1991

J. Heering

Implementing higher-order algebraic specifications

Computer Science/Department of Software Technology Report CS-R9150 December

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Implementing Higher-Order Algebraic Specifications

J. Heering
Department of Software Technology, CWI,
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Writing algebraic specifications that are to be executed as rewrite systems is similar to functional programming. There are some differences, however. Algebraic specification languages allow left-hand sides of equations to be complex first-order patterns that would not be allowed in functional languages. Functional languages, on the other hand, have powerful higher-order features not offered by algebraic specification languages. Some functional languages combine higher-order functions with linear first-order patterns involving free data type constructors, thus offering a limited (but highly expressive) mixture of functional programming and algebraic specification. A more ambitious integration of the two is obtained by allowing both signatures and equations in algebraic specifications to be higher-order. Operational experiments with such higher-order algebraic specifications can be performed by translating them to λProlog, an extension of Prolog to polymorphically typed λ-terms based on higher-order unification.

Key Words & Phrases: higher-order algebraic specification, integration of algebraic specification and functional programming, higher-order term rewriting, higher-order matching, λProlog.

1991 CR Categories: D.1.1 [Programming techniques]: Applicative (functional) programming; D.2.1 [Software engineering]: Requirements/specifications - *Languages;* D.3.3 [Programming languages]: Language constructs and features - *Abstract data types;* F.3.2 [Logics and meanings of programs]: Semantics of programming languages - *Algebraic approaches to semantics;* F.4.2. [Mathematical logic and formal languages]: Grammars and other rewriting systems.

1991 Mathematics Subject Classification: 68N17 [Software]: Logic programming; 68Q42 [Theory of computing]: Rewriting systems; 68Q65 [Theory of computing]: Abstract data types; algebraic specification.

Note: Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II - GIPE II).

1. INTRODUCTION

1.1. Higher-order algebraic specifications

Conventional algebraic data type specifications consist of a first-order signature and a set of equations. Equations may contain first-order variables, which are implicitly or explicitly universally quantified. The signature defines the abstract syntax of a language of terms whose semantics is given by the equations. Such specifications are usually implemented by interpreting them as (first-order) term rewriting systems [Klo90]. Each equation is interpreted as a left-to-right rewrite rule and the resulting rewrite system is used to evaluate terms by reducing them to normal form (if any). The annoying fact that this asymmetric interpretation of inherently symmetric equations may lead to rewrite systems that are incomplete with respect to equational deduction from the original specification does not concern us here.

Writing algebraic specifications that are to be executed as rewrite systems is similar to functional programming. There are some differences, however. Algebraic specification languages allow left-hand sides of equations to be complex first-order patterns that would not be allowed in functional languages. Functional languages, on the other hand, have powerful higher-order features not offered by algebraic specification languages.

Report CS-R9150
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Some functional languages (e.g., Hope [BMS80, Bai90]) combine higher-order functions with linear first-order patterns involving free data type constructors, thus offering a limited (but highly expressive) mixture of functional programming and algebraic specification. A more ambitious integration of the two is obtained by allowing both signatures and equations in algebraic specifications to be higher-order. The higher-order signature defines the abstract syntax of a language of typed λ-terms whose semantics is given by the equations.

Recently, development and implementation of higher-order algebraic specification languages was advocated by Jouannaud and Okada [JO91] and, having frequently felt the need for higher-order equations in algebraic specifications ourselves, we thought it would be interesting to be able to perform operational experiments with them. Higher-order term rewriting requires, first of all, higher-order matching, which is the special case of higher-order unification in which one of the terms involved does not contain free variables. Two readily available systems incorporating higher-order unification are λProlog [NM88], an extension of Prolog to typed λ-terms, and the generic theorem prover Isabelle [PN90]. Since we had some experience with schemes for translating first-order algebraic specifications to Prolog (see the surveys by Drosten [Dro88] and Bouma and Walters [BW89]), we chose λProlog as our target system.

It would be nice if the notion of initial algebra specification, which has unequivocal meaning in the first-order case [MG85], had an equally unequivocal higher-order analogue. This does not seem to be the case, however, since it depends on the precise notion of higher-order model one prefers. Meinke, for instance, assumes models to be extensional higher-order algebras and shows that in this setting higher-order initial algebra specification is strictly more powerful than its first-order counterpart [Mei90, Mei91]. Poigné, on the other hand, considers both extensional and intensional models [Poi86]. Although these questions are beyond our present scope, we shall briefly return to them since the precise notion of initial algebra semantics adopted affects the degree of incompleteness of our implementation scheme.

1.2. Higher-order term rewriting

Higher-order term rewriting, the mechanism we use to execute higher-order algebraic specifications, is more powerful, but also considerably less manageable than its first-order counterpart. The following examples illustrate some of its possibilities and problems.

(I) Consider the signature

sorts s, *bool*

functions $a: s$
$f,g: s \to s$
$if: bool \times s \times s \to s$

variables $X,Y: s$
$F: s \to s$ (second-order variable)
$B,B': bool$

and the second-order equation

$$if(B,F(X),F(Y)) = F(if(B,X,Y)) \qquad (1)$$

(cf. Section 3.3 of [Hee86]). The left-hand side of (1) matches

$$if(B',g(f(a)),g(f(f(a))))$$

in three different ways, namely, for

$F=\lambda V.g(f(V))$	$X=a$	$Y=f(a)$	$B=B'$
$F=\lambda V.g(V)$	$X=f(a)$	$Y=f(f(a))$	$B=B'$
$F=\lambda V.V$	$X=g(f(a))$	$Y=g(f(f(a)))$	$B=B'$.

Thus, whereas a first-order match has at most a single solution, a higher-order match may have many. It may even have solutions that leave some of the variables in the left-hand side of the rewrite rule uninstantiated, something that cannot happen in the first-order case either. For instance, the left-hand side of (1) matches

$if(B',a,a)$

for

$F=\lambda V.a$	$X=X$	$Y=Y$	$B=B'$
$F=\lambda V.V$	$X=a$	$Y=a$	$B=B'$.

The first solution leaves X and Y uninstantiated. If (1) is interpreted as a left-to-right rewrite rule, this is no problem since both variables are eliminated by β-reduction after substitution of the solution in the right-hand side:

$$if(B',a,a) \overset{(1)}{\rightarrow} (\lambda V.a)(if(B',X,Y)) \overset{(\beta)}{\rightarrow} a.$$

A solution instantiating F to $\lambda V.V$ exists for any *if*-term and is —at least in this case—algebraically harmless. The danger of non-termination it entails can be averted by adopting a parallel reduction strategy treating all solutions on an equal basis, or by a simple loop check. For reasons of efficiency we have chosen the latter alternative.

(II) Consider the second-order equation

$$cons(X,G(cons(X,L))) = cons(X,G(L)) \qquad (2)$$

with the signature from example (I) plus the additional declarations

sort *lst*

functions $cons: s \times lst \rightarrow lst$
$nil: lst$

variables $L: lst$
$G: lst \rightarrow lst$ (second-order variable).

If interpreted as a rewrite rule, equation (2) deletes the rightmost element of a pair of identical list elements. For instance, its left-hand side matches the list

$cons(a,cons(f(a),cons(a,nil)))$

in two ways, namely, for

$X=a$	$G=\lambda V.cons(f(a),V)$	$L=nil$, and
$X=a$	$G=\lambda V.cons(f(a),cons(a,nil))$	$L=L$.

Substitution of the first solution in the right-hand side of (2) yields

$$cons(a,(\lambda V.cons(f(a),V))(nil)) \overset{(\beta)}{\rightarrow} cons(a,cons(f(a),nil)).$$

The second solution is algebraically harmless, but useless from an operational viewpoint. In fact, the left-hand side of (2) matches any list of the form $cons(x,l)$ for

$X=x$	$G=\lambda V.l$	$L=L$

whether x occurs in l or not, yielding

$$cons(x,l) \overset{(2)}{\rightarrow} cons(x,(\lambda V.l)(L)) \overset{(\beta)}{\rightarrow} cons(x,l).$$

The danger of non-termination can be averted in the same way as before, but one clearly has to check all matches of a higher-order rule carefully. For instance, deleting the leftmost element of a pair of identical list elements by means of

$$cons(X,G(cons(X,L))) = G(cons(X,L)) \qquad (3)$$

(which has the same left-hand side as (2)) leads to

$$cons(x,l) \overset{(3)}{\rightarrow} (\lambda V.l)(cons(x,L)) \overset{(\beta)}{\rightarrow} l.$$

This is incorrect since l need not contain x. The simpler equation

$$cons(X,H(X)) = H(X) \tag{4}$$

with H a second-order variable of type $s \rightarrow lst$ has the same problem.

(III) Although it did not happen in examples (I) and (II), variables in the left-hand side of a higher-order rewrite rule that are left uninstantiated after matching may enter the reduct. We borrow the following example from Nipkow's paper on higher-order critical pairs [Nip91]. The rule

$$f(g(F(X),F(a))) \rightarrow f(X) \tag{5}$$

can be applied to the term $f(g(a,a))$ in two ways, one of which instantiates F to $\lambda V.a$ and leaves X uninstantiated, thus yielding the result $f(X)$.
To get rid of this problem and to eliminate ambiguous rules such as (3) and (4), Nipkow restricts left-hand sides of rules to so-called *patterns*. A pattern is a term in β-normal form such that each free variable F occurring in it is applied only to terms that are η-equivalent to distinct bound variables. Unfortunately, this restriction also rules out many useful equations such as (1) and (2), both of which have left-hand sides containing free variables whose arguments again contain free variables. Clearly, as Nipkow himself points out, more general left-hand sides should be allowed. Since equations (2) and (3) have the same left-hand side, a more liberal restriction that accepts (2) but rejects (3) will have to take both sides of equations into account. See also Section 4.4 of [JO91].
We do not impose any a priori restriction, but equations that may cause uninstantiated variables to be introduced in the reduct are not necessarily treated correctly by our λProlog code and should be avoided.

(IV) Whereas first-order term rewriting requires subterm matching, higher-order rewriting can do without explicit subterm lookup if each equation $t_1 = t_2$ is extended to $H(t_1) = H(t_2)$ with H a polymorphic higher-order variable not free in t_1 or t_2. In this case, higher-order matching of the extended left-hand side with the full input term performs the subterm lookup implicitly. Like before, useless instantiations of H to $\lambda X.s$, where s does not contain X, can be rejected by a simple loop check. This approach is used in Section 2.

1.3. λProlog

λProlog is an extension of Prolog to typed λ-terms [NM88]. Basically, the functions declared in a λProlog program generate a domain of polymorphically typed λ-terms, and polymorphic higher-order unification takes the place of first-order unification in the proof procedure.

Since λ-terms may be subject to α-, β-, and η-reduction, the term domain underlying a λProlog program is not purely syntactic. Furthermore, unlike first-order unification, higher-order unification is neither decidable nor unitary. As a consequence, in λProlog backtracking to an alternative unifier of the same pair of terms may occur and the search for a higher-order unifier may go on forever.

Higher-order matching, the special case of higher-order unification we need, was conjectured to be decidable in the simply typed case (no polymorphism) by Huet [Hue76], but this is still an open problem. The third-order case was recently shown to be decidable by Dowek [Dow91a]. On the other hand, Dowek also showed that strongly polymorphic higher-order matching is undecidable [Dow91b]. λProlog supports ML-style polymorphism, so we included it in our notion of higher-order algebraic specification as well. As far as we know, the "intermediate" case of higher-order matching in combination with ML-style polymorphism has not yet been settled, so it may still turn out to be decidable. In the version of λProlog we used* the implementation of polymorphic higher-order unification was incomplete and this caused some problems. These will be explained in due course. The examples in Section 1.2 show that higher-order matches with multiple solutions, none of them subsumed by any of the other ones, are no exception. In our λProlog code, backtracking to an alternative solution may occur as a result of loop checking.

This rudimentary knowledge of λProlog in combination with a basic understanding of Prolog (see, for instance, [Bra86]) suffices to understand the next section.

---

* Version 2.7 (October 1988). It was obtained by anonymous *ftp* from *duke.cs.duke.edu*.

2. TRANSLATING HIGHER-ORDER ALGEBRAIC SPECIFICATIONS TO λPROLOG

2.1. A very simple scheme

Consider the following higher-order algebraic specification:

module *N*
begin
sorts *nat*, *bool*, *lst*(*A*)

functions
$zero: nat$
$succ: nat \rightarrow nat$
$add: nat \times nat \rightarrow nat$
$t, f: bool$
$if: bool \times A \times A \rightarrow A$
$nil: lst(A)$
$cons: A \times lst(A) \rightarrow lst(A)$
$map: (A \rightarrow B) \times lst(A) \rightarrow lst(B)$
$compose: (B \rightarrow C) \times (A \rightarrow B) \rightarrow A \rightarrow C$

equations

$$add(X,zero) = X \quad (6)$$
$$add(X,succ(Y)) = succ(add(X,Y)) \quad (7)$$
$$if(t,X,Y) = X \quad (8)$$
$$if(f,X,Y) = Y \quad (9)$$
$$if(B,F(X),F(Y)) = F(if(B,X,Y)) \quad (10)$$
$$cons(X,F(cons(X,L))) = cons(X,F(L)) \quad (11)$$
$$map(F,nil) = nil \quad (12)$$
$$map(F,cons(X,L)) = cons(F(X),map(F,L)) \quad (13)$$
$$compose(F,G) = \lambda X.F(G(X)) \quad (14)$$

end *N*.

Identifiers whose first character is a capital letter are variables. Their type is not declared explicitly (although it might have been), but is determined by the context in which they occur. For instance, *X* has type *nat* in (6), but polymorphic type *A* (with *A* a type variable) in (8).

In addition to the two carriers corresponding to sorts *nat* and *bool*, the higher-order initial algebra of *N* has an infinite number of first-order carriers corresponding to *lst*(τ) for any monotype τ. In particular, τ may be a functional monotype such as *nat* → *nat* or another *lst*-monotype. The higher-order carriers (function spaces) of the initial algebra consist of the appropriately typed functions definable in terms of the signature of *N*.

Equations (10) and (11) are polymorphic versions of (1) and (2) in Section 1.2. Because the structure of their left-hand side is too complicated, neither would have been allowed in a first-order algebraic specification or a functional program. Equations (12)-(14), on the other hand, could have been written in virtually the same way in Hope (see Chapter 6 of [Bai90]). Note, however, that in view of equation (11) *cons* is not a free constructor.

Using the scheme outlined in example (IV) of Section 1.2, we translate *N* to the following λProlog module:

```
module lpN.

kind nat      type.
kind bool     type.
kind lst      type -> type.

type zero     nat.
type succ     nat -> nat.
```

```
type add      nat -> nat -> nat.
type t        bool.
type f        bool.
type if       bool -> A -> A -> A.
type nil      (lst A).
type cons     A -> (lst A) -> (lst A).
type map      (A -> B) -> (lst A) -> (lst B).
type compose  (B -> C) -> (A -> B) -> A -> C.

type reduce   A -> A -> o.
type extrule  A -> A -> o.

extrule (H (add X zero))            (H X).                          %%% (6')
extrule (H (add X (succ Y)))        (H (succ (add X Y))).           %%% (7')
extrule (H (if t X Y))              (H X).                          %%% (8')
extrule (H (if f X Y))              (H Y).                          %%% (9')
extrule (H (if B (F X) (F Y)))      (H (F (if B X Y))).             %%% (10')
extrule (H (cons X (F (cons X L)))) (H (cons X (F L))).             %%% (11')
extrule (H (map F nil))             (H nil).                        %%% (12')
extrule (H (map F (cons X L)))      (H (cons (F X) (map F L))).%%% (13')
extrule (H (compose F G))           (H (X \ (F (G X)))).            %%% (14')

reduce X Y :- extrule X Z,
              not(X = Z), %%% loop check - X,Z ground
                reduce Z Y.                                         %%% (15)
reduce X X.                                                         %%% (16)
```

Arguments of predicates are separated by spaces rather than commas in λProlog, and the argument list of a predicate is not delimited by brackets. The syntax of λ-terms is similar to that of Lisp. Every predicate or function is at most unary, so larger arities have to be reduced to arity 1 by currying, that is, by replacing types $s_1 \times \cdots \times s_k \to s_0$ in the algebraic specification with types `s1 -> ... -> sk -> s0` in λProlog. As usual, the type constructor `->` is right-associative. Predicates always have type `··· -> o`.

`Kind` declarations are used to introduce type constructors. The three `kind` declarations in the first lines of `lpN` introduce the zero-adic type constructors `nat` and `bool`, and the monadic type constructor `lst`. These correspond to the sorts *nat*, *bool*, and *lst*(*A*) of *N*. Thus, apart from the declarations of the auxiliary predicates `extrule` and `reduce`, the correspondence between the signatures of *N* and `lpN` is straightforward. The translation of equations is equally straightforward. Put in the context of a new higher-order variable `H`, the left- and right-hand side of an equation become the first and second argument of the corresponding `extrule` fact. Note that $\lambda X. \cdots$ in the right-hand side of (14) becomes `(X \ ···` in λProlog. In addition to the `extrule` facts corresponding to the equations of *N*, the body of `lpN` consists of the clauses `(15)` and `(16)` for `reduce`. These are independent of *N*.

The normal form of a term *t* in the term language defined by the signature of *N* is obtained by submitting to `lpN` the question

```
?- reduce t' NF.
```

where `t'` is the corresponding term in the term language of `lpN`. Since free variables in *t* (if any) should not be instantiated during rewriting, they do not correspond to λProlog variables in `t'`, but are modelled by "simulated variables" (generic constants) `x`, `y`, ... in the following examples. Thus, even if *t* contains free variables, `t'` is a ground term.

Rewriting proceeds as follows. The `reduce` predicate attempts to apply `extrule` and, if successful, calls itself recursively on the reduct after performing the loop check `not(X = Z)`, where `not`

is the negation-as-failure predicate and = denotes higher-order unification. The loop check rejects algebraically correct but operationally useless matches (cf. Section 1.2, examples (I) and (II)). When it is evaluated, the values of both `X` and `Z` are ground terms because (i) the translated input term `t'` is always ground, and (ii) the equations are assumed to be such that their interpretation as left-to-right rewrite rules does not cause uninstantiated variables to enter the reduct (cf. Section 1.2, example (III)).

The rewrite strategy of `lpN` is determined primarily by the fact that β-reduction is a built-in rewrite rule that is performed implicitly by λProlog during unification, and by the order of the `extrule` facts. Redexes for rule r_m are reduced before redexes for rule r_n if $m<n$. The redex selection strategy for each individual rule is determined by λProlog's higher-order unification strategy. The latter can be influenced to some extent by the setting of the `projfirst` switch of the λProlog system. We reproduce a short sample run of the λProlog system using `lpN`:

```
?- use lpN.
lpN
yes

?- switch projfirst on.   %%% slightly more efficient in this
yes                       %%% application than projfirst off

?- switch tvw off.        %%% no type variable instantiation warnings
 yes

?- reduce (add (add zero (succ zero))) NF.
                          %%% first add is partially parameterized

NF = Var49 \ (add (succ zero) Var49) .
 yes

?- reduce (if y (cons f nil) (cons t nil)) NF.
                          %%% y is a "simulated variable" - see above
NF = cons (if y f t) nil .
 yes

?- reduce (if y (add (succ zero) (succ zer )) (succ (succ zero))) NF.
                          %%% y is a "simul  ed variable" - see above
NF = succ (succ zero) .
 yes

?- reduce (cons (cons suc  il) (cons (cons succ nil) nil)) NF.

NF = cons (cons succ nil) nil .
 yes

?- reduce ((compose (X \ (add X X)) (X \ (add X X))) (succ zero)) NF.

NF = succ (succ (succ (succ zero))) .
 yes

?- reduce (map (X \ (add X X))
|:           (cons zero (cons (succ zero) (cons zero nil)))) NF.

NF = cons zero (cons (succ (succ zero)) nil) .
 yes
```

```
?- reduce (map (compose succ) (cons succ (cons succ nil))) NF.
                          %%% compose is partially parameterized

NF = cons Var1900 \ (succ (succ Var1900)) nil .
 yes                      %%% see also Section 2.2

?- reduce (if y succ succ) NF.

NF = if y succ succ .     %%% NF = succ expected - see below
 yes
```

The last example is not reduced properly because the implementation of polymorphic higher-order unification in the version of λProlog we used was incomplete. When matching `if y succ succ` with the left-hand side of `(10')`, the polytype `A1 -> nat -> nat` initially inferred for `H` is never instantiated to `(nat -> nat) -> nat -> nat`. It is interesting to see how the matching behaves in this case:

```
?- switch tvw on.         %%% give type variable instantiation warnings
 yes

?- switch printtypes on. %%% print types of terms
 yes

?- if y succ succ = (H (if B (F X) (F Y))).
                          %%% "=" denotes higher-order unification
Trying to project on an argument with type
       A1
Do you want to go on? (y/n)y
Assuming for the moment that target type is primitive

H = Var24 : A1 \ Var25 : nat \
    (if y Var26 : nat \ (succ Var26) Var27 : nat \ (succ Var27) Var25)
B = B : bool
X = X : A1
F = F : A1 -> A2
Y = Y : A1 ;

 no
```

The only solution found leaves all variables in the left-hand side of `(10')` except `H` uninstantiated and is rejected by the loop check (cf. the examples in Section 1.2). The expected solution is found if the more precise type `(nat -> nat) -> nat -> nat` is associated with `H` in an *ad hoc* fashion:

```
?- if y succ succ = (H : (nat -> nat) -> nat -> nat (if B (F X) (F Y))).

H = Var26 : nat -> nat \ Var27 : nat \ (Var26 Var27)
B = y
X = X : A1
F = Var28 : A1 \ Var29 : nat \ (succ Var29)
Y = Y : A1 ;

H = Var54 : nat -> nat \ Var55 : nat \
```

```
    (if y Var56 : nat \ (succ Var56) Var57 : nat \ (succ Var57) Var55)
B = B : bool
X = X : A1
F = F : A1 -> nat -> nat
Y = Y : A1 ;

 no
```

The first solution yields the expected reduct when substituted in the right-hand side of (10′). The second solution is a more precisely typed version of the useless one found previously.

One of the rules of higher-order equational logic is the abstraction rule

$$\frac{\vdash t_1 = t_2}{\vdash \lambda X.t_1 = \lambda X.t_2}.$$

According to this rule, one would expect $\lambda Y.add(Y,zero)$ to be reduced to $\lambda Y.Y$, since $add(Y,zero)$ reduces to Y by equation (6). lpN does not do this, however:

```
?- reduce (add y zero) NF.
                          %%% y is a "simulated variable"
NF = y .                  %%% OK, but ...
 yes

?- reduce (Y \ (add Y zero)) NF.

NF = Y \ (add Y zero) .   %%% no reduction - first argument of (6′)
 yes                      %%% does not match "inside" an abstraction
```

Although this behavior of lpN is in accordance with ordinary functional programming practice, it should be noted that it is incomplete with respect to the above abstraction rule.

If the initial model is assumed to be extensional [Mei90] (see also Section 1.1), one would not only expect $\lambda Y.add(Y,zero)$ to be reduced to $\lambda Y{:}nat.Y$, but $\lambda Y.add(zero,Y)$ as well since it is extensionally equal to $\lambda Y{:}nat.Y$. Needless to say, our implementation does not do this either (cf. [Hee86]).

Finally, we give an example showing that lpN is not confluent for terms containing free variables. An alternative normal form can be obtained by backtracking. Note that lpN does not do this automatically.

```
?- reduce (if y (add x zero) (add x (succ zero))) NF.
                          %%% x and y are "simulated variables"

NF = if y x (succ x) ;    %%% first normal form

NF = if y x (succ (add x zero)) ;
                          %%% not a normal form
  ...

NF = add x (if y zero (succ zero)) ;
                          %%% second normal form
  ...

 no
```

The general translation scheme should be clear from `lpN`. The auxiliary names `reduce`, `extrule` and `H` should be chosen carefully to avoid clashes with user-defined names. Similarly, overloading of names that have a predefined meaning in λProlog (`true`, `false`, `list`, ...) should be avoided. Apart from the above-mentioned incompleteness problems and the possible non-termination of higher-order matching (which we have not encountered so far), the scheme is correct for higher-order rewrite systems that do not introduce new variables in the reduct, and that are terminating with the simple loop check shown as well as confluent. For rewrite systems lacking the latter property, the input term may have other normal forms besides the one computed.

2.2. Improving efficiency by adding specialized λProlog code

Some efficiency can be gained by combining the above method with one of the first-order schemes discussed in [Dro88, BW89]. To illustrate the general idea, we take Drosten and Ehrich's first-order scheme. In this case the λProlog code generated for *N* becomes:

```
module lpN2.

import lpN.  %%% see Section 2.1

type reduce2      A -> A -> o.
type analyze      A -> A -> o.
type prenormalize A -> A -> o.
type rule         A -> A -> o.

rule (add X zero)            X.                                %%% (6'')
rule (add X (succ Y))        (succ (add X Y)).                 %%% (7'')
rule (if t X Y)              X.                                %%% (8'')
rule (if f X Y)              Y.                                %%% (9'')
rule (if B (F X) (F Y))      (F (if B X Y)).                   %%% (10'')
rule (cons X (F (cons X L))) (cons X (F L)).                   %%% (11'')
rule (map F nil)             nil.                              %%% (12'')
rule (map F (cons X L))      (cons (F X) (map F L)).           %%% (13'')
rule (compose F G)           (X \ (F (G X))).                  %%% (14'')

analyze (succ I1) K       :- analyze I1 K1,
                             prenormalize (succ K1) K.        %%% (17)
analyze (add I1 I2) K     :- analyze I1 K1, analyze I2 K2,
                             prenormalize (add K1 K2) K.      %%% (18)
analyze (if I1 I2 I3) K   :- analyze I1 K1, analyze I2 K2,
                             analyze I3 K3,
                              prenormalize (if K1 K2 K3) K.   %%% (19)
analyze (cons I1 I2) K    :- analyze I1 K1, analyze I2 K2,
                             prenormalize (cons K1 K2) K.     %%% (20)
analyze (map I1 I2) K     :- analyze I1 K1, analyze I2 K2,
                             prenormalize (map K1 K2) K.      %%% (21)
analyze (compose I1 I2) K :- analyze I1 K1, analyze I2 K2,
                             prenormalize (compose K1 K2) K.%%% (22)

analyze X K               :- prenormalize X K.               %%% (23)

prenormalize X Y          :- rule X Z,
                             not(X = Z),  %%% loop check
                              analyze Z Y.                    %%% (24)
prenormalize X X.                                            %%% (25)
```

```
reduce2 X Y                    :- analyze X Z, reduce Z Y.        %%% (26)
                                  %%% reduce is defined in lpN
```

lpN2 extends lpN with code that is very similar to the Prolog code that would be generated by Drosten and Ehrich's scheme for N had it been a first-order specification. For each p-ary function f in the signature of N ($p \geq 1$), lpN2 contains a clause

```
analyze (f I1 ... Ip) K    :- analyze I1 K1,..., analyze Ip Kp,
                                 prenormalize (f K1 ... Kp).
```

Clause (23) catches everything not matched by the first argument of the preceding analyze cases. The facts (6'')-(14'') correspond directly to the equations (6)-(14). Clause (26) links the new code to the old code imported from lpN. The clauses (23)-(26) are independent of N.

The normal form of a term t in the term language defined by the signature of N is obtained by submitting to lpN2 the question

```
?- reduce2 t' NF.
```

where t' is the corresponding term in the term language of lpN2 (which is the same as that of lpN). Like before, free variables in t have to be replaced by "simulated variables" in t' (see Section 2.1).

On the examples we tried, lpN2 was from 1 to 5 times faster than lpN. It may actually be slightly slower if analyze is unable to perform any reductions. Consider, for instance, the term

```
(compose succ succ) zero.
```

The first argument of (22) does not match (its type is not even compatible), so the work done by analyze is wasted and the reduction to succ (succ zero) is performed by reduce using (14') with

```
H = Var : nat -> nat \ (Var zero)
F = succ
G = succ .
```

On the other hand, the reduction of

```
map (compose succ) (cons succ (cons succ nil))
```

to cons Var \ (succ (succ Var)) nil is speeded up by a factor of 5. Whereas lpN spends a large amount of time on useless matches, lpN2 performs the reduction in a highly deterministic manner using analyze.

3. FURTHER WORK

From a logical viewpoint, higher-order algebraic specification constitutes a natural integration of first-order algebraic specification and functional programming. Whether it is also a useful one, remains to be decided. We intend to perform further experiments with it using the implementation schemes discussed in this paper and perhaps more efficient ones still to be developed.

REFERENCES

[Bai90] R. Bailey, *Functional Programming with Hope* (Ellis Horwood, 1990).

[BMS80] R. Burstall, D. MacQueen, and D. Sannella, Hope: an experimental applicative language, in: *Conference Record of the 1980 Lisp Conference*, Stanford, 1980, 136-143.

[Bra86] I. Bratko, *Prolog Programming for Artificial Intelligence* (Addison-Wesley, 1986).

[BW89] L.G. Bouma and H.R. Walters, Implementing algebraic specifications, in: J.A. Bergstra, J. Heering, and P. Klint, eds., *Algebraic Specification* (ACM Press/Addison-Wesley, 1989) 199-

282.

[Dow91a] G. Dowek, Third-order matching is decidable, Rapport de Recherche, INRIA-Rocquencourt, 1991.

[Dow91b] G. Dowek, The undecidability of pattern matching in calculi where primitive recursive functions are representable, Rapport de Recherche, INRIA-Rocquencourt, 1991.

[Dro88] K. Drosten, Translating algebraic specifications to Prolog programs: a comparative study, in: J. Grabowski, P. Lescanne, and W. Wechler, eds., *Algebraic and Logic Programming*, Lecture Notes in Computer Science, Vol. 343 (Springer-Verlag, 1988) 137-146.

[Hee86] J. Heering, Partial evaluation and ω-completeness of algebraic specifications, *Theoretical Computer Science*, 43 (1986) 149-167.

[Hue76] G. Huet, Résolution d'équations dans les langages d'ordre 1,2, . . . , ω, Thèse de Doctorat d'Etat, Université de Paris-VII, 1976.

[JO91] J.-P. Jouannaud and M. Okada, A computation model for executable higher-order algebraic specification languages, in: *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science* (IEEE Computer Society Press, 1991) 350-361.

[Klo90] J.W. Klop, Term rewriting systems, Report CS-R9073, CWI, Amsterdam, 1990. To appear in: S. Abramsky, D. Gabbay, and T. Maibaum, eds., *Handbook of Logic in Computer Science*, Vol. II (Oxford University Press).

[Mei90] K. Meinke, Universal algebra in higher types, Report CSR 12-90, Computer Science Division, Department of Mathematics and Computer Science, University College of Swansea, September 1990.

[Mei91] K. Meinke, A recursive second order initial algebra specification of primitive recursion, Report CSR 8-91, Computer Science Division, Department of Mathematics and Computer Science, University College of Swansea, June 1991.

[MG85] J. Meseguer and J.A. Goguen, Initiality, induction, and computability, in: M. Nivat and J.C. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, 1985) 459-541.

[Nip91] T. Nipkow, Higher-order critical pairs, in: *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science* (IEEE Computer Society Press, 1991) 342-349.

[NM88] G. Nadathur and D. Miller, An overview of λProlog, in: R.A. Kowalsi and K.A. Bowen, eds., *Logic Programming - Proceedings of the Fifth International Conference and Symposium*, Vol. 1 (The MIT Press, 1988) 810-827.

[PN90] L.C. Paulson and T. Nipkow, Isabelle tutorial and user's manual, Technical Report No. 189, Computer Laboratory, University of Cambridge, January 1990.

[Poi86] A. Poigné, On specifications, theories, and models with higher types, *Information & Control*, 68 (1986) 1-46.