Mechanical verification of algorithm termination

Kostas Arkoudas

June 24, 1997

1 Introduction

An important problem in many verification logics is proving that a recursively defined algorithm halts for all appropriate inputs. Termination is important for demonstrating total correctness. For instance, proving that QuickSort produces a sorted permutation of any given list of numbers presupposes that the algorithm halts on all input lists. Termination is also important from an operational standpoint—infinite loops are generally undesirable. If we have a formal guarantee that every procedure written by the programmer halts on all inputs, then we can be assured that execution will never diverge. In this paper we present an efficient and fully automatic method for proving that a given algorithm halts on all inputs.

Of course in any Turing-complete formalism the problem of determining whether an arbitrary algorithm computes a total function is mechanically undecidable (more undecidable, in fact, than the halting problem[1]). Accordingly, the best we can hope for is a conservative approximation. The claimed advantage of the present method is that it is a good approximation, i.e., that it verifies the termination of a wide variety of algorithms that a programmer might wish to express in practice; and furthermore, that it does so efficiently, and without requiring any help from or interaction with the user.

There are other alternatives for ensuring termination. One is to constrain the syntax of the language in such a way that the programmer can only define total functions, in which case termination is guaranteed *a priori*. This is the approach taken in Martin-Löf's type theory [10], in which only primitive recursive definitions are possible. Gödel's T system [5] is another formalism in which only total functions are definable. From a theoretical

[1]In the arithmetic hierarchy the halting problem is in Σ_1, while the totality problem is 1-complete for Π_2.

viewpoint these formalisms are quite powerful, in the sense that any total function a programmer might want to compute "in the real world" will likely be definable in the system. It is known, for example, that any computable function which is provably total in first-order arithmetic is definable in Gödel's T system. This extensional power, however, derives from the use of high-order functionals and often has little bearing in practice.

Without resorting to functionals one is limited to first-order primitive recursive definitions (in which a recursive call is of the form `f(sel(x))` for some *selector* `sel` that decreases the size of the input `x` by exactly one), which are useful, of course, but not quite adequate. Many well-known algorithms—such as Euclid's algorithm and most sorting algorithms—are defined in a non-primitive-recursive way. Such algorithms are often defined by some form of course-of-values recursion in which the size of an argument to a recursive call may decrease by an arbitrary quantity, not just by one. In MergeSort, for instance, the algorithm is recursively called on each *half* of the input list; and in Euclid's algorithm, too, the arguments to the successive recursive calls decrease by large leaps—the main reason why it is such an excellent method for computing greatest common divisors.

Such algorithms would be rejected by a system that only accepted primitive recursive definitions, even though, extensionally speaking, the functions computed by these algorithms would be definable by *some* primitive recursive definition, albeit a convoluted and inefficient one. The method presented in this paper does not increase the extensional power of these systems, but it does recognize algorithms such as the above that use course-of-values recursion.

Another alternative is to retain a Turing-complete language but require the users to *prove* to the system that their algorithms always halt. This is the approach taken in the Boyer-Moore theorem prover [1], in which the user must "help" the system understand why an algorithm terminates by formulating certain inductive lemmata that provide relevant hints. The main advantage of this method is its broad applicability, the fact that it can be used on a large class of algorithms. 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—an expectation that naive users often cannot meet.

Our method stems from Walther's work on the subject [13], and is based on the idea of *argument-bounded* functions. A function `g` is argument-bounded (a.b. for short) if it does not increase the size of its argument;

i.e., if the size of g(x) is $\leq$ the size of x, for any x.[2] An example is the tail function on lists. Oftentimes the argument to a recursive call of an algorithm f has the form f(g(x)), where g is an a.b. function such as tail and x is the input of f. Under certain conditions an a.b. function g actually *reduces* the size of its argument, i.e., we have $SZ(\mathtt{g(x)}) < SZ(\mathtt{x})$. The basic idea is to verify that these conditions obtain whenever a recursive call f(g(x)) is made. This, of course, will entail that every time a recursive call is made the size of the argument strictly decreases; and since size cannot go on decreasing indefinitely, the algorithm must terminate.

When g is a selector such as pred or tail then the conditions under which $SZ(\mathtt{g(x)}) < SZ(\mathtt{x})$ are simple: x must have positive size. In that case the size of g(x) is exactly one less than the size of x. Thus, to verify the termination of f, we must show that in the context of every recursive call f(g(x)), the argument x has positive size. This is usually easy to do, because recursive calls f(g(x)) almost always occur under explicitly stated assumptions such as $\mathtt{x} \neq 0$, or $\mathtt{x} \neq$ empty, etc. This is why the termination of primitive recursive definitions is so easy to verify.

Selectors, of course, are known to be a.b. by definition. We do not have to infer that. The challenge in moving from primitive recursion to course-of-values recursion is to mechanically identify a.b. functions g *other than selectors*; infer conditions under which such functions g actually reduce the size of their arguments; and verify that these conditions hold whenever a recursive call f(g(x)) is made.

Consider, for example, the recursive call Gcd(Minus(n,m),m) made in the body of Euclid's algorithm for computing Gcd(n,m). The key here is to somehow know that Minus(n,m) never increases the size of n (i.e., that Minus is "1-bounded"), and further, that it actually *decreases* n whenever both n and m are non-zero. We then simply have to verify that every time the recursive call Gcd(Minus(n,m),m) is made, both n and m are non-zero—a fact which is almost handed to us on a platter by the algorithm's definition. Or consider the algorithm MinSort(l) which returns empty if l is empty and add(Min(l),MinSort(RemoveMin(l))) otherwise, where RemoveMin(l) is a previously defined algorithm that returns a copy of l with its smallest element removed.[3] Again the key here is to know that RemoveMin is a.b., and that it reduces the size of its input list whenever the latter is non-empty. Then all we have to do is verify that l is non-empty in the context of the recursive call MinSort(RemoveMin(l)), which again is transparent from the

[2] For the purposes of this introduction we focus on unary functions.

[3] Assume for simplicity that all the elements of l are distinct.

algorithm's definition.

Walther [13] has a sophisticated system for (a) inferring that a function `g` is a.b., (b) deriving a disjunction of conditions that is sufficient *and necessary* for `g` to reduce its argument, and (c) verifying that one of the disjuncts is true whenever a recursive call `f(g(x))` is made. At the heart of this arrangement is a calculus for inferring size inequalities. We modify that calculus in a way that expedites inference without sacrificing expressiveness. We then drastically simplify (b) by considering only one type of condition: whether or not the input `x` has positive size. I.e., we only try to prove one specific lemma: that $SZ(\texttt{x}) > 0$ is sufficient for $SZ(\texttt{g(x)}) < SZ(\texttt{x})$. As a result, step (c) is greatly simplified as well: we only need to check that in the context of a recursive call `f(g(x))`, we have $SZ(\texttt{x}) > 0$, so we can "fire" our lemma. As we have already indicated, inferring conditions of the form $SZ(\texttt{x}) > 0$ is usually very easy.

In contrast, because the conditions inferred in (b) by Walther's method are generally much more complicated than our simple "$SZ(\texttt{x}) > 0$" triggers, step (c) becomes much harder in his system. In particular, he assumes that a general-purpose induction theorem prover is available which can be invoked at step (c) to verify the disjunction generated in (b). On the one hand this level of sophistication enables his system to recognize some algorithms which our method does not. On the other hand, even if we assume one is available, calling up a general-purpose theorem prover to inductively verify hypotheses of essentially arbitrary complexity can be quite expensive. By contrast, our method does not require any special theorem-proving capabilities and has a provably low worst-case complexity.[4] Moreover, empirical tests on a variety of benchmarks have shown the success rate of our method to be nearly identical to that of Walther's.

The remainder of this paper is organized as follows. In the next section we describe IFL, a simple purely functional language that we will use to illustrate our method. In Sec. 3 we define the central concepts on which our method is based, and in the following section we offer a preview without getting into much detail. Sec. 5 explains the main parts of the method in detail, while the next section discusses some examples. Sec. 7 illustrates some typical scenarios in which the method would not work, Sec. 8 examines the possibility of future extensions, and finally Sec. **??** reviews previous similar work.

[4] And in fact on average the method has been found to be extremely fast. The termination of 35 functions taken from [13], [1], and elsewhere, was verified by the existing implementation of our method (in C++) in slightly less than a minute.

2 The language

2.1 Syntax

Datatypes

The language offers concrete data types very similar to those of ML. The main difference is that polymorphism is not supported. Another difference is that the user must explicitly specify selectors (destructors) for each constructor, as pattern matching is not available.[5] In particular, an IFL datatype is defined as follows:

$$\texttt{datatype}\ T = \mathit{ConDec}\ \ \{\text{“}\texttt{|}\text{”}\ \mathit{ConDec}\}\texttt{;}^{6}$$

where each constructor declaration *ConDec* is either a plain identifier (representing a constructor of no arguments), or an expression of the form

$$con(sel_1 : type_1, \ldots, sel_n : type_n)$$

where *con*, sel_i, and $type_i$ are identifiers; sel_i will be the selector associated with the i^{th} argument position of *con*. Some well-known data types are defined below in this notation.

```
datatype Boolean = true | false;

datatype NatNum = zero | succ(pred:NatNum);

datatype NatList = empty | add(head:NatNum,tail:NatList);

datatype BTree = null |
                 node(value:NatNum,left:BTree,right:BTree);

datatype LamTerm = var(index:NatNum) |
                   abs(formal:NatNum,body:LamTerm) |
                   app(rator:LamTerm,rand:LamTerm);
```

A constructor $c(sel_1 : T_1, \ldots, sel_n : T_n)$ of a datatype T is viewed as a function $c : T_1 \times \cdots \times T_n \to T$ that produces objects of type T. E.g. the signatures of `zero` and `succ` are `zero`$: \to$ `NatNum` and `succ`$:$`NatNum` $\to$ `NatNum`. We call T_i the **ith argument type** of c. A constructor of zero arguments is called a **constant**. We define the **objects of type** T as follows:

1. A constant of T is an object of type T.

[5] The method itself does not essentially depend on these restrictions and could be adapted to a language with pattern-matching and even polymorphism; see Sec. 8.

[6] The notation $\{A\}$ means zero or more occurences of A.

2. If $c : T_1 \times \cdots \times T_n \rightarrow T$ is a constructor of T and $w_1, \ldots, w_n$ are objects of types $T_1, \ldots, T_n$, respectively, then $c(w_1, \ldots, w_n)$ is an object of type T.

Thus, an object is simply a Herbrand term that is built exclusively from constructors. E.g. the objects of type `NatNum` are `zero`, `succ(zero)`, `succ(succ(zero))`, and so on.

We write sel_i^c to denote the selector associated with the i^{th} argument position of constructor c; e.g., $sel_2^{\texttt{add}} = \texttt{tail}$. Selectors are the inverses of constructors. A constructor c of arity $n > 0$ "binds" or "pulls together" n objects $w_1, \ldots, w_n$ to produce a new complex object w, which, in some sense, contains the w_i objects. By contrast, when sel_i^c is applied to w it retrieves the component object w_i—it *projects* the object w along the i^{th} co-ordinate. In symbols, we have $sel_i^c(c(w_1, \ldots, w_n)) = w_i$.

If a constructor c of T takes an argument of type T then it is called **reflexive**. Examples are `succ`, `add`, and `app`. Constructors that are not reflexive are called **irreflexive**. Examples are `zero`, `empty`, and `var`. Note that a datatype T must have at least one irreflexive constructor so that production of T-objects can get started, so to speak. In a similar vein, a selector is called reflexive if its type is the same as its argument type. Thus, `pred` and `tail` are reflexive selectors, while `head` and `formal` are irreflexive. We will write $RCON_T$ and $IRCON_T$ for the reflexive and irreflexive constructors of T, respectively, and CON_T for $RCON_T \cup IRCON_T$. The set of all selectors of all constructors of T will be written as SEL_T.

Our stipulations so far do not specify what happens if we apply a selector of a constructor $c : T_1 \times \cdots \times T_n \rightarrow T$, 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. E.g. what is "the value" of `pred(zero)` or of `formal(var(zero))`? This is a problem because we wish to regard sel_i^c as a *total* function from T to T_i; that is, sel_i^c should return some object of type T_i or another regardless of what object of type T it is given as input. We adopt the convention that when sel_i^c is applied to an object of type T whose top symbol is not c, the result is the **minimal** object of type T_i. In general, the minimal object of a datatype T is defined to be either (i) the leftmost constant of T, if one exists, or else (ii) the object obtained by applying the leftmost irreflexive constructor of T (one must exist) to the minimal objects of its argument types. For instance, the minimal objects of `NatList` and `LamTerm` are `empty` and `var(zero)`, respectively.

Functions

A function definition has the form

$\texttt{function}\ \ \textit{fun-name}(\textit{par}_1 : T_1, \ldots, \textit{par}_n : T_n) : T = \textit{fun-body};$

where *fun-body* is either (i) a single term r of type T (we will shortly specify what we mean by "a term of type T"), or else it is (ii) an if-then-else expression of the form

$$\texttt{if}\ E_1\ \texttt{then}\ r_1\ \texttt{else}\ \cdots\ \texttt{if}\ E_m\ \texttt{then}\ r_m\ \texttt{else}\ r_{m+1}$$

for some $m \geq 1$. The expression $[(E_1, r_1), \ldots, (E_m, r_m), (\texttt{true}, r_{m+1})]$, for $m \geq 0$, will be used as a shorthand for the body of an arbitrary function (case (i) would correspond to $m = 0$ and case (ii) to $m > 0$). Each r_i must be a term of type T, while each boolean expression E_i is either an equality of the form $r = s$ for two terms r and s of the same type, or else it is either a negation or a conjunction or a disjunction built from other boolean expressions. Two simple examples illustrating this syntax are included below.

```
function Minus(n,m:NatNum):NatNum =
if m = zero then n
else
  Minus(pred(n),pred(m));

function Member(n:NatNum,l:NatList):Boolean =
if l = empty then false
else
  if head(l) = n then true
  else
    Member(n,tail(l));
```

Note that, owing to the `else` clauses, each condition E_i in the body of a function is (operationally) equivalent to $E_i \wedge \neg E_{i-1} \wedge \cdots \wedge \neg E_1$. Thus our choice of syntax automatically ensures that every function is *deterministic* and *case-complete*, so that for any sequence of appropriate inputs exactly one condition in the body of the function will be true.

Scripts

A **script** $\mathcal{S}$ is defined to be a finite *sequence* of definitions of data types and functions. Scripts are sequences because the idea is that they are developed incrementally, in stages: we usually start from scratch, with the empty script $\mathcal{S}_0$; then we obtain a script $\mathcal{S}_1$ by defining the first data type T_1; then we extend $\mathcal{S}_1$ with a new definition, either of a second datatype T_2 or of our first function f_1, thereby otaining a new script $\mathcal{S}_2$; then on with a new definition, and so forth. Thus, a script is like a Scheme or ML environment built through the Read-Eval-Print loop. If $\mathcal{S}$ is a non-empty script comprising n datatypes $T_1, \ldots, T_n$ and m functions $f_1, \ldots, f_m$, we will write

- $CON_{\mathcal{S}}$ for $CON_{T_1} \cup \cdots \cup CON_{T_n}$ (and likewise for $RCON_{\mathcal{S}}$ and $IRCON_{\mathcal{S}}$),
- $SEL_{\mathcal{S}}$ for $SEL_{T_1} \cup \cdots \cup SEL_{T_n}$,
- $FUN_{\mathcal{S}}$ for $\{f_1, \ldots, f_m\}$,
- $\Sigma_{\mathcal{S}}$ for $CON_{\mathcal{S}} \cup SEL_{\mathcal{S}} \cup FUN_{\mathcal{S}}$ (this is the **signature** of $\mathcal{S}$), and
- $TYPES_{\mathcal{S}}$ for $\{T_1, \ldots, T_n\}$.

Now suppose we are given a mapping $\mathcal{V}$ that assigns to each type T in $TYPES_{\mathcal{S}}$ a (possibly empty) set $\mathcal{V}(T)$ of "variables of type T" which can be used to denote arbitrary objects of type T. We then define a **term of type** T, for any $T \in TYPES_{\mathcal{S}}$, to be either a variable in $\mathcal{V}(T)$ or an expression of the form $g(t_1, \ldots, t_k)$, where $g : T_1 \times \cdots \times T_k \rightarrow T$ is a member of $\Sigma_{\mathcal{S}}$ and $t_1, \ldots, t_k$ are terms of types $T_1, \ldots, T_k$, respectively. For example, if $\mathcal{S}$ contains `NatNum`, `NatList`, `Minus` and `Member`, then `Minus(n,succ(zero))` and `Member(zero,add(succ(zero),empty))` are terms of types `NatNum` and `NatList`, respectively (assuming that `n` is a variable of type `NatNum`). A **ground term** is one that does not contain any variables. From the two foregoing examples only the second term is ground.

We will write $\mathcal{T}_T(\Sigma_{\mathcal{S}}, \mathcal{V})$ for the set of all terms of type T, and $\mathcal{T}_T(\Sigma_{\mathcal{S}})$ for the set of all ground terms of type T. Furthermore, we will write $\mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$ for the set of all terms of type T, for any $T \in TYPES_{\mathcal{S}}$; so, assuming that $TYPES_{\mathcal{S}} = \{T_1, \ldots, T_n\}, \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V}) = \mathcal{T}_{T_1}(\Sigma_{\mathcal{S}}, \mathcal{V}) \cup \cdots \cup \mathcal{T}_{T_n}(\Sigma_{\mathcal{S}}, \mathcal{V})$. Likewise, the expression $\mathcal{T}(\Sigma_{\mathcal{S}})$ will denote the set of all ground terms of any type. If we let $\mathcal{V}_{\emptyset}$ stand for the mapping that assigns to each type the empty set of variables, then $\mathcal{T}(\Sigma_{\mathcal{S}}) = \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V}_{\emptyset})$.

Next, we define the syntax of **boolean expressions** over $\Sigma_{\mathcal{S}}$ and $\mathcal{V}$ as follows:

$$E ::= \texttt{true} \mid \texttt{false} \mid (s = t) \mid \neg E \mid (E \wedge E) \mid (E \vee E)$$

for any $s, t \in \mathcal{T}_T(\Sigma_{\mathcal{S}}, \mathcal{V}), T \in TYPES_{\mathcal{S}}$. Expressions of the form $\neg(s = t)$ are written as $(s \neq t)$. Finally, $\mathcal{B}(\Sigma_{\mathcal{S}}, \mathcal{V})$ will designate the set of all boolean expressions over $\Sigma_{\mathcal{S}}$ and $\mathcal{V}$.

When a script $\mathcal{S}$ is extended with a new datatype or function definition, we must check that certain syntactic conditions are satisfied. For a definition of a new datatype T we need to make sure that all the constructor and selector names are fresh (do not appear in any previous definitions), that there is at least one irreflexive constructor, and that each argument type of every constructor is either a previously defined datatype or T itself. For a definition of a new function $f : T_1 \times \cdots \times T_n \rightarrow T$ with parameters $x_1 : T_1, \ldots, x_n : T_n$ and body $[(E_1, r_1), \ldots, (E_m, r_m), (\texttt{true}, r_{m+1})]$, we must ensure that (i) all parameters are distinct, (ii) $T_1, \ldots, T_n, T$ are all in $\mathit{TYPES}_{\mathcal{S}}$, and (iii) for every $j = 1, \ldots, m+1$ we have $r_j \in \mathcal{T}_T(\Sigma_{\mathcal{S}} \cup \{f\}, \mathcal{V}_f)$ and $E_j \in \mathcal{B}(\Sigma_{\mathcal{S}} \cup \{f\}, \mathcal{V}_f)$, where $\mathcal{V}_f$ maps each T_i, $i \in \{1, \ldots, n\}$, to the set of those parameters amongst $x_1, \ldots, x_n$ that are of type T_i, and every other type to $\emptyset$.

A definition that satisfies these constraints is called *well-formed*. A script is called **well-formed** if it is obtainable from the empty script through a series of well-formed definitions. Finally, a script is **admissible** if (a) it is well-formed, and (b) every user-defined function terminates for all apropriate inputs. Checking (a) is easy; (b) is the undecidable task, and is the subject of this paper.

2.2 Semantics

Let a well-formed script $\mathcal{S}$ be given. The interpreter $Eval_{\mathcal{S}}$ shown in Fig. 1 maps an arbitrary ground term in $\mathcal{T}(\Sigma_{\mathcal{S}})$ to an object in $\mathcal{T}(\mathit{CON}_{\mathcal{S}})$. The interpreter can be seen as an implementation of the operational semantics of IFL: the objects of $\mathcal{S}$ are considered to be in normal form, whereas a ground term t that contains function symbols from $\mathit{FUN}_{\mathcal{S}}$ specifies a computation and needs to be reduced to normal form. $Eval_{\mathcal{S}}$ performs this reduction by rewriting t in accordance with the definitions of the various functions in $\mathit{FUN}_{\mathcal{S}}$ and the conventions regarding constructors and selectors.

For any $t \in \mathcal{T}(\Sigma_{\mathcal{S}})$, we write $Eval_{\mathcal{S}}(\mathrm{t})\!\downarrow$ to indicate that $Eval_{\mathcal{S}}$ halts when started with t as input. We can now formally define a user-defined function $f : T_1 \times \cdots T_n \rightarrow T$ to be **terminating** iff $Eval_{\mathcal{S}}(f(w_1, \ldots, w_n))\!\downarrow$ for all objects $w_1 \in T_1, \ldots, w_n \in T_n$. As a convention, when $\mathcal{S}$ is implicitly understood or immaterial, we will simply write *Eval* instead of $Eval_{\mathcal{S}}$.

[6] The expression $E[w_1/x_1, \ldots, w_n/x_n]$ is obtained from E by consistently replacing each occurence of x_i by w_i. For brevity, we will occasionally write this as $E[w_i/x_i]$. Note that the order in which the substitutions are performed is immaterial since the w_i are ground.

```
;; For a constant con ∈ CON_S :
     Eval_S con = con |
;; For a non-constant con ∈ CON_S :
     Eval_S con(t_1,...,t_n) = con(Eval_S(t_1),...,Eval_S(t_n))  |
;; For a selector sel_i^c ∈ SEL_S, where Minimal(c,i) returns the
;; minimal object of the i^th argument type of c:
     Eval_S sel_i^c(t) = case  Eval_S(t)  of
                          c(t_1,...,t_n) ==>  t_i |
                          _ ==>  Minimal(c,i) |
;; For a user-defined function f ∈ FUN_S  with n parameters x_1,...,x_n
;; and body [(E_1,r_1),...,(E_m,r_m),(true,r_{m+1})]:
     Eval_S f(t_1,...,t_n) =
       let val (w_1,...,w_n) = (Eval_S(t_1),...,Eval_S(t_n))
           fun do_if(E::tl_bexps,r::tl_terms) =
               if TEval_S(E[w_1/x_1,...,w_n/x_n])^7 then
                  Eval_S(r[w_1/x_1,...,w_n/x_n])
               else
                  do_if(tl_bexps,tl_terms)  in
           do_if([E_1,...,E_m,true],[r_1,...,r_{m+1}])  end
     and
     TEval_S (s = t) = (Eval_S(s) = Eval_S(t)) |
     TEval_S (¬E) = not(TEval_S(E)) |
     TEval_S (E_1 ∧ E_2) = TEval_S(E_1) andalso TEval_S(E_2) |
     TEval_S (E_1 ∨ E_2) = TEval_S(E_1) orelse TEval_S(E_2);
```

Figure 1: An interpreter for IFL, written in ML-like notation.

3 Theoretical underpinnings

3.1 Object size

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 $SZ_{\texttt{NatList}}(\texttt{empty}) = 0$, $SZ_{\texttt{NatList}}(\texttt{add}(\texttt{zero}, \texttt{add}(\texttt{succ}(\texttt{zero}), \texttt{empty}))) = 2$, etc. More formally, we have

$$SZ_{\texttt{NatList}}(l) = \begin{cases} 0 & \text{if } l \equiv \texttt{empty} \\ 1 + SZ_{\texttt{NatList}}(\texttt{tail(l)}) & \text{if } l \equiv \texttt{add}(\cdots). \end{cases} \tag{1}$$

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:

$$SZ_{\texttt{BTree}}(t) = \begin{cases} 0 & \text{if } t \equiv \texttt{null} \\ 1 + SZ_{\texttt{BTree}}(\texttt{left(t)}) + & \\ \qquad SZ_{\texttt{BTree}}(\texttt{right(t)}) & \text{if } t \equiv \texttt{node}(\cdots). \end{cases} \tag{2}$$

Each of the two size measures given above applies only to objects of its own particular type, but nevertheless they both use the same essential idea: if an object is built from an *irreflexive* constructor, let its size be zero (the first clauses of definitions 1—2); if it is built from a *reflexive* constructor, let its size be *one more* than the sum of the sizes of its components of the same type (tail clauses of 1—2). This simply reflects the intuition that the irreflexive constructors of a type T produce "simple" 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 `BTree`s `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:

$$SZ_T(s) = \begin{cases} 0 & \text{if } s \not\equiv rcons(\cdots) \\ 1 + SZ_T(t_{i_1}) + \cdots + SZ_T(t_{i_m}) & \text{if } s \equiv rcons(t_1, \ldots, t_n) \end{cases} \tag{3}$$

for any $rcons \in RCON_{\mathcal{S}}$ with reflexive argument positions $i_1, \ldots, i_m$. If the type at hand is implicitly understood or can be easily deduced from the context, we may drop the subscript and simply write $SZ(s)$.

We distinguish between $SZ(s)$ and the size of s as a Herbrand term, which we will denote by $|s|$. Viewing terms as trees, $|s|$ is simply the number of nodes of s. E.g., for $s = \texttt{add(succ(zero), empty)}$ we have $|s| = 4$, whereas $SZ(s) = 1$. In addition, $|s|$ makes sense for any $s \in \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$, including non-ground terms, while SZ is only defined for objects.

3.2 A size ordering and the GTZ predicate

Now let $s, t \in \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$ be two terms of the same type T in some *admissible* script $\mathcal{S}$, let $E_1, \ldots, E_n \in \mathcal{B}(\Sigma_{\mathcal{S}}, \mathcal{V})$, and let $v_1, \ldots, v_k$ be all and only the variables that occur in s or in t or in one of the E_j, where each v_i is of some type T_i. We will write $\{E_1, \ldots, E_n\} \models s \preceq t$ to indicate that for all objects $w_1 : T_1, \ldots, w_k : T_k$ for which $TEval_{\mathcal{S}}(E_1[w_i/v_i] \wedge \cdots \wedge E_n[w_i/v_i]) = \texttt{true}$, we have

$$SZ_T(Eval_{\mathcal{S}}(s[w_i/v_i])) \leq SZ_T(Eval_{\mathcal{S}}(t[w_i/v_i])). \tag{4}$$

When there are no variables to be replaced (i.e. when s, t, and the E_i are ground), $\{E_1, \ldots, E_n\} \models s \preceq t$ simply means that if $TEval(E_1 \wedge \cdots \wedge E_n) = \texttt{true}$ then $SZ(Eval(s)) \leq SZ(Eval(t))$. When there are no boolean expressions we simply write $s \preceq t$ instead of $\emptyset \models s \preceq t$. Thus the inequality $s \preceq t$ holds iff every ground instance of it holds. For example, we have $\texttt{Minus(n,m)} \preceq \texttt{n}$ because no matter what objects we substitute for `n` and `m`, the size of `Minus(n,m)` will always be less than or equal to the size of `n`.

We use similar notation for strict inequalities: $\{E_1, \ldots, E_n\} \models s \prec t$ iff (4) holds with $<$ in place of $\leq$, and $s \prec t$ for $\emptyset \models s \prec t$; e.g. $\texttt{l} \prec \texttt{add(n,l)}$. Finally, we write $\{E_1, \ldots, E_n\} \models GTZ(s)$ to mean that for all objects $w_1 \in T_1, \ldots, w_k \in T_k$, if $TEval(E_1[w_i/v_i] \wedge \cdots \wedge E_n[w_i/v_i]) = \texttt{true}$ then $SZ(Eval(s[w_i/v_i])) > 0$. We will write $GTZ(s)$ for $\emptyset \models GTZ(s)$. Thus $GTZ(s)$ iff the result of evaluating any ground instance of s has positive size. For instance, we have $GTZ(\texttt{succ(n)})$, but $\neg GTZ(\texttt{Minus(n,m)})$. For ground s, of course, we have $GTZ(s)$ iff $SZ(Eval(s)) > 0$; e.g. we have $GTZ(\texttt{succ(zero)})$ since $SZ(Eval(\texttt{succ(zero)})) = 1 > 0$.

Note that if the top function symbol of s is a constructor then deciding $GTZ(s)$ is trivial—simply check whether the constructor is reflexive. For we always have $GTZ(rc(\cdots))$ and $\neg GTZ(irc(\cdots))$ for every reflexive (irreflexive) constructor rc (irc). However, in general all three of the relations we have defined in this section ($s \preceq t, s \prec t$, and $GTZ(s)$) are undecidable because the terms s and t can involve variables, which may range over infinitely many objects. Consequently, in the absence of a fortuitous counter-example any complete procedure for deciding any of these conditions would have to perform infinitely many numerical comparisons, which is not possible in a finite amount of time. But if we restrict these problems to ground terms only, then they become decidable in virtue of the strong normalization guaranteed by the admissibility of $\mathcal{S}$; we could then simply evaluate all the relevant ground terms and decide accordingly.

3.3 Argument-bounded functions

A key role in our method is played by *argument-bounded functions.* A function $f : T_1 \times \cdots \times T_n \longrightarrow T$ is i-**bounded** for some $i \in \{1, \ldots, n\}$ if $T_i = T$ and $f(v_1, \ldots, v_n) \preceq v_i$ (for some variables $v_1 : T_1, \ldots, v_n : T_n$). Likewise, f is **strictly** i-bounded if the foregoing condition holds with $\prec$ in place of $\preceq$. We call f (strictly) argument-bounded (abbreviated "a.b.") if it is (strictly) i-bounded in at least one i.[8]

3.4 The termination theorem

In this paper we will only be concerned with termination of **normal** algorithms. In IFL, a function f is called normal if every conditional (E_j, r_j) in its body is such that (i) E_j contains no recursive calls $f(\cdots)$, and (ii) r_j contains no *nested* recursive calls $f(\cdots f(\cdots) \cdots)$.
This is not a severe restriction since most function definitions encountered in practice meet these requirements. Non-normal functions such as McCarthy's 91–function [8] or Takeuchi's function for reversing a list [11], although mathematically interesting, tend to be of little practical relevance.

Following an idea introduced by Floyd [4], most methods for proving termination, including ours, are based on the concept of well-founded sets. Formally, a well-founded set is an ordered pair (W, R), where W is a 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 seqences $x_1, x_2, x_3, \ldots$ such that $x_1\ R\ x_2,\ x_2\ R\ x_3,\ x_3\ R\ x_4, \ldots$. If we read $x\ R\ y$ as "x is larger than y", in some abstract sense of "large", then to say that R is well-founded is to say that 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 3.1 (Main termination theorem) *Let $\mathcal{S}$ be an admissible script and let $\mathcal{S}'$ be an extension of $\mathcal{S}$ obtained by defining a new normal function*

[8] McAllester and Arkoudas [9] use the (perhaps more suggestive) terms "conserver" and "reducer" for what we call an a.b. and strictly a.b. function, respectively.

$f : T_1 \times \cdots \times T_n \to T$ with n parameters $x_1 : T_1, \ldots, x_n : T_n$ and body $[(E_1, r_1), \ldots, (E_m, r_m), (\texttt{true}, r_{m+1})]$. Then f is terminating in $\mathcal{S}'$ iff there is a well-founded set (W, R) and a function $Q : T_1 \times \cdots \times T_n \to W$ such that for each recursive call $f(t_1, \ldots, t_n)$ in each $r_j, j = 1, \ldots, m+1$, the following holds for any n objects $w_1 \in T_1, \ldots, w_n \in T_n$: if $TEval_{\mathcal{S}}(E_j[w_i/x_i]) = \texttt{true}$, then $Q(w_1, \ldots, w_n) \; R \; Q(Eval_{\mathcal{S}}(t_1[w_i/x_i]), \ldots, Eval_{\mathcal{S}}(t_n[w_i/x_i]))$.

A function Q that meets the above requirements is called a **termination function for** f. Note that the assumptions that f be normal and $\mathcal{S}$ be admissible are crucial, otherwise the defining property of Q as stated in the theorem would be nonsensical. As regards normalcy, if some t_p contained an additional recursive call $f(t'_1, \ldots, t'_n)$ then the expression $Eval_{\mathcal{S}}(t_p[w_i/x_i])$ would be undefined because $f \notin \Sigma_{\mathcal{S}}$ and thus we cannot use $Eval_{\mathcal{S}}$ to compute $t_p[w_i/x_i]$; and we would have the same problem if an E_j contained a recursive call to f: the value $TEval_{\mathcal{S}}(E_j[w_i/x_i])$ would then be ill-defined for the same reasons. But to be sure that these values are well-defined we also need to know that $Eval_{\mathcal{S}}$ and $TEval_{\mathcal{S}}$ will halt, and this is the guarantee that the admissibility of $\mathcal{S}$ affords us.

The termination function Q should be seen as mapping each n–tuple of objects $w_1 \in T_1, \ldots, w_n \in T_n$ to some "quantity" in W. In our method (W, R) will be $(N, >)$, so the values of Q will be numeric. Furthermore, Q will be based on the size function *SZ*. In particular, we will have

$$\mathcal{Q}(w_1, \ldots, w_n) = \sum_{i \,\in\, \mathrm{MS}_f} SZ(w_i) \tag{5}$$

where MS_f is the so-called **measured subset** of f (in the terminology of Boyer and Moore [1]). This is the set of all argument positions $i \in \{1, \ldots, n\}$ such that for every recursive call $f(t_1, \ldots, t_n)$ in each r_j, $j = 1, \ldots, m+1$, we have $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models t_i \preceq x_i$.

The main idea, as explained in the next section, is to verify that each time a recursive call is made, the size of at least one argument in MS_f decreases. It is important to insist that size decrease must occur within the measured subset; merely requiring that the size of *some* argument should decrease with each recursion is not sufficient (let `f(x,y:NatNum) = f(y,x)` and consider any call `f(a,b)` with `a` $\neq$ `b`; every time a recursive call is made the size of at least one argument decreases, yet evaluation clearly diverges). The measured subset provides us with a top fringe of sorts over certain argument positions that can only move lower.

4 Method outline

Our method, in a nutshell, will be to attempt to prove that the function $\mathcal{Q}$, as defined by (5), is indeed a termination function for f. Thus from the outset there are two potential sources of failure, even when we only consider normal functions:

1. $\mathcal{Q}$ may not in fact be a termination function for f.

2. Our method may fail *to prove* that $\mathcal{Q}$ is a termination function for f, even if it is. This is to be expected in view of the problem's undecidability: there is no algorithm which will output "Yes" if *and only if* $\mathcal{Q}$ is a termination function for f.

Nevertheless, it turns out—as a matter of empirical fact—that $\mathcal{Q}$ is indeed a termination function for various commonly encountered f, so the first problem is not of dismal proportions. As regards the inevitable incompleteness, we again have to resort to an empirical argument: our method often turns out to be "complete enough" for practical purposes.

Of course if we had an oracle we could easily devise an algorithm for determining whether or not $\mathcal{Q}$ is is a termination function for f:

1. Compute MS_f: starting with $\mathrm{MS}_f = \emptyset$, repeat for each $i = 1, \ldots, n$:
2. Check whether $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models t_i \preceq x_i$ for every
3. recursive call $f(t_1, \ldots, t_n)$ in each $r_j, j = 1, \ldots, m+1$. If yes,
4. set $\mathrm{MS}_f \leftarrow \mathrm{MS}_f \cup \{i\}$.
5. If $\mathrm{MS}_f = \emptyset$, halt with answer = "No".
6. For every recursive call $f(t_1, \ldots, t_n)$ in each $r_j, j = 1, \ldots, m+1$, do:
7. If there is no $i \in \mathrm{MS}_f$ such that $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models t_i \prec x_i$,
8. halt with answer = "No".
9. Output answer = "Yes".

It is not difficult to see that the above "algorithm" is both sound and complete. The oracle, of course, would be needed at lines 2 and 7, for deciding the two conditions $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models t_i \preceq x_i$ and $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models t_i \prec x_i$, which are undecidable, as we have already discussed.

In the absence of oracles, we approximate. First, instead of dealing with an arbitrary boolean expression E_i in its full generality, we filter from it certain simple syntactic facts called *size atoms*, which we can put to good use, and ignore everything else. A size atom is either an equality of the form $t = rc(\cdots)$, for a reflexive constructor rc, or an inequality $t \neq irc(sel_1^{irc}(t), \ldots, sel_m^{irc}(t))$, for an irreflexive constructor irc of m arguments.

Size atoms can help us prove $GTZ(\cdots)$ assertions. E.g., from the atoms `n = succ(m)` and `l` $\neq$ `empty` we can immediately infer $GTZ($`n`$)$ and $GTZ($`l`$)$, respectively. The following is a simple algorithm that takes an $E \in \mathcal{B}(\Sigma_{\mathcal{S}}, \mathcal{V})$ and extracts from it a set $\widehat{E}$ of size atoms that are logically implied by E (to facilitate exposition, we assume that negation signs in E appear in front of equalities $s = t$ only)[9]:

$$\begin{aligned}
\widehat{(s = rc(t_1, \ldots, t_n))} &= \{(s = rc(t_1, \ldots, t_n))\} \\
\widehat{(t \neq irc(sel_1^{irc}(t), \ldots, sel_m^{irc}(t)))} &= \{(t \neq irc(sel_1^{irc}(t), \ldots, sel_m^{irc}(t)))\} \\
\widehat{(E_1 \wedge E_2)} &= \widehat{E_1} \cup \widehat{E_2} \\
\widehat{(E_1 \vee E_2)} &= \widehat{E_1} \cap \widehat{E_2}
\end{aligned}$$

while $\widehat{E} = \emptyset$ for every E that does not match any of the four above patterns. Clearly, $E \models \widehat{E}$. Further, for any $s, t \in \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$ and any set of size atoms $\{A_1, \ldots, A_k\}$ over $\Sigma_{\mathcal{S}}$ and $\mathcal{V}$, we define a relation $s \mathrel{\widehat{\preceq}} t \equiv \{A_1, \ldots, A_k\}$ (to be read "s is less than or equal to t modulo the size atoms $A_1, \ldots, A_k$") in a way that soundly approximates $\{A_1, \ldots, A_k\} \models s \preceq t$. To wit,

$$s \mathrel{\widehat{\preceq}} t \equiv \{A_1, \ldots, A_k\} \text{ implies } \{A_1, \ldots, A_k\} \models s \preceq t.$$

In particular, when $\{A_1, \ldots, A_k\} = \emptyset$ we have $s \mathrel{\widehat{\preceq}} t \Rightarrow s \preceq t$. The advantage of the $\widehat{\preceq}, \equiv$ relation, of course, is that it is decidable—and easily so. Given arbitrary s, t, and $A_1, \ldots, A_k$, we can determine whether or not $s \mathrel{\widehat{\preceq}} t \equiv \{A_1, \ldots, A_k\}$ in $O(|s|)$ time. We then develop a similar approximation for the strict inequality problem, and finally we can do away with oracles: we simply use the earlier "algorithm" with $s \mathrel{\widehat{\preceq}} t \equiv \widehat{E_j} \cup \widehat{\neg E_{j-1}} \cup \cdots \cup \widehat{\neg E_1}$ in place of $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models t_i \preceq x_i$, and likewise for strict inequalitites.

5 The method

5.1 Proving termination

Recall that scripts are developed incrementally: at each step the user extends the current script by defining a new data type or a new function. An important part of our method consists of proving functions to be a.b., so at any given point let the set $B_i \subset \Sigma_{\mathcal{S}}$ comprise those functions in the current script $\mathcal{S}$ that have been proven to be i-bounded. Note that B_1 will include

[9] Clearly, any Boolean expression can be transformed to this form by repeatedly applying DeMorgan's laws to it.

$$\frac{}{t \mathrel{\widehat{\preceq}} t}\ [\text{R1}] \qquad \frac{}{irc(s_1,\ldots,s_n) \mathrel{\widehat{\preceq}} t}\ [\text{R2}] \qquad \frac{s_i \mathrel{\widehat{\preceq}} t}{g(s_1,\ldots,s_n) \mathrel{\widehat{\preceq}} t}\ [\text{R3}]$$

for every $irc \in IRCONS_{\mathcal{S}}$ for every $g \in B_i$

$$\frac{\widehat{GTZ}(t),\ s_{i_1} \mathrel{\widehat{\preceq}} sel^{rc}_{i_1}(t),\ \ldots,\ s_{i_m} \mathrel{\widehat{\preceq}} sel^{rc}_{i_m}(t)}{rc(s_1,\ldots,s_n) \mathrel{\widehat{\preceq}} t}\ [\text{R4}]$$

for every $rc \in RCONS_{\mathcal{S}}$ with reflexive argument positions $i_1,\ldots,i_m$

$$\frac{t = rc(s_1,\ldots,s_n)}{\widehat{GTZ}(t)}\ [\text{R5}]$$

for every $rc \in RCONS_{\mathcal{S}}$

$$\frac{t \neq irc_1(sel^{irc_1}_1(t),\ldots,sel^{irc_1}_{m_1}(t)),\ldots,t \neq irc_n(sel^{irc_n}_1(t),\ldots,sel^{irc_n}_{m_n}(t))}{\widehat{GTZ}(t)}\ [\text{R6}]$$

where $irc_1,\ldots,irc_n$ are all the irreflexive constructors of T,
for every $T \in TYPES_{\mathcal{S}}$

Figure 2: Definition of the relation $\widehat{\preceq}$.

all the reflexive selectors in $SEL_{\mathcal{S}}$ by default, since all such selectors are 1-bounded. The intended modus operandi is as follows. Immediately after the user has defined a new function $f : T_1 \times \cdots \times T_n \to T$, the system tries to prove that f is terminating. If it fails, the definition is rejected and the user must either "try again" by providing an alternative definition (of the same mathematical object), or move on to another task. If f is proven to be terminating, then the next step is to attempt to determine whether it is i-bounded for every $i = 1,\ldots,k$ such that $T_i = T$. If we succeed in proving that f is i-bounded for such an i, we set $B_i \leftarrow B_i \cup \{f\}$. Exactly how we go about deciding whether f is i-bounded is discussed in Sec. 5.2.

The rule schemas in Fig. 5.1 are intended to define two relations $\widehat{\preceq}$ and $\widehat{GTZ}$ on $\mathcal{T}(\Sigma_{\mathcal{S}},\mathcal{V})$, relative to a set of size atoms, that approximate $\preceq$ and GTZ, respectively. For the remainder of this section, fix a script $\mathcal{S}$, sets

$B_i \subseteq \Sigma_{\mathcal{S}}$, and a map $\mathcal{V}$ on $TYPES_{\mathcal{S}}$. We will write $\mathcal{R}_{\mathcal{S}}$ for the logic program comprising all and only the $\mathcal{S}$-specific instances of rules [R1]–[R6]. E.g., assuming that $\mathcal{S}$ contains `NatNum` and `LamTerm`, and that `Minus` $\in B_1$, $\mathcal{R}_{\mathcal{S}}$ will contain, amongst others, the rules $\texttt{Minus}(s_1, s_2) \mathrel{\widehat{\preceq}} t \Leftarrow s_1 \mathrel{\widehat{\preceq}} t$ (an instance of [R3]), $\texttt{zero} \mathrel{\widehat{\preceq}} t \Leftarrow$ (instance of [R2]), and $\widehat{GTZ}(t) \Leftarrow t \neq \texttt{var(index(}t\texttt{))}$ (instance of [R6]). The *Herbrand universe* of the program $\mathcal{R}_{\mathcal{S}}$, denoted $HU(\mathcal{R}_{\mathcal{S}}, \mathcal{V})$ is defined as the set of all Herbrand terms that can be formed from the function symbols in $\Sigma_{\mathcal{S}}$ and the variables in the various $\mathcal{V}(T)$, where the latter as treated as constants.[10] Thus $HU(\mathcal{R}_{\mathcal{S}}, \mathcal{V})$ contains all the terms in $\mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$ as well as ill-typed but arity-respecting terms such as `tail(zero)`. Note that we define the Herbrand universe of $\mathcal{R}_{\mathcal{S}}$ not only with respect to the function symbols that appear in "the text" of $\mathcal{R}_{\mathcal{S}}$, but with respect to all the function symbols in $\Sigma_{\mathcal{S}}$ and the constants in the various $\mathcal{V}(T)$.

Now let $\{A_1, \ldots, A_n\}$ be a (possibly empty) set of size atoms over $\Sigma_{\mathcal{S}}$ and $\mathcal{V}$, and let $\mathcal{M}$ be the least Herbrand model of the program $\mathcal{R}_{\mathcal{S}} \cup \{A_1, \ldots, A_n\}$, with respect to $HU(\mathcal{R}_{\mathcal{S}}, \mathcal{V})$. Then we *define* $s \mathrel{\widehat{\preceq}} t \equiv \{A_1, \ldots, A_k\}$ to hold, for any $s, t \in \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$, iff $s \mathrel{\widehat{\preceq}} t$ is true in $\mathcal{M}$. By virtue of the completeness of SLD resolution, an equivalent, more operational definition, is to say that $s \mathrel{\widehat{\preceq}} t \equiv \{A_1, \ldots, A_k\}$ if *and only if* there is a successful SLD-derivation of $s \mathrel{\widehat{\preceq}} t$ from the program $\mathcal{R}_{\mathcal{S}} \cup \{A_1, \ldots, A_k\}$, a condition which we shall express by writing $\mathcal{R}_{\mathcal{S}} \cup \{A_1, \ldots, A_k\} \vdash s \mathrel{\widehat{\preceq}} t$. The latter notation draws more attention to the particular algorithmic aspects of our method, and for this reason we will hereafter use it instead of $s \mathrel{\widehat{\preceq}} t \equiv \{A_1, \ldots, A_k\}$.

The following result shows that, procedurally, this rule system is very efficient:

Proposition 5.1 *Consider the logic program $\mathcal{R}_{\mathcal{S}} \cup \{A_1, \ldots, A_k\}$ and pick any rule for selecting goals from a query. The SLD-tree of any ground query of the form $s \mathrel{\widehat{\preceq}} t$ has $O(|s|)$ nodes.*

The restriction to ground queries is critical, as the presence of variables[11] can easily lead to infinite derivations. For example, assuming that $\mathcal{R}_{\mathcal{S}}$ has a clause $f(s) \mathrel{\widehat{\preceq}} t \Leftarrow s \mathrel{\widehat{\preceq}} t$ for some $f \in B_1$, the query $s \mathrel{\widehat{\preceq}} t$ has an infinite

[10] From the viewpoint of $\mathcal{R}_{\mathcal{S}}$ the only variables are those that denote arbitrary terms in $HU(\mathcal{R}_{\mathcal{S}}, \mathcal{V})$. (In a Prolog implementation, these would be represented by identifiers with an upper-case first letter; in our presentation, in Fig. 5.1, they are represented by the letters s and t). At the meta-level of the rules, the "object variables" in the various $\mathcal{V}(T)$ are considered to be constants.

[11] Again, logic variables, not IFL variables.

SLD-tree. Of course for our purposes this is harmless since we are only interested in posing ground queries. We conclude:

Proposition 5.2 *For any $s, t \in \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$ and size atoms $A_1, \ldots, A_k$ over $\Sigma_{\mathcal{S}}$ and $\mathcal{V}$, $\mathcal{R}_{\mathcal{S}} \cup \{A_1, \ldots, A_k\} \vdash s \mathrel{\widehat{\preceq}} t$ can be decided in $O(|s|)$ time.*[12]

Soundness is not difficult to show either.

Proposition 5.3 *$\mathcal{R}_{\mathcal{S}} \cup \widehat{E_1} \cup \cdots \cup \widehat{E_n} \vdash s \mathrel{\widehat{\preceq}} t$ implies $\{E_1, \ldots, E_n\} \models s \preceq t$.*

At this point we have a sound, efficient, and fairly powerful sustem for deducing statements of the form $\{E_1, \ldots, E_n\} \models s \preceq t$. The following is a sample SLD-derivation proving $\{\mathtt{n} \neq \mathtt{zero}\} \models \mathtt{succ(Minus(pred(n),m))} \preceq \mathtt{n}$, assuming that $\mathtt{Minus} \in B_1$:

$$\begin{array}{rcl}
\mathtt{succ(Minus(pred(n),m))} \mathrel{\widehat{\preceq}} \mathtt{n} & \Leftarrow & [\mathrm{R4}] \\
\widehat{GTZ}(\mathtt{n}), \mathtt{Minus(pred(n),m)} \mathrel{\widehat{\preceq}} \mathtt{pred(n)} & \Leftarrow & [\mathrm{R6}] \\
\mathtt{n} \neq \mathtt{zero}, \mathtt{Minus(pred(n),m)} \mathrel{\widehat{\preceq}} \mathtt{pred(n)} & \Leftarrow & \mathtt{n} \neq \mathtt{zero} \\
\mathtt{Minus(pred(n),m)} \mathrel{\widehat{\preceq}} \mathtt{pred(n)} & \Leftarrow & [\mathrm{R3}] \\
\mathtt{pred(n)} \mathrel{\widehat{\preceq}} \mathtt{pred(n)} & \Leftarrow & [\mathrm{R1}] \\
& \Box &
\end{array}$$

A few additional rules are needed for inferring strict size inequalities. We single out four such rule schemas in Fig. 5.1. They should be self-explanatory, with the exception of [R9], which is based on the important notion of a function being strictly i-bounded *in a set of argument positions.* The basic idea is that, oftentimes, whether or not an i-bounded function f returns an object of *strictly smaller* size than that of its i^{th} argument depends on whether some of its arguments have positive size. For example, the 1-bounded `Minus` is also strictly 1-bounded whenever both of its arguments are non-zero. In symbols, $GTZ(\mathtt{n},\mathtt{m}) \models \mathtt{Minus(n,m)} \prec \mathtt{n}$, where we write $GTZ(t_1, \ldots, t_k)$ as an abbreviation for the conjunction $GTZ(t_1), \ldots, GTZ(t_k)$.

As another example, the selector `tail` (which is 1-bounded in virtue of being reflexive), is also strictly 1-bounded whenever its argument is a non-empty list, i.e. $GTZ(\mathtt{l}) \models \mathtt{tail(l)} \prec \mathtt{l}$. In fact all reflexive selectors strictly reduce arguments of positive size. In general, for any i-bounded g of n arguments and any non-empty subset $\{a_1, \ldots, a_m\} \subset \{1, \ldots, n\}$, we say that g is **strictly i-bounded in positions** $\{a_1, \ldots, a_m\}$ iff $g(s_1, \ldots, s_n) \prec s_i$ whenever $GTZ(s_{a_1}, \ldots, s_{a_m})$. Thus all reflexive selectors are strictly 1-bounded in $\{1\}$. In section 5.3 we will discuss in detail how we go about

[12] Counting resolution steps as the fundamental units of computation.

$$\frac{\widehat{GTZ}(t)}{irc(s_1,\ldots,s_n) \mathrel{\widehat{\prec}} t}\ [\mathrm{R7}] \qquad \frac{s_i \mathrel{\widehat{\prec}} t}{g(s_1,\ldots,s_n) \mathrel{\widehat{\prec}} t}\ [\mathrm{R8}]$$

for every $irc \in IRCON_{\mathcal{S}}$ $\qquad$ for every $g \in B_i$

$$\frac{\widehat{GTZ}(s_{a_1},\ldots,s_{a_m}),\ s_i \mathrel{\widehat{\preceq}} t}{g(s_1,\ldots,s_n) \mathrel{\widehat{\prec}} t}\ [\mathrm{R9}]$$

for every $g \in B_i$ proven to be strictly i-bounded in $\{a_1,\ldots,a_m\}$

$$\frac{s_{i_1} \mathrel{\widehat{\prec}} sel^{rc}_{i_1}(t),\ldots,s_{i_m} \mathrel{\widehat{\prec}} sel^{rc}_{i_m}(t)}{rc(s_1,\ldots,s_n) \mathrel{\widehat{\prec}} t}\ [\mathrm{R10}]$$

for every $rc \in RCON_{\mathcal{S}}$ with reflexive argument positions $i_1,\ldots,i_m$.

Figure 3: Definition of the relation $\widehat{\prec}$.

proving a function to be strictly i–bounded in a set of argument positions; for now the definition alone will suffice. It should be said, however, that for every $g \in B_i$ there will be *at most* one instance of [R9], a stipulation we choose to make in the interest of efficiency.[13]

From now on $\mathcal{R}_{\mathcal{S}}$ should be understood to comprise all $\mathcal{S}$-specific instances of [R1]–[R10], and our definitions of $HU(\mathcal{R}_{\mathcal{S}},\mathcal{V})$ and $\mathcal{M}$ should be adjusted accordingly. Similarly to our definition of $\widehat{\preceq}, \equiv$, we define $s \mathrel{\widehat{\prec}} t \equiv \{A_1,\ldots,A_k\}$ to mean that $s \mathrel{\widehat{\prec}} t$ is true in $\mathcal{M}$, or equivalently, that

$$\mathcal{R}_{\mathcal{S}} \cup \{A_1,\ldots,A_k\} \vdash s \mathrel{\widehat{\prec}} t. \tag{6}$$

As regards complexity, Prop. 5.2 is not affected by the addition of the new rules since [R1]–[R6] are self-contained. By contrast, deciding (6) is slightly more expensive (in the worst case), mainly because both strict and weak goals may be generated in the course of a derivation, on account of [R9]. In particular, it can be proved that the SLD-tree of any ground query $s \mathrel{\widehat{\prec}} t$ with respect to the program $\mathcal{R}_{\mathcal{S}} \cup \{A_1,\ldots,A_k\}$ has $O(|s|^2)$ nodes.

Proposition 5.4 *For any $s,t \in \mathcal{T}(\Sigma_{\mathcal{S}},\mathcal{V})$ and size atoms $A_1,\ldots,A_k$ over $\Sigma_{\mathcal{S}}$ and $\mathcal{V}$, $\mathcal{R}_{\mathcal{S}} \cup \{A_1,\ldots,A_k\} \vdash s \mathrel{\widehat{\prec}} t$ can be decided in $O(|s|^2)$ time.*

[13] In particular, Prop. 5.4 does hot hold without this provision.

With regard to soundness, the readers should have little trouble convincing themselves that an instance of [R7]–[R10] can never lead from a true antecedent to a false conclusion.

At this point most of the pieces of our method have fallen into place: to verify that a new function f is terminating, follow the algorithm given in page 15, with $\mathcal{R}_{\mathcal{S}} \cup \{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \vdash t_i \widehat{\preceq} x_i$ in place of $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models t_i \preceq x_i$, and likewise for the strict inequality test. Two things remain to be explained: how we prove that a function is i-bounded, and how we prove that a function is strictly i-bounded in a certain subset of its argument positions. These are respectively discussed in the next two sections.

We end this section with a simple illustration, a termination proof for the function `Member` defined in Sec. 2.1. First we need to approximate the measured subset of `Member`, and we begin by checking $i = 1$, the first argument position. There is only one recursive call in the body of the function, in r_3, and after computing $\widehat{E_3} = \emptyset, \widehat{\neg E_2} = \emptyset, \widehat{\neg E_1} = \{\mathtt{l} \neq \mathtt{empty}\}$, we easily determine $\mathcal{R}_{\mathcal{S}} \cup \{\mathtt{l} \neq \mathtt{empty}\} \vdash \mathtt{n} \widehat{\preceq} \mathtt{n}$ (through [R1]), thus concluding that $1 \in \mathrm{MS}_{\mathtt{Member}}$. Similarly, for $i = 2$, we readily determine that $\mathcal{R}_{\mathcal{S}} \cup \{\mathtt{l} \neq \mathtt{empty}\} \vdash \mathtt{tail(l)} \widehat{\preceq} \mathtt{l}$ (through [R3]). We infer that both argument positions are in the measured subset. Next, we need to verify that each time a recursive call is made, the size of at least one argument in the measured subset decreases. There is only one recursive call to consider. We begin by checking whether the first argument in the measured subset strictly goes down, but we find that $\mathcal{R}_{\mathcal{S}} \cup \{\mathtt{l} \neq \mathtt{empty}\} \nvdash \mathtt{n} \widehat{\prec} \mathtt{n}$, so our hopes shift to the second (and last) element of the measured subset. Here we successfully find that $\mathcal{R}_{\mathcal{S}} \cup \{\mathtt{l} \neq \mathtt{empty}\} \vdash \mathtt{tail(l)} \widehat{\prec} \mathtt{l}$ (by [R9] and [R6]), and hence we conclude that `Member` is terminating. It is easy to see that this derivation is an instance of a more general proof schema, based on combinations of [R9] and [R6], that works for *any* primitive recursive definition. Such definitions are recognized by the system very efficiently.

5.2 Proving that a function is argument-bounded

Once a newly defined function $f : T_1 \times \cdots \times T_n \to T$ has been proven terminating, the next order of business is to attempt to prove that f is i-bounded for every $i \in \{1, \ldots, n\}$ for which $T_i = T$. If we succeed in proving this for such an i, we set $B_i \leftarrow B_i \cup \{f\}$.

Our method will take advantage of the fact that we already know f to be terminating. We will essentially try to prove that $f(t_1, \ldots, t_n) \preceq t_i$ by induction on the running time of the call $f(t_1, \ldots, t_n)$; since f is total, the said time must be finite. We do this by showing, with the aid of the appropriate inductive hypotheses, that $\{E_j, \neg E_{j-1}, \ldots, \neg E_1\} \models r_j \preceq x_i$ for each $j = 1, \ldots, m+1$. In particular, for each such j, let r_j contain a recursive call $f(s_1, \ldots, s_n)$. By the inductive hypothesis, we know that $f(s_1, \ldots, s_n) \preceq s_i$, which can be cast in a way that incorporates transitivity as

$$f(s_1, \ldots, s_n) \preceq t \Leftarrow s_i \preceq t \ \ \text{[Ind]}.$$

We then compute $\mathcal{R}_{\mathcal{S}} \cup \widehat{E_j} \cup \widehat{\neg E_{j-1}} \cup \cdots \cup \widehat{\neg E_1} \cup \{\text{[Ind]}\} \vdash r_j \mathrel{\widehat{\preceq}} x_i$ (i). If the verdict is negative we halt with a "No" answer (which is what makes the method incomplete, as f may in fact be i-bounded), otherwise we continue with $j+1$. If (i) goes through for every $j = 1, \ldots, m+1$, we conclude that f is i-bounded, a clearly sound conclusion. Note that if an r_j contains multiple recursive calls $f(s_1^1, \ldots, s_n^1), \ldots, f(s_1^k, \ldots, s_n^k)$, we postulate an inductive hypothesis $f(s_1^p, \ldots, s_n^p) \preceq t \Leftarrow s_i^p \preceq t$ $[\text{Ind}_p]$ for each one of them, $p = 1, \ldots, k$; i.e., we compute

$$\mathcal{R}_{\mathcal{S}} \cup \widehat{E_j} \cup \widehat{\neg E_{j-1}} \cup \cdots \cup \widehat{\neg E_1} \cup \{[\text{Ind}]_1, \ldots, [\text{Ind}]_k\} \vdash r_j \mathrel{\widehat{\preceq}} x_i.$$

To illustrate, we show how this algorithm would prove that `Minus` is 1-bounded. For $j = 1$ there are no recursive calls in $r_1 \equiv$ `m`, and hence no inductive hypotheses to be posited; we simply need to verify that $\mathcal{R}_{\mathcal{S}} \cup \{\texttt{n} = \texttt{zero}\} \vdash \texttt{m} \mathrel{\widehat{\preceq}} \texttt{m}$, which is trivial (by [R1]). The next statement, $j = 2$, contains the recursive call `Minus(pred(m),pred(n))`, so let [Ind] be the clause representing our inductive hypothesis for this recursive call:

$$\texttt{Minus(pred(m),pred(n))} \mathrel{\widehat{\preceq}} t \Leftarrow \texttt{pred(m)} \mathrel{\widehat{\preceq}} t.$$

We now have to check whether

$$\mathcal{R}_{\mathcal{S}} \cup \{\texttt{n} \neq \texttt{zero}\} \cup \{\text{[Ind]}\} \vdash \texttt{Minus(pred(m),pred(n))} \mathrel{\widehat{\preceq}} \texttt{m}.$$

The goal $\texttt{Minus(pred(m),pred(n))} \mathrel{\widehat{\preceq}} \texttt{m}$ unifies with the head of [Ind] under the binding $t \mapsto \texttt{m}$, thereby yielding the subgoal $\texttt{pred(m)} \mathrel{\widehat{\preceq}} \texttt{m}$, which follows easily from [R3]. Thus we conclude that `Minus` is 1-bounded.

5.3 Deriving strict inequality lemmas

After a function $f : T_1 \times \cdots \times T_n \rightarrow T$ has been proven to be i-bounded for some i, the next step is to try to obtain a strict inequality lemma of the form [R9]. The idea is to iterate through the power-set of $\{1, \ldots, n\}$, trying, for each subset $\{a_1, \ldots, a_m\} \subset \{1, \ldots, n\}$, to prove that

$$GTZ(x_{a_1}, \ldots, x_{a_m}) \models f(x_1, \ldots, x_n) \prec x_i. \tag{7}$$

The subsets of $\{1, \ldots, n\}$ are visited in order of increasing cardinality. We stop when we find the first subset $\{a_1, \ldots, a_m\}$ that satisfies (7) or when there are no more subsets to consider, whichever occurs first. In the former case we add to $\mathcal{R}_{\mathcal{S}}$ the following instance of [R9]: $f(s_1, \ldots, s_n) \mathrel{\widehat{\prec}} t \Leftarrow \widehat{GTZ}(s_{a_1}, \ldots, s_{a_m}), s_i \mathrel{\widehat{\preceq}} t$, a slightly more general formulation of (7) incorporating transitivity. Thus $\mathcal{R}_{\mathcal{S}}$ will contain at most one instance of [R9] for every $g \in B_i$. The fact that this algorithm is exponential in the number of arguments of f is inconsequential since in practice that number ranges from 1 or 2 in most cases to 5 or 6 in rare instances.

Now (7) would certainly follow if we verified that

$$\mathcal{R}_{\mathcal{S}} \cup \widehat{GTZ}(x_{a_1}, \ldots, x_{a_m}) \cup \widehat{E_j} \cup \widehat{\neg E_{j-1}} \cup \cdots \cup \widehat{\neg E_1} \vdash r_j \mathrel{\widehat{\prec}} x_i \tag{8}$$

for *every* $j = 1, \ldots, m+1$. But we would be doing more work than necessary that way. We can cut corners by completely ignoring those branches (E_j, r_j) that are inconsistent with the antecedent $GTZ(x_{a_1}, \ldots, x_{a_m})$, in the sense that if E_j holds then $GTZ(x_{a_1}, \ldots, x_{a_m})$ could not possibly be true. Thus, we check (8) only for those j for which $Con(E_j \wedge \neg E_{j-1} \wedge \cdots \wedge \neg E_1, \{x_{a_1}, \ldots, x_{a_m}\}) = \texttt{true}$, where Con is the algorithm below that takes an $E \in \mathcal{B}(\Sigma_{\mathcal{S}}, \mathcal{V})$ and a set of terms $\{t_1, \ldots, t_k\}, t_i \in \mathcal{T}(\Sigma_{\mathcal{S}}, \mathcal{V})$, and returns `false` only when E logically implies $\neg GTZ(t_i)$ for some $i \in \{1, \ldots, k\}$:

$Con\ (t_i = irc(\cdots))\ \{t_1, \ldots, t_k\} =$ `false |`
$Con\ (E_1 \wedge E_2)\ T = Con(E_1, T)$ `andalso` $Con(E_2, T)$ `|`
$Con\ (E_1 \vee E_2)\ T = Con(E_1, T)$ `orelse` $Con(E_2, T)$ `|`
Con _ _ = `true`;

This is a conservative approximation. If $Con(E, \{t_1, \ldots, t_k\})$ returns `false` then we can be sure that $E \models \neg GTZ(t_i)$ for some $i \in \{1, \ldots, k\}$, but if the result is `true` then we may still have $E \models \neg GTZ(t_i)$. For our purposes this is acceptable; it simply means that we will attempt to verify (8) even though this is not necessary in proving (7).

We can also take advantage of the fact that f is known to terminate and form appropriate inductive hypotheses, as we do when proving i-boundedness. In particular, when trying to prove $r_j \mathrel{\widehat{\prec}} x_i$ for some r_j containing $k > 0$ recursive calls $f(s_1^1, \ldots, s_n^1), \ldots, f(s_1^k, \ldots, s_n^k)$, we can assume k inductive hypotheses $f(s_1^p, \ldots, s_n^p) \mathrel{\widehat{\prec}} t \Leftarrow \widehat{GTZ}(s_{a_1}^p, \ldots, s_{a_m}^p), s_i \mathrel{\widehat{\preceq}} t$ [Ind$_p$], one for each recursive call, $p = 1, \ldots, k$. Accordingly, instead of (8) we attempt to verify the stronger

$$\begin{gathered}\mathcal{R}_{\mathcal{S}} \cup \{\widehat{GTZ}(x_{a_1}, \ldots, x_{a_m}), [\mathrm{Ind}_1], \ldots, [\mathrm{Ind}_k]\} \\ \cup\, \widehat{E_j} \cup \widehat{\neg E_{j-1}} \cup \cdots \cup \widehat{\neg E_1} \vdash r_j \mathrel{\widehat{\prec}} x_i.\end{gathered}$$

If the above goes through for every j for which $E_j \wedge \neg E_{j-1} \wedge \neg E_j \wedge \cdots \wedge \neg E_1$

is consistent with $GTZ(x_{a_1},\ldots,x_{a_m})$, we conclude that f is strictly i-bounded in positions $\{a_1,\ldots,a_m\}$.

We again illustrate on `Minus`. Assuming we already have `Minus` $\in B_1$, we begin our traversal of the power-set of $\{1,2\}$ by considering $\{1\}$, trying to derive the lemma $\texttt{Minus}(s_1,s_2) \mathrel{\hat{\prec}} t \Leftarrow \widehat{GTZ}(s_1), s_1 \mathrel{\hat{\preceq}} t$. Since this does not in fact hold, it is not derivable. We then consider $\{2\}$, which fails for similar reasons. Finally we come to check $\{1,2\}$. We find $Con(\texttt{n} = \texttt{zero}, \{\texttt{n},\texttt{m}\}) =$ `false`, thus we only need to consider the second statement, for which we need to verify that

$$\mathcal{R}_{\mathcal{S}} \cup \{\widehat{GTZ}(\texttt{n},\texttt{m}), [\text{Ind}], \texttt{m} \neq \texttt{zero}\} \vdash \texttt{Minus(pred(n),pred(m))} \mathrel{\hat{\prec}} \texttt{n} \quad (9)$$

where [Ind] is the inductive clause

$$\texttt{Minus(pred(n),pred(m))} \mathrel{\hat{\prec}} t \Leftarrow \widehat{GTZ}(\texttt{pred(n)},\texttt{pred(m)}), \texttt{pred(n)} \mathrel{\hat{\preceq}} t.$$

This does not help us here, but since `Minus` $\in B_1$, $\mathcal{R}_{\mathcal{S}}$ contains the clause $\texttt{Minus}(s_1,s_2) \mathrel{\hat{\prec}} t \Leftarrow s_1 \mathrel{\hat{\prec}} t$ (as an instance of [R8]), whose head unifies with our goal in (9) with the bindings $s_1 \mapsto \texttt{pred(n)}, s_2 \mapsto \texttt{pred(m)}, t \mapsto \texttt{n}$, backchaining to $\texttt{pred(n)} \mathrel{\hat{\prec}} \texttt{n}$, which follows immediately from [R9] via $\widehat{GTZ}(\texttt{n})$ and [R1]. We thus conclude that `Minus` is strictly 1-bounded in $\{1,2\}$ and augment $\mathcal{R}_{\mathcal{S}}$ with the appropriate instance of [R9].

6 Examples and improvements

We demonstrate the method on some functions that are well-known not to have natural primitive recursive definitions. One of them is Euclid's algorithm, which in IFL can be expressed as follows:

```
function Gcd(n,m:NatNum):NatNum =
if n = zero ∨ m = zero then Max(n,m)
else
   if Leq(n,m) = true then Gcd(n,Minus(m,n))
   else
      Gcd(Minus(n,m),m);
```

Because `Minus` $\in B_1$, we are able to correctly identify the measured subset of `Gcd` as $\{1,2\}$: For the first position, we have $\mathcal{R}_{\mathcal{S}} \cup \mathrm{A} \vdash \texttt{n} \mathrel{\hat{\preceq}} \texttt{n}$ for the first recursion, and $\mathcal{R}_{\mathcal{S}} \cup \mathrm{A} \vdash \texttt{Minus(n,m)} \mathrel{\hat{\preceq}} \texttt{n}$ for the second one, where A is the set of size atoms $\{\texttt{n} \neq \texttt{zero}, \texttt{m} \neq \texttt{zero}\}$. We then find that in the first recursive call the size of the second argument strictly decreases (specifically, $\mathcal{R}_{\mathcal{S}} \cup A \vdash \texttt{Minus(m,n)} \mathrel{\hat{\prec}} \texttt{m}$, by [R9] and [R5]), while in the second recursion it is the first argument that strictly decreases ($\mathcal{R}_\mathcal{S} \cup A \vdash$ Minus(n,m) $\hat{\prec}$ n, again by [R9] and [R5]).

Next we consider QuickSort, whose implementation uses the auxilliary functions GetSmaller(n,l), which returns a copy of l containing only those numbers that are $\leq$ n, and GetLarger(n,l), which does the opposite:

```
function GetSmaller(n:NatNum,l:NatList):NatList =
if l = empty then empty
else
   if Greater(head(l),n) = true then GetSmaller(n,tail(l))
   else
      add(head(l),GetSmaller(n,tail(l)));
```

This is a primitive recursive definition, so we readily find the measured subset to comprise both argument positions, with the second argument strictly decreasing with each recursive call: $\mathcal{R}_\mathcal{S} \cup \{$l $\neq$ empty$\} \vdash$ tail(l) $\hat{\prec}$ l (by virtue of [R9] and [R6], as with all primitive recursions). The termination of GetLarger is similarly established. The system will also successfully deduce that the two functions are 2-bounded, augmenting $\mathcal{R}_\mathcal{S}$ with the following instances of [R3] and [R8]:

$$\begin{array}{lcl} \texttt{GetSmaller}(s_1, s_2) \mathrel{\hat{\preceq}} t & \Leftarrow & s_2 \mathrel{\hat{\preceq}} t \\ \texttt{GetSmaller}(s_1, s_2) \mathrel{\hat{\prec}} t & \Leftarrow & s_2 \mathrel{\hat{\prec}} t \end{array}$$

With the aid of these lemmas the termination of QuickSort below is easily verified:

```
function QuickSort(l:NatList):NatList =
if l = empty then empty
else
 Append(QuickSort(GetSmaller(head(l),tail(l))),
      add(head(l),QuickSort(GetLarger(head(l),tail(l)))));
```

In practice the existing implementation of the method incorporates a few minor enhancements which increase the verification power of the system without significantly affecting its complexity. One of them can be illustrated via the MergeSort function below:

```
function MergeSort(l:NatList):NatList =
if l = empty then empty
else
 if tail(l) = empty then l
 else
    Merge(MergeSort(RemoveOdd(l)),MergeSort(RemoveEven(l)));
```

where `RemoveOdd(l)` returns a copy of `l` with all elements of `l` in odd positions removed:

```
function RemoveOdd(l:NatList):NatList =
if l = empty  ∨ tail(l) = empty then empty
else
   add(head(tail(l)),RemoveOdd(tail(tail(l))));
```

and

```
function RemoveEven(l:NatList):NatList =
if l = empty then empty
else
   if tail(l) = empty then l
   else
       add(head(l),RemoveEven(tail(tail(l))));
```

In Sec. 5.3 we pointed out that, oftentimes, whether or not an i-bounded function f decreases the size of its i^{th} argument depends on whether some of its arguments have positive size. A more general version of this claim is in fact true. Consider a unary a.b. function f for simplicity. Whether or not $f(w) \prec w$ often depends not merely on whether $GTZ(w)$ but rather on whether $GTZ(sel(w))$ or $GTZ(sel(sel(w)))$, etc., for some selector sel of the appropriate type. This is the case with `RemoveEven`, for example; `RemoveEven(l)` $\prec$ `l` does *not* follow from GTZ(`l`) (all one-element lists are counter-examples), but it does follow from GTZ(`tail(l)`). As another example, a halving function `Half(n)` would decrease `n` only if `n` > 1; thus `Half(n)` $\prec$ `n` would be entailed by GTZ(`pred(n)`) but not by GTZ(`n`).

These cases arise in programming naturally, and to account for them the algorithm given in Sec. 5.3 iterates through the power-set of $\{x_1, \ldots, x_n\} \cup \Delta$ rather than just $\{x_1, \ldots, x_n\}$, where Δ is the set of all terms s such that $s \neq irc(\cdots)$ or $s = rc(\cdots)$ is a size atom in the body of f. Thus in the case of `RemoveEven` we would iterate through $\{$`l`, `tail(l)`$\}$ rather than just $\{$`l`$\}$. As before, for every visited subset X, we examine every (E_j, r_j) for which $Con(E_j, X) =$ `true`, and we try to derive $r_j \mathrel{\widehat{\prec}} x_i$ from $\mathcal{R}_{\mathcal{S}}$, $GTZ(X)$, and the appropriate inductive hypotheses. If we succeed in this for every "consistent" (E_j, r_j), we augment $\mathcal{R}_{\mathcal{S}}$ with the appropriate clause. For `RemoveEven` that would be `RemoveEven(`s`)` $\mathrel{\widehat{\prec}} t \Leftarrow \widehat{GTZ}$(`tail(`$s$`)`)$, s \mathrel{\widehat{\preceq}} t$. This lemma is in fact the key ingredient in the termination proof for `MergeSort`.

7 Where the method fails

There are three main factors that can prevent the method from recognizing a terminating function f. We discuss each below, in order of increasing importance.

Firstly, f may not be normal. This would not pose a serious problem in practice, as non-normal algorithms are not common.

Secondly, the size measure (5) may not be a termination function for f, either because $\mathrm{MS}_f = \emptyset$ or because it is not the case that the size of some argument in MS_f strictly decreases with each recursive call. For instance, the size decrease could be erratic (consider a zig-zag pattern where with each recursion one argument increases by one and the other decreases by two, alternately), or there may be no size decrease at all—the arguments may in fact increase. Consider, for instance, a function $f(n)$ which halts for $n \geq 1000$ and recurses on $n+1$ for $n < 1000$. A slightly more realistic example given by Walther [13] is an implementation of `Minus(n,m)` which returns `zero` if `n` $\leq$ `m` and `succ(Minus(n,succ(m)))` otherwise. Here it is the difference between `n` and `m` that strictly decreases with each recursion, not the size of any arguments. Such scenarios are not very frequent, however, so this case need not be a serious cause of concern either.

Thirdly, (5) may be a termination function but our method may not be able to prove it. This would be the most frequent cause of failure. There are several places where incompleteness creeps in. To take a simple example, the method relies heavily on scanning boolean expressions in search of size atoms of the form $s \neq irc(\cdots)$ and $s = rc(\cdots)$, in order to conclude $GTZ(s)$. If a boolean expression conveys this fact not through a size atom but through some other expression that logically implies it, our method will falter. Consider, for example, an implementation of `Plus(n,m)` which retuns `succ(Plus(n,pred(m)))` if `m` $>$ `zero` and `n` otherwise. To verify that the argument `pred(m)` strictly decreases, our method will be looking verbatim for an atom such as `m` $\neq$ `zero`; it is not smart enough to deduce that `m` $>$ `zero` in fact implies `m` $\neq$ `zero`.

Another serious source of incompleteness stems from failing to recognize that a function `g` is a.b. If that function is later used in a recursive call `f(g(x))`, we will not be able to verify that `f` terminates. But even if we prove that `g` is a.b., we may still fail to derive any strictness lemmas (in the manner of Sec. 5.3) that we might later need for verifying that the size of `g(n)` is actually smaller than that of `n`. This is by far the most serious pitfall. In general, proving that `g` is a.b. is not enough. We must obtain *strictness conditions*, i.e., conditions which are sufficient to guarantee that g will decrease its argument. Now the only type of strictness conditions our method considers[14] are $GTZ(\mathtt{x})$ (or $GTZ(\{\mathtt{x}_{\mathtt{a}_1}, \ldots, \mathtt{x}_{\mathtt{a}_m}\})$ in general, when g has multiple arguments). There are two problems here:

- These strictness conditions are very simple and although they often turn out to be sufficient in practice, there are cases in which they are not, i.e., cases in which it is not true that `g(x)` will decrease x whenever $GTZ(\mathtt{x})$.
- And even if these conditions are in fact sufficient, our method may not be able to verify this, i.e., the algorithm we give in Sec. 5.3 may fail.

8 Possible future extensions

Extensions are possible in two different directions: in the language and in the method's power. We briefly discuss each possibility.

8.1 Extending the language

The language in this paper is quite rudimentary. If the method is to have any practical value outside of laconic theorem provers,[15] it must be made usable in the more realistic context of a modern language with rich features such as polymorphism, high-order functions, mutual recursion, pattern-matching, and state. We briefly speculate on how the method would interact with each of these features:

Polymorphism

This should be a straightforward addition. The method is based on recognizing irreflexive constructors, reflexive selectors, and a.b. functions, and neither of these is affected by the presence of type variables. From the method's viewpoint the polymorphic datatype

```
datatype 'a List = empty | add(head:'a,tail:'a List);
```

is identical to the monomorphic datatype `NatList` of Sec. 2.1. All that matters is that the constructor `empty` is irreflexive while `add` is reflexive in the second argument position. According to definition (3), whether a list is

[14] Our method will also automatically instantiate [R8] immediately after g has been proven a.b., which is a type of strictness condition (one that we can take for granted in virtue of the transitivity of $\prec$), but experience shows that [R8] is not as useful as [R9].

[15] Perhaps by a compiler that needs a guarantee that certain functions terminate.

made of numbers, strings, or other lists is irrelevant to its size. For these reasons, the termination proof for, say, the function `Member` in Sec. 2.1, would carry over verbatim to a polymorphic version of the function.

High-order functions

Since our method is based on size and there appears to be no natural way of designating any finite quantity as the size of a function, the method could simply ignore those argument positions receiving functions as arguments. This is a perfectly reasonable strategy. In high-order languages such as Scheme or ML, termination usually follows because some finite measure (often the size) of *non-functional* arguments strictly decreases with each recursive call; functional arguments rarely play a role in this—consider the functionals `map` and `foldr`, for instance.

Mutual recursion

Mutual recursion does not pose a serious challenge either. One straightforward way of handling it is to unfold the recursion chain. Suppose, for example, that we extended IFL with a simple `and` construct like that of ML for defining mutually recursive functions:

```
function Even(n:NatNum):Boolean =
if n = zero then true
else
    Odd(pred(n))
and
function Odd(m:NatNum):Boolean =
if m = zero then false
else
    Even(pred(m));
```

To verify the termination of `Even`, we replace all calls `Odd(`s`)` in its body with B[s/`m`] where B is the body of `Odd` and `m` its parameter.[16] In this case we would obtain:

```
function Even(n:NatNum):Boolean =
if n = zero then true
else
   if pred(n) = zero then false
   else
```

[16] We may have to rename B first to avoid variable captures.

```
        Even(pred(pred(n)));
```
whose termination is easily verified by our method.

Pattern matching

This is also unproblematic. Pattern-matching is merely a convenience for the programmer that (greatly) enhances readability; it is easily desugared into the type of notation we have been using. E.g. a clause such as `Append add(x,xs) y = add(x,Append(xs,y))` becomes

`if l` $\neq$ `empty then add(head(l),Append(tail(l),y))`,

where `l` stands for the previously anonymous argument `add(x,xs)`. Alternatively, instead of desugaring we can tailor the method to directly support definitions by pattern matching. This simply requires us to identify those variables in a pattern p that appear in a reflexive constructor position; whenever we subsequently encounter such a variable as an argument to a recursive call inside the scope of p, we treat it as an application of a reflexive selector. Note that extracting size atoms in order to derive $GTZ(s)$ assertions is no longer necessary under pattern-matching, as that information is already handed to us for free.

State

This is by far the most troublesome feature. Suppose we had references (cells), for instance. One may be inclined to regard the "size" of a cell as the size of whatever object is stored therein. This would seem to allow us to prove the termination of functions that recursively call themselves with cells containing smaller and smaller objects. But side effects rear their ugly head: `function f(n:Cell-Of NatNum):NatNum = if !n = 0 then 0 else (n := !n + 1; f(cell(!n - 1)))`.[17] A naive extension of our method will conclude that `f` terminates, even though it clearly gets into an infinite loop whenever `!n` is positive. Therefore, it seems wiser to simply ignore reference arguments, just as we suggested for functions. Algorithms that perform course-of-values recursion on cell contents, after all, are not common; algorithms that terminate on account of such recursion are even less common. Of course in the presence of unrestricted destructive assignment (e.g. Scheme's `set!`) the above perils are unavoidable and soundness can no longer be guaranteed. Even in those languages, however,

[17]Where `!` and `:=` are the dereferencing and assignment operators, respectively, and $(E_1; \cdots; E_n)$ is a sequencing form

the method may be employed beneficially if assignment is used in a disciplined manner.

8.2 Increasing the deductive power

There are several ways to make the inference engine more sophisticated. The present method could be the starting point for proving termination. In case of failure, we could try another approach from a certain fixed "bag" of techniques. One such technique can be based on the lexicographic extension of $\prec$ to finite sequences of terms:

$$[s_1, s_2, \ldots, s_n] \prec [t_1, t_2, \ldots, t_n] \Leftrightarrow (s_1 \prec t_1) \vee (s_1 \preceq t_1 \wedge s_2 \prec t_2) \vee \cdots .$$

We could then try to verify

$$\mathcal{R}_{\mathcal{S}} \cup E_j \cup \widehat{\neg E_{j-1}} \cup \cdots \cup \widehat{\neg E_1} \vdash [t_1, \ldots, t_n] \mathrel{\widehat{\prec}} [x_1, \ldots, x_n]$$

for every recursive call $f(t_1, \ldots, t_n), j = 1, \ldots, m+1$. Note that this would allow us to recognize non-normal algorithms. We can easily verify the termination of Ackerman's function, for example, with this particular method.

Other possibilities include the incorporation of standard techniques for proving termination of rewrite systems [2, 12, 6, 3], working with measures other than size, and enriching the existing rule system with additional clauses. Clearly, however, after a certain point the power of the system becomes inversely proportional to its efficiency.

Acknowldegements

The author gratefully acknowledges David McAllester's valuable input and contributions to the ideas in this paper, as well as much helpful feedback by Olin Shivers. Any remaining errors, of course, are solely the fault of Dr. Shivers.

References

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

[2] Nachum Dershowitz and Z. Manna, Proving termination with multiset orderings, *Commun. ACM* **22**, 1979, 465-476.

[3] N. Dershowitz, Termination of Rewriting, *J. Symbolic Computation* (1987), **3**, 69-116.

[4] 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.

[5] K. Gödel, Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes, *Dialectica* **12**, 1958, 280-287. English translation in: On a hitherto unexplored extension of the finitary standpoint, *J. Philos. Logic* **9**, 1980.

[6] J. V. Guttag, D. Kapur, D. R. Musser, On proving uniform termination and restricted termination of rewriting systems, *SIAM J. Comput.* **12**, 1983, 189-214.

[7] Z. Manna, *Termination of Algorithms*, Ph.D Thesis, Computer Science Department, Carnegie–Mellon University, Pittsburgh, PA 1968.

[8] Z. Manna, *Mathematical Theory of Computation*, McGraw Hill, New York, 1974.

[9] D. McAllester and K. Arkoudas, Walther recursion, CADE 13, 1996.

[10] B. Nordström, K. Peterson, and J. Smith, *Programming in Martin-Lof's type theory*, International series of monographs on Computer Science, 7, Oxford Science Publications, 1990.

[11] J. S. Moore, A mechanical proof of the termination of Takeuchi's function, *Inf. Process. Let.* **9** (4), 1979, 176-181.

[12] A. Pettorossi, Comparing and putting together recursive path orderings, simplification orderings and non-ascending property for termination proofs of term rewriting systems, Proceedings of the Eighth EATCS International Colloquium on Automata, Languages and Programming, Acre, Israel, *Springer Lec. Notes Comp. Sc.* **115**, 1981, 432-447.

[13] C. Walther, On proving the termination of algorithms by machine, *Artificial Intelligence*, **71**, 1994, 101-157.