Walther Recursion

David McAllester[1] and Kostas Arkoudas[2]

[1] AT&T Labs, 600 Mountain Ave, Murray Hill N.J. 07974, dmac@research.att.com, (908)582-5412

[2] MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge MAss. 02139, koud@ai.mit.edu

Abstract. Primitive recursion is a well known syntactic restriction on recursive definitions which guarantees termination. Unfortunately many natural definitions, such as the most common definition of Euclid's GCD algorithm, are not primitive recursive. Walther has recently given a proof system for verifying termination of a broader class of definitions. Although Walther's system is highly automatible, the class of acceptable definitions remains only semi-decidable. Here we simplify Walther's calculus and give a syntactic criterion on definitions which guarantees termination. This syntactic criteria generalizes primitive recursion and handles most of the examples given by Walther. We call the corresponding class of acceptable definitions "Walther recursive".

1 Introduction

One of the central problems in verification logics, such as the Boyer-Moore theorem prover [2], [10], is the need to prove termination for recursive definitions. Many logics, such as that of Boyer and Moore, assume that all function symbols define total functions. Even in systems where partial functions are allowed, such as the IMPS system [11], proofs of termination are still important. For example, proving a lemma of the form $\forall x\ f(x) > x$ will still require proving that f terminates.

Some definitions can be proved to terminate by translation into a term rewriting system and the application of standard term rewriting techniques [4, 3]. Unfortunately, all automated systems for termination analysis for rewrite systems use well founded term orders such that $u \leq v$ implies $f(\ldots u \ldots) \leq f(\ldots v \ldots)$ and $u \leq f(\ldots u \ldots)$. There are many natural functional definitions where termination can not be verified with any such ordering. For example, under the natural definition of Euclid's GCD algorithm we have

$$\texttt{GCD}(6,\ 2) \rightarrow \texttt{GCD}(\texttt{MINUS}(6,\ 2),\ 2).$$

By the above properties we must have $6 \leq \texttt{MINUS}(6,\ 2)$ and $\texttt{GCD}(\texttt{MINUS}(6,\ 2),\ 2) \leq \texttt{CGD}(6,\ 2)$. The problem here is the use of orderings on terms rather than orderings on values. Clearly the value of $\texttt{MINUS}(6,\ 2)$ is smaller than 6.

The Boyer-Moore prover allows termination proofs based on user-defined orderings. The orderings must be proved to be well founded and every recursive

call must decrease a specified measure argument according to the user-defined ordering. This allows for termination proofs of a very large class of recursive definitions. Unfortunately, it also requires considerable user interaction both in defining an appropriate well founded order and in proving that the order is decreased on recursive calls. Such user interaction often requires an understanding of the internal machinery of the theorem prover and can be quite difficult for naive users.

Primitive recursion is a well known syntactic condition which ensures termination. Although primitive recursion was originally formulated for the natural numbers, it has a natural generalization to the Herbrand universe of first order terms. With this simple generalization a wide variety of functions can be given natural primitive recursive definitions. For example, the natural definitions of the Lisp functions APPEND, REVERSE, MEMBER, UNION, and FLATTEN are all primitive recursive. Unlike sophisticated termination analyses, primitive recursion is a simple syntactic property easily checked by the programmer.

Unfortunately, many functions, such as the greatest common divisor and a variety of sorting functions, do not have natural primitive recursive definitions. The functions themselves (as opposed to their natural definitions) are primitive recursive — they can be given unnatural definitions that are syntactically primitive recursive. But the unnatural primitive recursive definitions make formal verification more difficult. We would like an easy formal proof, for example, that GCD(x, y) divides both x and y — or that a given sort procedure produces a sorted list.

Primitive recursion is a syntactic condition ensuring that recursive calls involve smaller arguments. The notion of size is fixed — there is no choice of ordering. Over the natural numbers one uses the standard notion of size. Over the Herbrand universe we use the normal notion of size of terms, i.e., number of subterms.[3] The natural definitions of GCD and various sorting procedures *do* have the property that recursive calls involve smaller arguments — there is no problem with the fixed value ordering. The problem lies in the syntactic test ensuring reduction. In the GCD function the recursive call is of the form GCD(MINUS(x, y), x) in a context where we know $x > 0$ and $y > 0$. Primitive recursion on natural numbers only allows recursive calls of the form f(PRED(x), ...) where PRED is the predecessor function and the call occurs in a context where we know $x > 0$.

In our formulation of the generalization to Herbrand terms, primitive recursion allows recursive calls of the form $f(\Pi_{c,i}(x), \ldots)$ where $\Pi_{c,i}$ is the "projection" or "selector" function such that $\Pi_{c,i}(c(x_1, \ldots, x_n)) = x_i$. This recursive call must occur in a context where x is known to be an application of the constructor c. The Lisp functions CAR and CDR are examples of projection functions and one can verify that, for example, APPEND is primitive recursive. Various sorting algorithms do not have natural primitive recursive definitions. For example, a natural definition for quick sort has the recursive call SORT(DIFF(CDR(x), y)))

[3] A slightly more sophisticated notion of size for Herbrand terms is introduced in section 4.

where DIFF is the set difference functions (which is itself primitive recursive). Again, the problem is not the choice of ordering, it is the method of verifying that the arguments are reduced.

Walther [9] has developed a calculus which is quite effective at deriving assertions of the form $u < x$ where u is a term containing the variable x. For example, if x and y are nonzero then $\texttt{MINUS}(x,\ y) < x$. If x is nonempty then $\texttt{DIFF}(\texttt{CDR}(x),\ y) < x$. The basic idea behind Walther's calculus is that certain user-defined functions act as "reducers" and others act as "conservers". In the term $\texttt{DIFF}(\texttt{CDR}(x),\ y) < x$ the function CDR acts as a reducer and the function DIFF acts as a conserver. More generally, a function f reduces its ith argument if $f(x_1,\ \ldots,\ x_n) < x_i$. Analogously, f conserves its ith argument if $f(x_1,\ \ldots,\ x_n) \leq x_i$. Subtraction reduces its first argument whenever both arguments are nonzero. The function CDR reduces its argument whenever that argument is a cons cell. The function DIFF conserves its first argument. Walther's calculus is a system for soundly inferring assertions of the form $u < x$ where u consists of appropriate applications of reducers and conservers.

As given in [9], Walther's calculus is so rich that determining whether the calculus derives $u < x$ is undecidable. The contribution of this paper is a syntactic type discipline which captures the essence of Walther's reducer-conserver analysis. By "syntactic" we mean an analysis that is both efficiently decidable and that can be performed by inspection when writing definitions — A programmer can easily check whether a given definition satisfies syntactic requirements. As in other type disciplines, programmers will have to write programs in a certain style — they must follow the discipline. However, the discipline of Walther recursion is significantly more liberal than that of primitive recursion.

Walther recursion is a first order type discipline.[4] The type system is founded on the concept of a regular data type over the Herbrand universe of terms built from data constructors. We require that data types be user-specified and that all procedure definitions be explicitly annotated with input and output types. This simplifies the system and allows our formulation to focus on Walther's reducer-conserver analysis independent of the type inference problem for regular types. See [1, 6] for algorithms for inferring regular types.

Section 2 describes the concept of a regular type and introduces the class of "monomorphic" regular types. The monomorphism assumption allows types to be viewed as disjoint unions of "atomic" types. This disjoint union property plays an important role in the syntactic analysis of conditional expressions. Section 3 describes our programming language and a basic type analysis. Section 4 gives a formal presentation of our version of Walther's reducer-conserver analysis. Section 4 also gives Walther recursive definitions of GCD and substitution in the λ-calculus. Section 5 gives some examples of Walther recursive definitions of sorting algorithms.

[4] However, it should be straightforward to generalize Walther recursion to higher order types in a manner analogous to Gödel's system T.

2 Regular Types over Monomorphic Constructors

Our formulation of Walther recursion as a first order type discipline is based on regular data types. A regular data type is a set of expressions definable by a grammar. For example, a type N of expressions representing natural numbers can be defined as follows.

$$N ::= \texttt{zero} \mid \texttt{s}(N)$$

The set of lists of natural numbers can be defined as follows.

$$LN ::= \texttt{natnum-nil} \mid \texttt{natnum-cons}(N,\ LN)$$

The set of pure λ-terms can be defined as follows.

$$V ::= \texttt{variable}(N)$$

$$L ::= V \mid \texttt{apply}(L,\ L) \mid \texttt{lambda}(V,\ L)$$

Most first order data types have natural definitions as grammars. Formally, a regular term grammar is a finite set of productions of the form $X \rightarrow u$ where u is a term constructed from nonterminal symbols and constructors where constants are treated as zero-ary constructors. For example, the grammar for natural numbers contains the production $N \rightarrow \texttt{s}(N)$. Each nonterminal of a regular term grammar generates a set of terms in the obvious way. A set of expressions is regular if it is the set generated by some nonterminal of some grammar.[5]

The above grammar includes the productions $L \rightarrow V$, $L \rightarrow \texttt{lambda}(V,\ E)$ and $V \rightarrow \texttt{variable}(N)$. Note that the constructor `variable` only appears once in the grammar even though the nonterminal V occurs several times. Grammars in which every constructor appears at most once have desirable properties.

> **Definition:** A regular term language will be called *monomorphic* if it can be defined by a grammar in which each constructor occurs at most once.

> **Definition:** A *monomorphic normal form* grammar is a grammar in which each constructor occurs at most once; every nonterminal symbol is classified as either an *aggregate nonterminal* or a *constructor nonterminal*; each constructor c is associated with a unique constructor nonterminal X_c; and all productions are either of the form $Y \rightarrow X_c$ or $X_c \rightarrow c(Y_1,\ \ldots,\ Y_n)$ where Y and each Y_i are aggregate nonterminals.

[5] Regular types can also be characterized as the term languages accepted by finite state tree automata [7, 8]. We find the grammar notation clearer. Properties of tree automata can always be stated directly on grammars. More specifically, a tree automaton can be defined to be a set of productions of the form $X \rightarrow c(Y_1,\ \ldots,\ Y_n)$ where each Y_i is a nonterminal symbol. It is not difficult to prove that every grammar can be put in this form. An automaton is top-down deterministic if for every nonterminal X and constructor c there is at most one production of the form $X \rightarrow c(Y_1,\ \ldots,\ Y_n)$. There are regular languages not definable by top-down deterministic grammars. An automaton is bottom-up deterministic if no two productions have the same right hand side. Every regular term language can be defined by a (unique minimal) bottom-up deterministic automaton.

Lemma: Every monomorphic regular term language can be defined by a monomorphic normal form grammar.

When types are defined by a monomorphic normal form grammar it is possible to represent types by finite sets of constructor symbols. Type unions, intersections, differences, and subtype tests can then be computed in the natural way on these finite sets of constructor symbols. Under this scheme an occurrence of a constructor symbol c in a type represents the language generated by a nonterminal X_c in the monomorphic normal form grammar. For distinction constructors c and c' we have that X_c and $X_{c'}$ generate disjoint languages. Hence operations on sets of constructors correspond to the analogous operations on the types being represented. Determining emptiness of an intersection of a set of nonterminals in an arbitrary (nonmonomorphic) regular term grammar is known to be EXPTIME hard [5]. Monomorphic regular types yield a considerable simplification.

Throughout this paper we assume a fixed user declared grammar defining the types of constructors. The user provides a grammar in which no constructor appears more than once and this grammar is automatically converted to a monomorphic normal form grammar with a nonterminal X_c for each constructor c. The term "type" will be used here to mean a finite set σ of constructor symbols representing the monomorphic regular term language generated by the nonterminals X_c for $c \in \sigma$. In practice nonterminals in the user-given grammar can be used to abbreviate the corresponding set of constructor symbols.

A production of the form $X_c \rightarrow c(Y_1, \ldots, Y_n)$ can be viewed as stating a type declaration for the constructor c, namely $c\colon Y_1 \times \ldots \times Y_n \rightarrow X_c$. A monomorphic normal form grammar can be viewed as a set of type declarations where the output types of distinct constructors are disjoint and where each input type is a union of (pairwise disjoint) output types. The monomorphic nature of these type declarations is the source of the phrase "monomorphic grammar".

Since the Hindley-Milner type systems used in ML does not allow subtyping, the type of λ-terms given above can not be represented in ML — in ML the type of variables can not be a proper subset of the type of λ-terms. In ML one might instead represent λ-terms by the following grammar.

$$L ::= \texttt{variable}(N) \mid \texttt{apply}(L,\ L) \mid \texttt{lambda}(N,\ L)$$

But this is technically a different type (a different set of Herbrand terms).

3 A Functional Programming Language

There are three kinds of function symbols in our language — constructors, projection functions, and user-defined functions. Each of these function symbols can be assigned a "function type" of the form $\sigma_1 \times \ldots \times \sigma_n \rightarrow \tau$ where τ and each σ_i are types as defined in the previous section. For a constructor function c we have $c : Y_1 \times \ldots \times Y_n \rightarrow X_c$ where the grammar defining types has the production $X_c \rightarrow c(Y_1, \ldots, Y_n)$. For each constructor of n arguments there are n projection functions of the form $\Pi_{c,i}$ with $1 \leq i \leq n$. The function $\Pi_{c,i}$ extracts the ith argument from an application of c, i.e., we have $\Pi_{c,i}(c(x_1, \ldots, x_n)) = x_i$. For

each projection $\Pi_{c,i}$ we have $\Pi_{c,i} : X_c \to Y_i$ where the grammar contains the production $X_c \to c(Y_1, \ldots, Y_n)$. Note that the projection function $\Pi_{c,i}$ can only be applied to applications of c. The type of a user-defined function is declared as part of its definition.

The set of terms of the language is defined by the following grammar

$$e ::= x \mid f(e_1, \ldots, e_2) \mid \texttt{if}(x{:}c\ e_1\ e_2) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

Note that the test in a conditional expression is of the form $x : c$ where x is a variable and c is a constructor symbol. The test is true if the value of x is an application of c. We will use $\texttt{if}(e_1 : c\ \ e_2\ e_3)$ as an abbreviation for $\texttt{let}\ x = e_1\ \texttt{in}\ \texttt{if}(x : c\ \ e_2\ e_3)$ and use $\texttt{if}(e_1\ e_2\ e_3)$ as an abbreviation for $\texttt{if}(e_1{:}\texttt{true}\ e_2\ e_3)$ where `true` is a constructor constant representing the Boolean value true.

Figure 1 gives rules for assigning types to terms. The rules involve sequents of the form $\rho \vdash e{:}\tau$ where ρ is a mapping from variables to types and τ is a type. The notation $\rho[x := \sigma]$ denotes the mapping from variables to types which is identical to ρ except that it maps x to the type σ. It is important to remember that types are sets of constructor nonterminals. Note that the IF rule types the two branches of the conditional under different type environments. The rules can be be used in a backward-chaining syntax directed way and typability is decidable in linear time (under a fixed monomorphic grammar).

$$\text{VAR}\quad \rho \vdash x : \rho(x)$$

$$\text{IF}\quad \frac{\rho[x := \{c\}] \vdash e_1{:}\sigma \qquad \rho[x := \rho(x) - \{c\}] \vdash e_2{:}\sigma}{\rho \vdash \texttt{if}(x{:}c\ e_1\ e_2){:}\sigma}$$

$$\text{LET}\quad \frac{\rho \vdash e_1{:}\sigma \qquad \rho[x := \sigma] \vdash e_2{:}\tau}{\rho \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2){:}\tau}$$

$$\text{SUB}\quad \frac{\rho \vdash e{:}\tau \qquad \tau \subseteq \sigma}{\rho \vdash e{:}\sigma}$$

$$\text{APP}\quad \frac{f : \sigma_1 \times \ldots \sigma_n \to \tau \qquad \rho \vdash e_1{:}\sigma_1 \qquad \vdots \qquad \rho \vdash e_n{:}\sigma_n}{\rho \vdash f(e_1, \ldots, e_n){:}\tau}$$

Fig. 1. The type inference rules for terms.

A *program* is a sequence of definitions of the form

$$F(x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n){:}\tau \equiv B$$

where F does not appear previously in the sequence; σ_i and τ are types; B is a term with no free variables other than $x_1, \ldots, x_n$; every defined function symbol in B is either F or is defined earlier in the sequence (mutual recursion is not allowed); and $\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash B{:}\tau$.

For example we can define addition on natural numbers as follows.

$$N ::= \texttt{zero} \mid \texttt{s}(N)$$

$$\texttt{PRED}(x{:}\texttt{s}){:}\, N \equiv \Pi_{s,1}(x)$$

$$\texttt{PLUS}(x{:}\, N,\ y{:}\, N){:}\, N \equiv \texttt{if}(x{:}\texttt{zero}\ \ y\ \texttt{s}(\texttt{PLUS}(\texttt{PRED}(x),\ y)))$$

Note that PRED can only be applied to applications of the constructor s and that x is declared to be of type N which is $\{X_{\texttt{s}},\ X_{\texttt{zero}}\}$. However, the IF rule checks the second branch of the conditional under a type environment in which x has type $\{X_{\texttt{s}}\}$.

4 Reducer-Conserver Analysis

Reducer-conserver analysis derives information of the form $u < x$ where x is a variable occurring in u. The analysis automatically generates and uses reducer and conserver lemmas of the form $f(x_1,\ \ldots,\ x_n) < x_i$ and $f(x_1,\ \ldots,\ x_n) \leq x_i$ respectively. For example, we would like to derive $\texttt{DIFF}(\texttt{CDR}(x),\ y) < x$ from the fact that CDR reduces and DIFF (set difference) conserves its first argument.

$N := \texttt{zero} \mid \texttt{s}(N)$

$\texttt{PRED}(x{:}\, s){:}\, N \equiv \Pi_{\texttt{s},1}(x)$

$\texttt{MINUS}(x{:}\, N,\ y{:}\, N){:}\, N \equiv \texttt{if}(y{:}\texttt{zero}\ \ x\ \texttt{if}(x{:}\texttt{zero}\ \ x\ \texttt{MINUS}(\texttt{PRED}(x),\ \texttt{PRED}(y))))$

$\texttt{RMINUS}(x{:}\, s,\ y{:}\, s){:}\, N \equiv \texttt{MINUS}(\texttt{PRED}(x),\ \texttt{PRED}(y))$

$B ::= \texttt{true} \mid \texttt{false}$

$\texttt{NOT}(x{:}\, B){:}\, B \equiv \texttt{if}(x{:}\texttt{true}\ \ \texttt{false}\ \texttt{true})$

$\texttt{ZERO?}(x{:}\, N){:}\, B \equiv \texttt{if}(x{:}\texttt{zero}\ \ \texttt{true}\ \texttt{false})$

```
LESS?(x: N, y: N): B ≡ if(x:zero
                          NOT(ZERO?(y))
                          if(y:zero
                             false
                             LESS?(PRED(x), PRED(y))))
```

```
GCD(x: N, y: N): N ≡ if(x:zero
                        y
                        if(y:zero
                           x
                           if(LESS?(x, y):true
                              GCD(x, RMINUS(y, x))
                              GCD(RMINUS(x, y), y))))
```

Fig. 2. A Walther Recursive Definition of GCD

Figure 2 gives a Walther recursive definition of GCD including all type definitions (grammars) and auxiliary functions. Under the definitions in the figure we have that PRED reduces its argument; MINUS conserves its first argument and RMINUS (restricted MINUS) reduces its first argument. To simplify the analysis we classify functions as reducers or conservers relative to any legal input. This creates a need for two definitions of minus — one that allows its arguments to be zero and hence is not guaranteed to reduce, and one that requires both arguments to be nonzero.

The simple classification of functions as reducers or conservers over all legal inputs is a significant simplification of Walther's original calculus. Walther uses a single function MINUS and creates a conditional lemma stating that MINUS reduces its first argument in the case where both arguments are nonzero. Here we avoid such conditionals because we want the calculus to be simple, i.e. "syntactic". Programmers should have no difficulty in determining by inspection if a given definition is acceptable. The automatic construction of conditional lemmas makes analysis by inspection more difficult. However, by forcing the programmer to use a restricted version of subtraction when using subtraction as a reducer, it is clear that we are imposing a "discipline" on the programmer. Unlike the discipline of primitive recursion however, the discipline of Walther recursion leaves us with a version of GCD which is still easy to formally verify — the formal proof that GCD(x, y) divides both x and y is significantly simpler under this Walther recursive version of GCD than under a primitive recursive version.

All of the recursive definitions in figure 2 terminate because they recurse on smaller arguments. In the case of MINUS and LESS? the reduction is done by PRED. In the case of GCD the reduction is done by RMINUS. Also, in the case of GCD we are using both arguments as measure arguments — each recursive call preserves both measure arguments and reduces at least one.

There are two analyses which must done to support Walther recursion (in addition to the type analysis of the previous section). First, we will define an analysis which automatically generates lemmas of the form $u < x$ from previously generated reducer and conserver lemmas for individual functions. Second, we will define an analysis which automatically generates reducer-conserver lemmas for each new definition. Both analyses are simple enough to be performed by inspection.

Figure 3 gives inference rules defining the first analysis. To simplify the rules, and to increase the precision of the analysis, we use assertions of the form $u \leq \Pi_{c,i}(x)$ rather than strict inequalities of the form $u < v$. These rules use as input reducer and conserver lemmas which have been automatically derived by syntactic analysis of earlier definitions. These lemmas appear in the rules inside square brackets. The rules derive assertions of the form $u \leq \Pi_{c_1,i_1}(\ldots \Pi_{c_n i_n}(x)\ldots)$ where x is a variable appearing in u. The rules make use of an index set R_c for each constructor c and the strictness rule STR and the reduction rule RED1 are restricted to "linear" constructors as we now explain.

The appropriate measure of size for lists of numbers is the length of the list.

REFL $\quad x \leq x$

STR $\quad \dfrac{u \leq \Pi_{c,i}(t) \quad i \in R_c, \;\; c \text{ linear}}{u \leq t}$

RED1 $\quad \dfrac{u \leq t \quad [f(\ldots x_j \ldots) \leq \Pi_{c,i}(x_j)] \quad i \in R_c, \;\; c \text{ linear}}{f(\ldots u \ldots) \leq \Pi_{c,i}(t)}$

RED2 $\quad \dfrac{[f(\ldots x_j \ldots) \leq \Pi_{c,i}(x_j)]}{f(\ldots u \ldots) \leq \Pi_{c,i}(u)}$

CONS1 $\quad \dfrac{u \leq t \quad [f(\ldots x_i \ldots) \leq x_i]}{f(\ldots u \ldots) \leq t}$

CONS2 $\quad \dfrac{\forall i \in R_c \;\; u_i \leq \Pi_{c,i}(t)}{c(u_1, \ldots, u_n) \leq t}$

IF $\quad \dfrac{e_1 \leq t \quad e_2 \leq t}{\texttt{if}(x{:}c \;\; e_1 \; e_2) \leq t}$

LET $\quad \dfrac{e_2[e_1/x] \leq t}{(\texttt{let } x = e_1 \texttt{ in } e_2) \leq t}$

Fig. 3. Rules for reducer-conserver analysis.

If numbers are represented as Herbrand terms then taking the weight of a term to be its total size, in say number of syntax nodes, would give an inappropriate measure of size for lists — one not well suited to proving, for example, termination of sorting functions. Following Walther we get a more desirable measure of size by associating each constructor c with an index set R_c and define the weight of an expression by the following equation.

$$w(c(u_1, \ldots, u_n)) = 1 + \sum_{i \in R_c} w(u_i)$$

Again following Walther, we adopt a heuristic for constructing the index set R_c automatically from the grammar. For each projection function $\Pi_{c,i}$ we take $\tau_{c,i}$ to be the output type of $\Pi_{c,i}$ so that we have $\Pi_{c,i} : \{c\} \rightarrow \tau_{c,i}$. Now we define R_c to be the set of indices i such that $c \in \tau_{c,i}$. Consider a list constructor cons used to construct lists of numbers. If u is such a list then cons can not appear as the top level constructor of the first element of u, i.e., $\texttt{cons} \notin \tau_{\texttt{cons},1}$. So we have $R_{\texttt{cons}} = \{2\}$ and the weight of a cons cell is one plus the weight of its cdr.

A constructor c is called a "base constructor" if R_c is empty. The weight of any application of a base constructor is 1. All constants are base constructors but it is sometimes useful to also have base constructors which take arguments. A constructor c will be called "linear" if $|R_c| = 1$ and for $i \in R_c$ we have that c is the only non-base constructor in $\tau_{c,i}$ (the output type of $\Pi_{c,i}$). The list constructor cons for building lists of numbers is a linear constructor. Tree constructors are not linear. An application of a linear constructor can always be thought of as a sequence and the weight of the term is one greater than its length.

As an example we consider the derivation of $\mathtt{SORT}(\mathtt{CDR}(x)) \leq \Pi_{\mathtt{cons},2}(x)$ from the lemmas $[\mathtt{CDR}(y) \leq \Pi_{\mathtt{cons},2}(y)]$ and $[\mathtt{SORT}(y) \leq y]$. First the reflexive rule (REFL) is used to derive $x \leq x$. Then the reducer rule (RED2) applies to the reducer lemma for CDR to derive $\mathtt{CDR}(y) \leq \Pi_{\mathtt{cons},2}(y)$. Finally, the first conserver rule (CONS1) applies the conserver lemma for SORT to derive $\mathtt{SORT}(\mathtt{CDR}(x)) \leq \Pi_{\mathtt{cons},2}(x)$.

The rule system is decidable — one can determine where $u \leq t$ is provable by enumerating all provable assertions of the form $v \leq t'$ where v is a subterm of u. Expansion of let expressions can cause an exponential growth in the printed length of an expression but it does not increase the number of distinct subterms. Hence the let rule does not introduce any significant inefficiency.

$$
\begin{array}{ll}
L & ::= V \mid \mathtt{apply}(L,\ L) \mid \mathtt{lambda}(V,\ L) \\
V & ::= \mathtt{variable}(LL) \\
LL & ::= \mathtt{lnil} \mid \mathtt{lcons}(L,\ LL)
\end{array}
$$

$$
\begin{array}{l}
\mathtt{RENAME}(x{:}V,\ y{:}V,\ w{:}L){:}L, \\
\equiv \mathtt{if}(w{:}\mathtt{variable} \\
\quad \mathtt{if}(\mathtt{EQUAL}(w,\ x){:}\mathtt{true}\ \mathtt{variable}(\Pi_{\mathtt{variable},1}(y))\ w) \\
\quad \mathtt{if}(w{:}\mathtt{apply} \\
\quad\quad \mathtt{apply}(\mathtt{RENAME}(x,\ y,\ \Pi_{\mathtt{apply},1}(w)),\ \mathtt{RENAME}(x,\ y,\ \Pi_{\mathtt{apply},2}(w)) \\
\quad\quad \mathtt{let\ newvar} = \mathtt{variable}(\mathtt{lcons}(y,\ \mathtt{lcons}(w,\ \mathtt{lnil}))) \\
\quad\quad\quad \mathtt{in\ lambda}(\mathtt{newvar} \\
\quad\quad\quad\quad\quad \mathtt{RENAME}(x,\ y,\ \mathtt{RENAME}(\Pi_{\mathtt{lambda},1}(w), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{newvar} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \Pi_{\mathtt{lambda},2}(w)))
\end{array}
$$

$$
\begin{array}{l}
\mathtt{SUBST}(x{:}L,\ y{:}V,\ w{:}L){:}L, \\
\equiv \mathtt{if}(w{:}\mathtt{variable} \\
\quad \mathtt{if}(\mathtt{EQUAL}(w,\ y){:}\mathtt{true}\ x\ w) \\
\quad \mathtt{if}(w{:}\mathtt{apply} \\
\quad\quad \mathtt{apply}(\mathtt{SUBST}(x,\ y,\ \Pi_{\mathtt{apply},1}(w)),\ \mathtt{SUBST}(x,\ y,\ \Pi_{\mathtt{apply},2}(w)) \\
\quad\quad \mathtt{let\ newvar} = \mathtt{variable}(\mathtt{lcons}(x,\ \mathtt{lcons}(w,\ \mathtt{lnil}))) \\
\quad\quad\quad \mathtt{in\ lambda}(\mathtt{newvar} \\
\quad\quad\quad\quad\quad \mathtt{SUBST}(x,\ y,\ \mathtt{RENAME}(\Pi_{\mathtt{lambda},1}(w), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{newvar} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \Pi_{\mathtt{lambda},2}(w)))
\end{array}
$$

Fig. 4. Substitution in the λ-calculus.

Figure 4 gives a Walther-recursive definition of substitution in the λ-calculus. The renaming of bound variables to avoid capture prevents the definition from being primitive recursive. Intuitively, the definitions in figure 4 are acceptable because RENAME conserves its last argument — the result of renaming is the same

size as the original. Reducer and conserver properties of functions are actually partial correctness assertions — they state that if the function F terminates then the value is bounded in a certain way. They are derived *before* termination analysis is performed and hence they can be used as part of the termination analysis. Given that RENAME conserves its third argument, reducer-conserver analysis can be used to prove that the third argument is reduced on every recursive call. A similar analysis holds in the definition of SUBST. The rule CONS2 is needed to verify the conservation property of RENAME in the (awkwardly written) base case.

The soundness of the rules STR and RED1 requires explanation. The difficult part is ensuring that the term $\Pi_{c,i}(t)$ in the conclusion of RED1 is well typed. This is done by defining a careful semantics for the assertions of the form $u \leq t$ generated by the rules. The assertion $u \leq t$ is taken to mean the conjunction of the following properties.

- If u is well typed then t well typed.
- If u is well typed then the top level constructor of the value of u is either a base constructor or the same as the top level constructor of the value of t.
- If u is well typed then the weight of the value of u is no larger than the weight of the value of t.

Now consider the rule RED1. We must show that the conclusion $f(\ldots u \ldots) \leq \Pi_{c,i}(t)$ satisfies all of the above conditions under the assumption that the antecedents do. To check the first condition suppose that $f(\ldots u \ldots)$ is well typed. We can assume that all instances of the reduction lemma satisfy this same invariant. Hence $\Pi_{c,i}(u)$ is well typed. This implies that the top level constructor of u is c. But since $i \in R_c$ we have that c is not a base constructor. Hence, the top level constructor of t must also be c. In which case $\Pi_{c,i}(t)$ is well typed. The second invariant follows from the assumption that it holds for all instances of the reduction lemma in the premise. The final invariant follows from the fact that c is linear and hence that $\Pi_{c,i}$ reduces weight by exactly 1.

Now consider the rule STR. The problematic invariant for this rule is the second one — that when u is well typed either the top level constructor of u is a base constructor or is the same as the top level constructor of t. Assume that u is well typed and is an application of a non-base constructor c'. From the first antecedent we have that the value of $\Pi_{c,i}(t)$ must also be an application of c'. But also from the first antecedent we have that $\Pi_{c,i}(t)$ is well typed and hence t is an application of c. But since c is linear we must have that c' is c. Hence both u and t are application of c and the invariant is satisfied in the conclusion. We leave it to the reader to verify the soundness of the other rules relative to the above semantics.

The second part of reducer-conserver analysis involves automatically generating new reducer and conserver lemmas for each newly defined function and checking termination of that function. Initially we have all reducer lemmas of the form $[\Pi_{c,i}(x) \leq \Pi_{c,i}(x)]$ where $i \in R_c$. Now consider a new definition

$$F(x_1{:}\sigma_1,\ \ldots,\ x_n{:}\sigma_n){:}\tau \equiv B$$

We construct all "self-supporting" conserver and reducer lemmas of the form $[F(x_1, \ldots, x_n) \leq x_i]$ and $[F(x_1, \ldots, x_n) \leq \Pi_{c,j}(x_i)]$ where $[F(x_1, \ldots, x_n) \leq t]$ is self-supporting if adding it to the list of reducer and conserver lemmas allows a derivation of $B \leq t$ (where B is the definition of F). After deriving new reducer and conserver lemmas we check termination. First we compute the set of arguments x_i such that for each recursive call $F(u_1, \ldots, u_n)$ in (the let expansion of) B we can derive either $u_i \leq x_i$ or $u_i \leq \Pi_{c,j}(x_i)$ for some projection $\Pi_{c,j}$.[6] Take this set of arguments to be the set of measure arguments for F. If there are no measure arguments then termination analysis fails. If there are measure arguments then we check that at each recursive call $F(u_1, \ldots, u_n)$ in (the let expansion of) B there is some measure argument x_i such that we can derive $u_i \leq \Pi_{c_1,j_1}(\ldots\Pi_{c_n,j_n}(x_i)\ldots)$ where we have at least one projection function applied to x_i on the right hand side. If at each recursive call at least one measure argument decreases in this way, then the definition terminates and is acceptable.

5 Some Sorting Examples

We now give a few sorting examples. The three definitions given here are essentially transcriptions of examples given in [9]. Unlike the termination test given there, however, the termination test given here is essentially syntactic. Some of the examples given in [9] fail the syntactic test described here. We start with a grammar for lists of numbers.

N ::= zero | s(N)
NL ::= nnil | ncons(N, NL)
HEAD(x:ncons): $N \equiv \Pi_{\mathrm{ncons},1}(x)$
TAIL(x:ncons): $NL \equiv \Pi_{\mathrm{ncons},2}(x)$

As discussed in the previous section, the analysis takes the weight of a list to be its length rather than the sum of the weight of its elements. Our first example is a Walther recursive version of selection sort. Each example is accompanied by relevant reducer and conserver lemmas that have been automatically generated from the definitions of auxiliary functions.

```
SSORT(x:NL): NL  ≡  if(x:nnil,
                       nnil
                       ncons(MIN(x), SSORT(REPLACE(MIN(x), HEAD(x), TAIL(x)))))
[REPLACE(x, y, z) ≤ z]
```

The definitions of MIN and REPLACE are not given but they are simple primitive recursions. The definition of SSORT shows that some creativity may be needed in constructing Walther recursive definitions. A more natural definition would use the recursive call SSORT(REMOVE(MIN(x), x)). Unfortunately REMOVE is not a reducer in general. Furthermore, it is not possible to express appropriate

[6] Because of the restrictions on the rule STR it is possible that for complex grammars we can derive $u_i \leq \Pi_{c,j}(x_i)$ but not $u_i \leq x_i$.

type restrictions on a restricted version of REMOVE (which might be analogous to RMINUS) because the type system corresponding to a monomorphic grammar does not support dependent types. However, by using REPLACE rather than REMOVE we arrive at a Walther recursive definition.

The next example is a version of quicksort. SMALLER-MEMBERS(x, y) returns the list of members of y which are smaller than the number x. DIFF(x, y) returns the list of members of x that are not members of y. Both of these functions are conservers.

```
QSORT(x:NL):NL  ≡  if(x:nnil,
                      nnil
                      APPEND(QSORT(SMALLER-MEMBERS(HEAD(x), TAIL(x))),
                             ncons(HEAD(x),
                                   QSORT(DIFF(TAIL(x),
                                              SMALLER-MEMBERS(HEAD(x), TAIL(x))))))
```

[SMALLER-MEMBERS(x, y) $\leq y$], [DIFF(x, y) $\leq x$]

The final example is a version of mergesort. The function EVEN takes a list and returns the set of elements which occur at even positions in that list (where the first element occurs at an odd position). The function EVEN is restricted so that it only applies to nonempty lists. Under this restriction, EVEN is a reducer. Note that the rule CONS2 is required to show that the the second recursive call involves a smaller argument.

```
MSORT(x:NL):NL  ≡  if(x:nnil,
                      nnil,
                      if(TAIL(x):nnil,
                         x,
                         MERGE(MSORT(EVEN(x)),
                               MSORT(ncons(HEAD(x), EVEN(TAIL(x)))))))
```

[EVEN(x) $\leq \Pi_{\mathrm{ncons},2}(x)$]

6 Discussion

Walther recursion generalizes primitive recursion and allows more natural, and hence more easily verifiable, definitions of functions such as GCD. Walther recursion is not a panacea for termination analysis however. There are many natural definitions of GCD and SORT which are not Walther recursive. However, Walther recursion seems sufficiently simple that one can learn to write in a Walther recursive style. The situation is similar to, but somewhat less restrictive than, primitive recursion.

Walther recursion does not introduce any new semantic functions — it only allows for more natural definitions of functions which already have (unnatural) primitive recursive definitions. However, it seems that syntactic liberalizations of primitive recursion have a role in verification systems. Furthermore, Walther recursion seems like a particularly natural and possibly distinguished liberalization.

One weakness of Walther recursion seems to be the absence of polymorphism or dependent types in the associated type system. It seems likely that the monomorphic grammars used to define types could be replaced by polymorphic type declarations for data constructors. It might also be possible to construct a purely syntactic treatment of "semantic" dependent types such as `MEMBER-OF`(x) where x is a list or `LIST-CONTAINING`(x) where x is a number. This might provide a syntactic notion of terminating recursion which would handle the natural definition of selection sort.

We are optimistic that sophisticated syntactic analysis methods will improve the effectiveness of termination verification and of verification more generally construed.

References

1. A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languges*, pages 163–173. Association for Computing Machinery, 1994.
2. Robert S. Boyer and J Struther Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.
3. N. Dershowitz and J.P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. MIT Press, 1990.
4. Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, August 1979.
5. Thom Frühwirth, Ehud Shapiro, Moshe Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 75–83. IEEE Computer Society Press, 1991.
6. N. Heintze. Set based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.
7. M. O. Rabin. Decidability of second order theories and automata on infinite trees. *Trans. of Amer. Math. Soc.*, 141:1–35, 1969.
8. W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics*, pages 133–164. MIT Press, 1990.
9. Cristoph Walther. On proving termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.
10. webmaster@cli.com. Home page for computational logic incorporated. http://www.cli.com/index.html.
11. Javier Thayer, William Farmer, Joshua Guttman. Imps: An interactive mathematical proof system. In *CADE-10*, pages 653–654. Springer-Verlag, 1990.

This article was processed using the LaTeX macro package with LLNCS style