}(s_1, s_2)).$$

Notice that rules of the form $GTZ(\cdots) \Longrightarrow GTZ(\cdots)$ do not have a significant effect on the complexity of the system. The worst-case complexity of deciding a $GTZ(\cdots)$ goal is still meager: if $t \equiv f(\cdots)$ and f is the i^{th} procedure defined in M (in temporal order), then the goal $GTZ(f(\cdots))$ will backchain at most i times[8]; after that it must ultimately be decided using rules [R5] and [R6] only, which, as we have seen, is a task of negligible difficulty (it is essentially a matter of looking up size equations in $\mathcal{R}$). Returning to `BubbleSort`, it is now easy to see that the foregoing technique will manage to prove the lemma

$$GTZ(s) \Longrightarrow GTZ(\texttt{Bubble}(s))$$

which will be used subsequently to establish the termination of `BubbleSort`.

[8]And that assumes that we have derived rules of the form $GTZ(\cdots) \Longrightarrow GTZ(g(\cdots))$ for every single procedure g defined in M prior to f, which is very unlikely.

Appendix A

Proofs

Theorem A.1.1 (Interpreter completeness) *For any module M and term $t \in \mathcal{T}(\Sigma_M)$, $Eval_M(t)$ returns a term w iff w is an object in $\mathcal{T}(\mathrm{CON}_M)$ such that $t \Longrightarrow_{\mathcal{T}}^{*} w$.*

Proof. If part. The key observation is the following:

Lemma A.1.1 *For all terms $s, t \in \mathcal{T}(\Sigma_M)$, if $s \Longrightarrow_{\mathcal{T}} t$ then $Eval_M(s) = Eval_M(t)$*[1].

Proof. By strong induction on the rewrite cost of the transition $s \Longrightarrow_{\mathcal{T}} t$. There are three cases to be distinguished:

Case 1: If $s \Longrightarrow_{\mathcal{T}} t$ by [T1], then

$$s \equiv g(w_1, \ldots, w_n, u_1, u_2, \ldots, u_m),\ t \equiv g(w_1, \ldots, w_n, u_1', u_2, \ldots, u_m),$$

and $u_1 \Longrightarrow_{\mathcal{T}} u_1'$. By the inductive hypothesis, $Eval_M(u_1) = Eval_M(u_1')$, and this entails that $Eval_M(s) = Eval_M(t)$.

Case 2: $s \Longrightarrow_{\mathcal{T}} t$ by [T2] or [T3], so that $Top(s) \in SEL_M$ and $s \downarrow 1, t \in \mathcal{T}(CON_M)$. In that case it is easy to see from lines 5-7 in the listing of $Eval_M$ that $Eval_M(s) =$

[1]Here the equality $Eval_M(s) = Eval_M(t)$ means that the outcome of evaluating s is identical to that of evaluating t, and is meaningful even in cases in which the value of either side is undefined. In particular, it says that if $Eval_M(s)$ diverges then $Eval_M(t)$ diverges as well, and vice versa; and that if $Eval_M(s)$ generates an error, then the exact same error is produced by computing $Eval_M(t)$, and conversely.

$t = Eval_M(t)$.

Case 3: If $s \Longrightarrow_T t$ by virtue of [T4], then $s \equiv f(w_1, \ldots, w_k)$ for some $f \in PROC_M$ with k parameters $par_1, \ldots, par_k$ and body

$$[<\mathtt{E}_1, \mathtt{r}_1>, \ldots, <\mathtt{E}_m, \mathtt{r}_m>, <\mathtt{True}, \mathtt{r}_{m+1}>]$$

and $t \equiv \mathtt{r}_{j+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]$ for some $\mathtt{j} \leq \mathtt{m}$ such that

$$\begin{aligned}
&\mathtt{E}_1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv E_1^1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B E_1^2[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B \cdots \\
&\qquad \cdots \Longrightarrow_B E_1^{n_1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B E_1^{n_1+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \mathbf{False} \\
&\qquad\qquad \vdots \\
&\mathtt{E}_j[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv E_j^1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B E_j^2[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B \cdots \\
&\qquad \cdots \Longrightarrow_B E_j^{n_j}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B E_j^{n_j+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \mathbf{False} \\
&\mathtt{E}_{j+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv E_{j+1}^1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B E_{j+1}^2[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B \cdots \\
&\qquad \cdots \Longrightarrow_B E_{j+1}^{n_{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_B E_{j+1}^{n_{j+1}+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \mathbf{True}.
\end{aligned}$$

The crucial observation is that

$$TEval_M(\mathtt{E}_1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) = \mathbf{False}$$

$$\vdots$$

$$TEval_M(\mathtt{E}_j[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) = \mathbf{False},$$

while $TEval_M(\mathtt{E}_{j+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) = \mathbf{True}$. This can be viewed as a consequence of the following more general proposition:

Proposition A.1.1 *For all $i \in \{1, \ldots, j+1\}$ and $\gamma \in \{1, \ldots, n_i\}$,*

$$TEval_M(E_i^\gamma[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) = TEval_M(E_i^{\gamma+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]).$$

Proof. By induction on the structure of $\phi \equiv E_i^\gamma[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]$. If $\phi \equiv (s = t)$, then $\phi \Longrightarrow_B E_i^{\gamma+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \psi$ either by one of {[B1],[B2]} or by one of {[B3],[B4],[B5],[B6]}. In the latter case the result follows trivially from the definition of $TEval_M$, lemma 2.5.1, and the fact that $TEval_M(TV) = TV$ for $TV \in$ {**True**,**False**}. In the former case, suppose $\phi \Longrightarrow_B \psi$ by [B1], so $\phi \equiv (u_1\, op\, u_2), \psi \equiv$

$(u_1' \; op \; u_2), op \in \{=, \neq\}$, and $u_1 \Longrightarrow_T u_1'$. By the (outer) inductive hypothesis,

$$Eval_M(u_1) = Eval_M(u_1')$$

and the result follows from the definition of $TEval_M$. A similar argument works if $\phi \Longrightarrow_B \psi$ in virtue of [B2], and thus the base case is complete. The inductive step (covering the cases in which ϕ is a conjunction or a disjunction) is trivial, and the proposition is established by structural induction. □

Now the above proposition implies that

$TEval_M(\mathtt{E}_1[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) = TEval_M(E_1^2[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) =$

$$\cdots = TEval_M(E_1^{n_1+1}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \equiv \textbf{False}) = \textbf{False} \; \ldots$$

and, in a similar fashion,

$$TEval_M(\mathtt{E}_\mathtt{j}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] = \textbf{False}, \; TEval_M(\mathtt{E}_{\mathtt{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) = \textbf{True}.$$

But then, given that $Eval_M(w_i) = w_i$ for $i = 1, \ldots, k$, lines **9-14** in the listing of $Eval_M$ show that

$$Eval_M(f(w_1, \ldots, w_k)) = Eval_M(\mathtt{r}_{\mathtt{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]).$$

The above cases exhaust all possibilities and the result follows by strong induction. ◇

Now suppose that w is an object such that $t \Longrightarrow_T^* w$. By a simple induction on the length of the $t \Longrightarrow_T^* w$ path we conclude, on the basis of the above lemma, that $Eval_M(t) = Eval_M(w)$. But, by lemma 2.5.1, $Eval_M(w) = w$, hence $Eval_M(t) = w$.

Only-if part. By strong induction on τ, the number of recursive invocations of $Eval_M$ spawned by the call $Eval_M(t)$. If we let $\sigma \equiv Top(t)$, there are three cases:

Case 1: $\sigma \in CON_M$. In that case we must have $\#(t) > 0$, hence the term that will be returned by $Eval_M(t)$ is $\sigma(Eval_M(t \downarrow 1), \ldots, Eval_M(t \downarrow \#(t)))$. Now, by the inductive hypothesis, each $Eval_M(t \downarrow i)$ will return an object w_i such that $t \downarrow$

$i \Longrightarrow_{\mathcal{T}}^{*} w_i$. But then $\sigma(w_1, \ldots, w_k)$ is also an object[2], and, furthermore, by repeated applications of the term evaluation progress rule [T1], we conclude that

$$t \equiv \sigma(t \downarrow 1, \ldots, t \downarrow \#(t)) \Longrightarrow_{\mathcal{T}}^{*} \sigma(w_1, \ldots, w_k).$$

Case 2: $\sigma \in SEL_M$. In that case $t \equiv sel_c^i(t_1)$ for some $c \in CON_T$, $T \in TYPES_M$, and, by the inductive hypothesis, $Eval_M(t_1)$ will return an object w_1 such that $t_1 \Longrightarrow_{\mathcal{T}}^{*} w_1$. Again by repeated applications of [T1] we conclude that $t \Longrightarrow_{\mathcal{T}}^{*} sel_c^i(w_1)$ (a). Because t is well-typed (by assumption, since $t \in \mathcal{T}(\Sigma_M)$) and $Eval_M$ preserves type, we must have $w_i \in \mathcal{T}(CON_M^T)$, i.e. $Top(w_1) \in CON_T$. Now if $Top(w_1) \equiv c$ then, by [T2], $sel_c^i(w_1) \Longrightarrow_{\mathcal{T}} w_1 \downarrow i$ (b), and, by combining (a) with (b) we are done, since $w_1 \downarrow i$ is precisely the term that will be returned by $Eval_M(t)$. On the other hand, if $Top(t) \not\equiv c$ then $Eval_M(t) = Minimal_M(T_i)$, where T_i is the i^{th} component type of c, and then by [T3] we will have $sel_c^i(w_1) \Longrightarrow_{\mathcal{T}} Minimal_M(T_i)$ (c), so the result follows from (a), (c), and the transitivity of $\Longrightarrow_{\mathcal{T}}^{*}$.

Case 3: $\sigma \in PROC_M$. In that case $t \equiv f(t_1, \ldots, t_k)$ for some $f \in PROC_M$ with k parameters $par_1, \ldots, par_k$ and body

$$[< \mathtt{E_1}, \mathtt{r_1} >, \ldots, < \mathtt{E_m}, \mathtt{r_m} >, < \mathtt{True}, \mathtt{r_{m+1}} >].$$

By the inductive hypothesis, for each $i = 1, \ldots, k$, $Eval_M(t_i)$ will return an object w_i such that $t_i \Longrightarrow_{\mathcal{T}}^{*} w_i$, so, by repeated applications of [T1], it follows that $t \Longrightarrow_{\mathcal{T}}^{*} f(w_1, \ldots, w_k)$ (i). Now by inspecting lines `12-14` in the listing of $Eval_M$ we can see that $Eval_M(t)$ will return the term $Eval_M(\mathtt{r_{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}])$ for some number $\mathtt{j} \leq \mathtt{m}$ such that $TEval_M(\mathtt{E_i}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}])$ = **False** (ii) for every $\mathtt{i} = \mathtt{1}, \ldots, \mathtt{j}$, and $TEval_M(\mathtt{E_{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}])$ = **True** (iii).

Therefore, if we can only prove that

$$\mathtt{E_i}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}}^{*} \textbf{False} \text{ for each } \mathtt{i} = \mathtt{1}, \ldots, \mathtt{j} \qquad (A.1)$$

[2]Note that each w_i object must be of the appropriate type (namely, the i^{th} component type of σ) since t is assumed to be well-typed and $Eval_M$ preserves type (lemma 2.5.2).

and

$$\mathtt{E}_{\mathtt{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}}^{*} \textbf{True}, \tag{A.2}$$

it will follow from [T4] that

$$f(w_1,\ldots,w_k) \Longrightarrow_{\mathcal{T}} \mathtt{r}_{\mathtt{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \tag{A.3}$$

and then eq.(A.3) in tandem with (i) will establish that $t \Longrightarrow_{\mathcal{T}}^{*} \mathtt{r}_{\mathtt{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]$ (iv); but then, again by the inductive hypothesis, we will be able to conclude that $Eval_M(\mathtt{r}_{\mathtt{j+1}}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}])$ will return an object w such that $\mathtt{r}_{\mathtt{j+1}} \Longrightarrow_{\mathcal{T}}^{*} w$ (v), and at that point {(iv),(v)} will entail that $t \Longrightarrow_{\mathcal{T}}^{*} w$, and the proof will be complete since w is the term that will be returned by the call $Eval_M(t)$.

Now eq.(A.1) and eq.(A.2) can be derived from the following more general observation:

Proposition A.1.2 *For all* $\mathtt{i} \in \{\mathtt{1},\ldots,\mathtt{j+1}\}$ *and* $TV \in$ {**True**,**False**},
if $TEval_M(\mathtt{E_i}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]) = TV$ *then* $\mathtt{E_i}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}] \Longrightarrow_{\mathcal{B}}^{*} TV$.

The above can be easily proved by (nested) induction on the structure of $\mathtt{E_i}[\overrightarrow{par_i} \mapsto \overrightarrow{w_i}]$: if the latter is of the form $(u_1\ op\ u_2)$ for $op \in \{=,\neq\}$, then, by the (outer) inductive hypothesis, $Eval_M(u_1)$ and $Eval_M(u_2)$ will respectively return two objects w_1 and w_2 such that $u_1 \Longrightarrow_{\mathcal{T}}^{*} w_1, u_2 \Longrightarrow_{\mathcal{T}}^{*} w_2$. Then by repeated applications of [B1] and [B2] we get $(u_1\ op\ u_2) \Longrightarrow_{\mathcal{B}}^{*} (w_1\ op\ w_2)$. But now it is clear that $(w_1\ op\ w_2) \Longrightarrow_{\mathcal{B}} TEval_M((w_1\ op\ w_2))$, hence we conclude that

$$(u_1\ op\ u_2) \Longrightarrow_{\mathcal{B}}^{*} TEval_M((w_1\ op\ w_2)) = TEval_M((u_1\ op\ u_2)).$$

The inductive step is even simpler, and this concludes the inner inductive argument as well as the original proof. □

Proposition A.1.3 *Let M' be the module obtained by extending an admissible module M with a normal procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$ and let $w_1,\ldots,w_k$ be any* *k objects of type $T_{par_1}, \ldots, T_{par_k}$, respectively. If $f(w_1, \ldots, w_k)$ has a finite recursion tree, then $Eval_{M'}(f(w_1, \ldots, w_k)) \downarrow$.*

Proof. By double strong induction on the size of $RT_{f(w_1,\ldots,w_k)}$, the recursion tree of the term $f(w_1, \ldots, w_k)$. Let `If` $\mathtt{E_{j+1}}$ `Then` $\mathtt{r_{j+1}}$ be the unique statement in the body of f such that $TEval_{M'}(\mathtt{E_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]) =$ **True**. Now because the various conditions `E` in the body of f do not contain the symbol f (since f is normal), we have

$$TEval_M(\mathtt{E_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = TEval_{M'}(\mathtt{E_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = \mathbf{True} \tag{A.4}$$

while

$$TEval_M(\mathtt{E_i}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = TEval_{M'}(\mathtt{E_i}[\overrightarrow{par} \mapsto \overrightarrow{w}]) = \mathbf{False} \tag{A.5}$$

for $\mathtt{1} \leq \mathtt{i} \leq \mathtt{j}$. So, to show that $Eval_{M'}$ halts on $f(w_1, \ldots, w_k)$, we only need to show that $Eval_{M'}$ halts on $\mathtt{r_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]$. We show the latter by proving the stronger statement that $Eval_{M'}(s) \downarrow$ for every $s \sqsubseteq \mathtt{r_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]$, by (nested) strong induction on the size of s.

Suppose $s \equiv g(s_1, \ldots, s_n)$ for some $n \geq 0$ and assume the result for all subterms of $\mathtt{r_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]$ of size smaller than that of $g(s_1, \ldots, s_n)$ (this is the *inner* inductive hypothesis). Now either $g \equiv f$ or not. Suppose $g \not\equiv f$. By the inner inductive hypothesis we get $Eval_{M'}(s_i) \downarrow$ for $i = 1, \ldots, n$, hence corollary 2.5.1 entails that $Eval_{M'}(s_i) \in \mathcal{T}(CON_{M'}) = \mathcal{T}(CON_M)$ for each such i. It follows that $g(Eval_{M'}(s_1), \ldots, Eval_{M'}(s_n)) \in \mathcal{T}(\Sigma_M)$, so, by lemma 2.5.4.a,

$$\begin{aligned} &Eval_{M'}(g(Eval_{M'}(s_1), \ldots, Eval_{M'}(s_n))) = \\ &\quad Eval_M(g(Eval_{M'}(s_1), \ldots, Eval_{M'}(s_n))). \end{aligned} \tag{A.6}$$

Now, by the admissibility of M and theorem 2.5.3 we see that

$$Eval_M(g(Eval_{M'}(s_1), \ldots, Eval_{M'}(s_n))) \downarrow$$

so, by (A.6), we conclude that $Eval_{M'}(g(Eval_{M'}(s_1), \ldots, Eval_{M'}(s_n))) \downarrow$. But then it follows from lemma 2.5.5 that $Eval_{M'}(g(s_1, \ldots, s_n)) \downarrow$. Now suppose that $g \equiv f$, so that $n = k$. Then, because f is normal, no s_i contains a recursive call to f, thus $s_i \in \mathcal{T}(\Sigma_M)$ for $i = 1, \ldots, n$ and $Eval_{M'}(s_i) = Eval_M(s_i)$ (lemma 2.5.4). But, by theorem 2.5.3 and the admissibility of M, we get $Eval_M(s_i) \downarrow$, thus we also have $Eval_{M'}(s_i) \downarrow$ for all $i \in \{1, \ldots, n\}$. Next, corollary 2.5.1 implies that $Eval_{M'}(s_i) \in \mathcal{T}(CON_{M'}) = \mathcal{T}(CON_M)$, and since $f(s_1, \ldots, s_k)$ is well-typed, we conclude from lemma 2.5.2 that each term $Eval_{M'}(s_i)$ is an object of type T_{par_i}. But now observe that the recursion tree of $f(Eval_{M'}(s_1), \ldots, Eval_{M'}(s_k))$ is a proper subtree of $RT_{f(w_1,\ldots,w_k)}$, so from the (outer) inductive hypothesis it follows that

$$Eval_{M'}(f(Eval_{M'}(s_1), \ldots, Eval_{M'}(s_k))) \downarrow .$$

At this point lemma 2.5.5 yields the desired conclusion that

$$Eval_{M'}(f(s_1, \ldots, s_n)) \downarrow .$$

We have now shown that $Eval_{M'}$ halts for every subterm of $\mathtt{r}_{\mathtt{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]$, thus including $\mathtt{r}_{\mathtt{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{w}]$ itself, and both inductive arguments are complete. □

Proposition A.1.4 *Let M' be the module obtained by augmenting an admissible module M with a normal procedure $f : T_{par_1} \times \cdots \times T_{par_k} \longrightarrow T$, and let $w_1, \ldots, w_k$ be any k objects of type $T_{par_1}, \ldots, T_{par_k}$ respectively. Then for every finite ordinal $\beta < ORD(RT_{f(w_1,\ldots,w_k)})$ there are terms $u_1, \ldots, u_k$ in $\mathcal{T}(\Sigma_M)$ such that*

$$Eval_{M'}(f(w_1, \ldots, w_k)) \Longrightarrow^* Eval_{M'}(f(u_1, \ldots, u_k))$$

where $Eval_{M'}(u_i) = (\pi_{RT_{f(w_1,\ldots,w_k)}}(\beta)) \downarrow i$ for every $i = 1, \ldots, k$.

Proof. By double strong induction on β. Let β be any non-zero finite ordinal strictly less than $ORD(RT_{f(w_1,\ldots,w_k)})$ and assume the result for all similar finite ordinals $< \beta$. Let $t \equiv f(t_1, \ldots, t_k)$ be $\pi(\beta)$ and let $p_t \equiv f(s_1, \ldots, s_k)$ be the parent of t in $RT_{f(w_1,\ldots,w_k)}$ ($t = \pi(\beta)$ must have a parent since $\beta > 0$). Since $\pi^{-1}(p_t) < \pi^{-1}(t)$

(equivalently, $Pos(p_t) <' Pos(t)$), the inductive hypothesis implies that

$$Eval_{M'}(f(w_1, \ldots, w_k)) \Longrightarrow^* Eval_{M'}(f(u_1, \ldots, u_k)) \tag{A.7}$$

for some terms $u_1, \ldots, u_k$ in $\mathcal{T}(\Sigma_M)$ such that

$$Eval_{M'}(u_i) = p_t \downarrow i = s_i \quad \text{for each } i \in \{1, \ldots, k\}. \tag{A.8}$$

We will proceed to show that there are terms $v_1, \ldots, v_k$ in $\mathcal{T}(\Sigma_M)$ such that

$$Eval_{M'}(f(u_1, \ldots, u_k)) \Longrightarrow^* Eval_{M'}(f(v_1, \ldots, v_k)) \tag{A.9}$$

and

$$Eval_{M'}(v_i) = t \downarrow i = t_i. \tag{A.10}$$

The result will then follow from (A.7) and (A.9) by the transitivity of $\Longrightarrow^*$ and (A.10).

Let `If` $\mathtt{E_{j+1}}$ `Then` $\mathtt{r_{j+1}}$ be the unique statement in the body of f such that

$$\begin{aligned} TEval_{M'}(\mathtt{E_{j+1}}[par_1 \mapsto Eval_{M'}(u_1), \ldots, par_k \mapsto Eval_{M'}(u_k)]) &= \\ TEval_{M'}(\mathtt{E_{j+1}}[par_1 \mapsto s_1, \ldots, par_k \mapsto s_k]) &= \\ TEval_{M}(\mathtt{E_{j+1}}[par_1 \mapsto s_1, \ldots, par_k \mapsto s_k]) &= \textbf{True}. \end{aligned}$$

By the same analysis that was carried out in the proof of the converse proposition, we can show that

$$TEval_{M'}(\mathtt{E_i}[par_1 \mapsto Eval_{M'}(u_1), \ldots, par_k \mapsto Eval_{M'}(u_k)]) = \textbf{False}$$

for all $\mathtt{i} \in \{\mathtt{1}, \ldots, \mathtt{j}\}$, hence we conclude that

$$Eval_{M'}(f(u_1, \ldots, u_k)) \Longrightarrow^* Eval_{M'}(\mathtt{r_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{s}]). \tag{A.11}$$

Now let $f(t_1^1, \ldots, t_k^1), \ldots, f(t_1^n, \ldots, t_k^n)$ be the $n > 0$ recursive calls contained in $\mathtt{r_{j+1}}$, ordered in the usual manner. Then the n children of $p_t \equiv f(s_1, \ldots, s_k)$ in $RT_{f(w_1,\ldots,w_k)}$ must be

$$\delta_1 \equiv f(Eval_M(t_1^1[\overrightarrow{par} \mapsto \overrightarrow{s}]), \ldots, Eval_M(t_k^1[\overrightarrow{par} \mapsto \overrightarrow{s}]))$$

$$\vdots$$

$$\delta_n \equiv f(Eval_M(t_1^n[\overrightarrow{par} \mapsto \overrightarrow{s}]), \ldots, Eval_M(t_k^n[\overrightarrow{par} \mapsto \overrightarrow{s}])).$$

One of these must be $t \equiv f(t_1, \ldots, t_k)$, say it is δ_m for some $m \in \{1, \ldots, n\}$, so that

$$t \downarrow i = t_i = Eval_M(t_i^m[\overrightarrow{par} \mapsto \overrightarrow{s}]) = Eval_{M'}(t_i^m[\overrightarrow{par} \mapsto \overrightarrow{s}]) \qquad \text{(A.12)}$$

for every $i = 1, \ldots, k$.
We now prove the following assertion: $Eval_{m'}(r) \downarrow$ for every $\mathtt{r} \sqsubset \mathtt{r_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{s}]$ to the left of

$$f(t_1^m, \ldots, t_k^m)[\overrightarrow{par} \mapsto \overrightarrow{s}] \equiv f(t_1^m[\overrightarrow{par} \mapsto \overrightarrow{s}], \ldots, t_k^m[\overrightarrow{par} \mapsto \overrightarrow{s}]) \equiv f(v_1, \ldots, v_k)$$

that is not a superterm of $f(v_1, \ldots, v_k)$ (note that we have set $v_i = t_i^m[\overrightarrow{par} \mapsto \overrightarrow{s}]$). This is easily done by (nested) strong induction on the size of $\mathtt{r}$. Let $\mathtt{r} \equiv g(p_1, \ldots, p_d)$ be a proper subterm of $\mathtt{r_{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{s}]$ to the left of $f(v_1, \ldots, v_k)$ that does not contain $f(v_1, \ldots, v_k)$ and postulate the (inner) inductive hypothesis. Then either $g \equiv f$ or not. If $g \not\equiv f$ then, by the inner inductive hypothesis, $Eval_{M'}(p_i) \downarrow$ for each $i = 1, \ldots, d$. Furthermore, we can use corollary 2.5.1 to infer that each $Eval_{M'}(p_i)$ is an object in $\mathcal{T}(CON_{M'}) = \mathcal{T}(CON_M)$, and hence that $g(Eval_{M'}(p_1), \ldots, Eval_{M'}(p_d)) \in \mathcal{T}(\Sigma_M)$. Therefore, by theorem 2.5.3 we conclude that

$$Eval_M(g(Eval_{M'}(p_1), \ldots, Eval_{M'}(p_d))) \downarrow$$

so, by lemma 2.5.4.a, we get

$$Eval_{M'}(g(Eval_{M'}(p_1), \ldots, Eval_{M'}(p_d))) \downarrow$$

and, finally, by lemma 2.5.5, that $Eval_{M'}(g(p_1,\ldots,p_d))\downarrow$.

Now suppose that $g\equiv f$. Then we must have

$$\mathbf{r}\equiv f(t_1^\alpha,\ldots,t_k^\alpha)[\overrightarrow{par}\mapsto\overrightarrow{s}]\equiv f(t_1^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}],\ldots,t_k^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}])$$

for some $\alpha<m$ (since $\mathbf{r}$ is to the left of $f(t_1^m,\ldots,t_k^m)[\overrightarrow{par}\mapsto\overrightarrow{s}]$). Since f is normal, $\mathbf{r}$ cannot properly contain a recursive call to f, hence $t_i^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}]\in\mathcal{T}(\Sigma_M)$ for each $i=1,\ldots,k$. Hence, by theorem 2.5.3 and lemma 2.5.4.a we conclude that

$$Eval_{M'}(t_i^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}])\downarrow$$

for each such i. Moreover, from corollary 2.5.1, the fact that $\mathbf{r}$ is well-typed, and lemma 2.5.2, we infer that each term $Eval_{M'}(t_i^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}])$ is an object of type T_{par_i}. Now the crucial observation is that because β is finite and the recursion tree of

$$\sigma\equiv f(Eval_{M'}(t_1^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}]),\ldots,Eval_{M'}(t_k^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}]))$$

is to the left of the recursion tree of

$$t\equiv f(t_1,\ldots,t_k)\equiv\delta_m\equiv f(Eval_M(t_1^m[\overrightarrow{par}\mapsto\overrightarrow{s}]),\ldots,Eval_M(t_k^m[\overrightarrow{par}\mapsto\overrightarrow{s}]))$$

in $RT_{f(w_1,\ldots,w_k)}$ (since $\alpha<m$), it must be that the recursion tree of σ is finite. Therefore, by proposition 3.5.1 we conclude that $Eval_{M'}(\sigma)\downarrow$, i.e. that

$$Eval_{M'}(f(Eval_{M'}(t_1^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}]),\ldots,Eval_{M'}(t_1^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}])))\downarrow$$

and, by lemma 2.5.5, that

$$Eval_{M'}(f(t_1^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}],\ldots,t_1^\alpha[\overrightarrow{par}\mapsto\overrightarrow{s}]))\downarrow$$

i.e. that $Eval_{M'}(\mathbf{r})\downarrow$.

Having established that $Eval_{M'}(\mathbf{r})\downarrow$ for every proper subterm of $\mathbf{r}_{j+1}[\overrightarrow{par}\mapsto\overrightarrow{s}]$

to the left of $f(v_1, \ldots, v_k)$ that is not a superterm of the latter, we can infer from lemma 2.5.3 that

$$Eval_{M'}(\mathtt{r}_{\mathtt{j+1}}[\overrightarrow{par} \mapsto \overrightarrow{s}]) \Longrightarrow^* Eval_{M'}(f(v_1, \ldots, v_k)). \tag{A.13}$$

Finally, from (A.11) and (A.13) it follows that

$$Eval_{M'}(f(u_1, \ldots, u_k)) \Longrightarrow^* Eval_{M'}(f(v_1, \ldots, v_k))$$

and the result follows by strong induction since

$$Eval_{M'}(v_i) = Eval_{M'}(t_i^m[\overrightarrow{par} \mapsto \overrightarrow{s}]) = t_i = t \downarrow i$$

(by (A.12)) for every $i = 1, \ldots, k$. □

Appendix B

Modules and first-order theories

Every admissible module M gives rise to a unique multi-sorted first-order theory FOT_M in a very natural way. In particular, the language of FOT_M consists of the signature Σ_M, along with the variables of V_M, the equality symbol $=$, the usual logical connectives $(\neg, \wedge, \vee, \Longrightarrow)$, and the two quantifiers $\forall$ and $\exists$. The terms of each type T are defined as usual (see section 2.3.1), and then the atomic formulas are taken to be all expressions of the form $(s = t)$, for any two terms s and t[1]. Finally, wffs are defined as follows:

- Every atomic formula is a wff.
- If ϕ and ψ are wffs, then so are $\neg\phi$, (ϕ), $[\phi]$, $\phi \wedge \psi$, $\phi \vee \psi$, and $\phi \Longrightarrow \psi$.
- If ϕ is a wff containing a variable $v \in V_T$, then $(\forall v \in T)\phi$ and $(\exists v \in T)\phi$ are wffs.

Free variables are defined as usual and then those wffs that contain no free variables are singled out as the **sentences** of FOT_M. An example is the sentence

$$(\forall n_1 \in \texttt{NatNum})(\forall n_2 \in \texttt{NatNum})[\texttt{Plus}(n_1, n_2) = \texttt{Plus}(n_2, n_1)]$$

[1]We do not impose the restriction that s and t be of the same type; in such cases $(s = t)$ will simply be false.

which asserts that binary addition is commutative. The *axioms* of FOT_M are $TAX_M \cup PAX_M$, where

$$TAX_M = \bigcup_{T \in TYPES_M} AX_T \quad \text{and} \quad PAX_M = \bigcup_{f \in PROC_M} AX_f.$$

Each AX_T is a set of axioms "describing" the data type T (similarly, AX_f is a set of axioms describing procedure f). $AX_{\texttt{NatNum}}$, for example, comprises the following axioms:

$$(\forall n \in \texttt{NatNum})\,[\texttt{0} \neq \texttt{succ}(n)] \tag{B.1}$$

$$(\forall n \in \texttt{NatNum})(\forall m \in \texttt{NatNum})\,[\texttt{succ}(n) = \texttt{succ}(m) \implies n = m] \tag{B.2}$$

$$(\forall n \in \texttt{NatNum})\,[(n = \texttt{0}) \vee (n = \texttt{succ(pred(}n\texttt{)))}] \tag{B.3}$$

$$(\forall n \in \texttt{NatNum})\,[\texttt{pred(succ(}n\texttt{))} = n] \tag{B.4}$$

$$\texttt{pred(0)} = \texttt{0} \tag{B.5}$$

It should be remarked that AX_T can be computed mechanically just by inspecting the definition of T, and in time linear in the definition's length. The same is true for AX_f, the axioms of which can be obtained algorithmically merely by universally quantifying the $\texttt{m}$ sentences

$$[\,\texttt{E}_\texttt{j} \bigwedge_{\texttt{i<j}} \neg\texttt{E}_\texttt{i} \implies \texttt{f}(\cdots) = \texttt{r}_\texttt{j}\,]$$

$\texttt{j} = \texttt{1}, \ldots, \texttt{m}+\texttt{1}$, where $[<\texttt{E}_1, \texttt{r}_1>, \ldots, <\texttt{E}_\texttt{m}, \texttt{r}_\texttt{m}>, <\texttt{True}, \texttt{r}_{\texttt{m}+1}>]$ is the body of f. For instance, $AX_{\texttt{Append}}$ comprises the following two sentences:

$$(\forall l_1 \in \texttt{NatList})(\forall l_2 \in \texttt{NatList})\;[\,(l_1 = \texttt{empty}) \implies \texttt{Append}(l_1, l_2) = l_2\,]$$

$$(\forall l_1 \in \texttt{NatList})(\forall l_2 \in \texttt{NatList})$$
$$[\,(l_1 \neq \texttt{empty}) \implies \texttt{Append}(l_1, l_2) = \texttt{add(head}(l_1), \texttt{Append(tail}(l_1), l_2))\,].$$

Next, we define the **standard interpretation** of FOT_M, notated by $\mathcal{I}_M$, as follows. The domain ("universe of discourse") is $\mathcal{T}(CON_M)$, i.e. the class of all objects. To each function symbol $g : T_1 \times \cdots T_n \longrightarrow T$ in Σ_M we assign the function $g^{\mathcal{I}_M} : \mathcal{T}(CON_M^{T_1}) \times \cdots \times \mathcal{T}(CON_M^{T_n}) \longrightarrow \mathcal{T}(CON_M^{T})$ defined as

$$f^{\mathcal{I}_M}(w_1, \ldots, w_n) = Eval_M(f(w_1, \ldots, w_n)).$$

For instance, if M is the module we have been working with all along, then $\texttt{Fact}^{\mathcal{I}_M} = \{< \texttt{0}, \texttt{s(0)} >, \ldots, < \texttt{s}^4(\texttt{0}), \texttt{s}^{24}(\texttt{0}) >, \ldots\}$. Finally, each term $t \in \mathcal{T}(\Sigma_M)$ is mapped to the object $t^{\mathcal{I}_M} = Eval_M(t)$, and then the **truth** of a sentence ϕ under the interpretation $\mathcal{I}_M$ is defined as follows:

- If $\phi \equiv (s = t)$ then ϕ is true iff $s^{\mathcal{I}_M} = t^{\mathcal{I}_M}$.
- If $\phi \equiv \neg\psi$, then ϕ is true iff ψ is not true.
- If $\phi \equiv \phi_1 \wedge \phi_2$ then ϕ is true iff both ϕ_1 and ϕ_2 are true.
- $\phi \equiv \phi_1 \vee \phi_2$ then ϕ is true iff either ϕ_1 or ϕ_2 is true.
- If $\phi \equiv \phi_1 \Longrightarrow \phi_2$ then ϕ is true iff $\neg\phi_1 \vee \phi_2$ is true.
- If $\phi \equiv (\exists v \in T)\psi$ then ϕ is true iff there is an object $w \in T$ such that $\psi[v \mapsto w]$ is true.
- If $\phi \equiv (\forall v \in T)\psi$ then ϕ is true iff $\neg(\exists v \in T)\neg\psi$ is true.

If ϕ is true under $\mathcal{I}_M$ we say that the latter **satisfies** ϕ and write $\mathcal{I}_M \models \phi$; the contary is indicated by $\mathcal{I}_M \not\models \phi$. In our continuing example:

$$\mathcal{I}_M \models (\forall n \in \texttt{NatNum})\,[\,\texttt{Leq(Exp(s}^2\texttt{(0)}, n\texttt{),Exp(s}^3\texttt{(0)}, n\texttt{))} = \texttt{True}\,] \tag{B.6}$$

$$\begin{gathered}\mathcal{I}_M \models (\forall n \in \texttt{NatNum})(\forall l \in \texttt{NatList}) \\ [\,\texttt{Member}(n, l) = \texttt{True} \Longleftrightarrow \texttt{Member}(n, \texttt{Reverse}(l)) = \texttt{True}\,]\end{gathered} \tag{B.7}$$

$$\mathcal{I}_M \models (\forall t \in \texttt{BTree})\,[\,\texttt{root-value}(t) = \texttt{head}(\texttt{PreOrder}(t))\,]. \qquad \text{(B.8)}$$

Eq.(B.6) asserts that $2^n \leq 3^n$, eq.(B.7) says that l and $\texttt{Reverse}(l)$ contain exactly the same elements, and eq.(B.8) says that the number at the root of a binary tree is the first one to be listed in a pre-order traversal (notice that eq.(B.8) holds even for `null` trees owing to our conventions regarding selectors and minimal objects). By contrast, the two following sentences are **false** under the standard interpretation of FOT_M:

$$(\exists n \in \texttt{NatNum})(\forall m \in \texttt{NatNum})\,[\,\texttt{Less}(m,n) = \texttt{True}\,]$$

$$(\forall l \in \texttt{NatList})\,[\,\texttt{Length}(l) = \texttt{s}(\texttt{0})\,].$$

Observe that the axioms of FOT_M are always true under $\mathcal{I}_M$ (intuitively, that is because the axioms have been "built into" the interpreter $Eval_M$). Therefore, coupled with a sound deduction system, these axioms could be used to derive valid (true) statements about the objects in the universe of M and the various algorithms in Σ_M that operate on them.

Bibliography

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

[2] R.W. Floyd, Assigning meanings to programs, in *Mathematical Aspects of Computer Science*, Proceedings Symposia in Applied Mathematics **19**, (American Mathematical Society, Providence RI, 1967) 19–32.

[3] D. König, *Fundamenta Mathematicae* **8**, 1926, 114-134.

[4] Z. Manna, *Termination of Algorithms*, Ph.D Thesis, Computer Science Department, Carnegie–Mellon University, Pittsburgh, PA 1968.

[5] Z. Manna, *Mathematical Theory of Computation*, McGraw Hill, New York, 1974.

[6] D. McAllester and K. Arkoudas, Walther recursion, CADE 13, 1996.

[7] J S. Moore, A mechanical proof of the termination of Takeuchi's function, *Inf. Process. Let.* **9** (4), 1979, 176-181.

[8] C. Walther, On proving the termination of algorithms by machine, *Artificial Intelligence*, **71**, 1994, 101-157.