Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

P.R.H. Hendriks

Lists and associative functions in algebraic specifications
- semantics and implementation -

Computer Science/Department of Software Technology Report CS-R8908 March

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.R.H. Hendriks

Lists and associative functions in algebraic specifications
- semantics and implementation -

Computer Science/Department of Software Technology Report CS-R8908 March

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Lists and Associative Functions in Algebraic Specifications — Semantics and Implementation —

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Adding lists and associative functions to an algebraic specification formalism will not add expressive power because both features are definable in such a formalism. In contrast, it is possible to generate more powerful implementations for specifications if these features are present.

Key Words & Phrases: Software Engineering, Algebraic Specification, Associativity, List, Specification Language, Term Rewriting, String Rewriting, Executable Specification, Prolog.

1985 Mathematics Subject Classification: 68N20 **[Software]:** Compilers and generators; 68Q50 **[Theory of computing]:** Grammars, rewriting systems; 68Q65 **[Theory of computing]:** Abstract data types.

1987 CR Categories: D.2.1 **[Software Engineering]:** Requirements/Specifications; D.3.1 **[Programming Languages]:** Formal Definitions and Theory; D.3.3 **[Programming Languages]:** Language Constructs - Abstract data types; D.3.4 **[Programming Languages]:** Processors; F.3.1 **[Logics and Meanings of Programs]:** Specifying and Verifying and Reasoning about Programs - *Specification techniques*; F.3.2 **[Logics and Meanings of Programs]:** Semantics of Programming Languages - *Algebraic approaches to semantics.*

Note: Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

Note: This paper will be submitted for publication elsewhere.

1. Introduction

Standard algebraic specifications only support the use of fixed arity functions, but using functions with iterated sorts in their input type often gives more elegant specifications. An iterated sort `S*` or `S+` indicates an argument of a function in which, respectively, zero or more terms, or one or more terms of the same sort `S` are allowed. Some examples of such functions are:

- natural numbers as lists of one or more digits:
  `nat: DIGIT+ -> NAT,`
- tables which are a list of zero or more pairs of keys and their corresponding entries:
  `pair: KEY # ENTRY -> PAIR  table: PAIR* -> TABLE,` and
- programs (in some simple programming language) which are defined as a list of one or more declarations followed by a list of zero or more statements:
  `prog: DECLARATION+ # STATEMENT* -> PROGRAM.`

Such lists of terms of the same sort are, of course, definable in standard algebraic specification formalisms. Consequently, these list operations will not add expressive power to the formalism. On the other hand, use of these lists improves readability of specifications in many cases (see Section 4.1 for an example).

It is possible to generate code from an algebraic specification automatically if it is viewed as a term

Report CS-R8908
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

rewriting system: each equation is read as a rewrite rule from left to right. In this setting one has to choose a bias in the representation of lists to create a confluent and terminating term rewriting system. As a consequence, auxiliary functions are needed if, for example, the first element as well as the last element of the list have to be inspected. Suppose we want to specify natural numbers as lists of one or more digits and the following head-tail-like representation of lists of digits is chosen:

```
inj : DIGIT              -> DIGIT-LIST
add : DIGIT # DIGIT-LIST -> DIGIT-LIST
nat : DIGIT-LIST         -> NAT
```

Using this representation it is easy to specify how to remove leading zeros from a natural number, but an auxiliary function is needed to access the last digit of the list in order to express that the successor (`succ : NAT -> NAT`) of a natural number ending in 1 is identical to the same list of digits ending in 2. Due to the fact, that lists as presented in this paper are flat structures, it is easier to write specifications from which executable code can be generated. They provide an elegant tool to combine term rewriting with string rewriting [Jan86].

Concatenation of lists is an associative binary operation. As a consequence, the semantics of algebraic specifications with lists can be expressed in terms of algebraic specifications with associativity. For this reason, we will first discuss algebraic specifications with associative binary operators and their implementation in terms of rewritting modulo associativity. Associativity of a binary function is denoted by adding the **assoc**-attribute to it. This predicate is also available in the specification languages AXIS [RC88], CEC [BGS88] and OBJ [FGJM85, GKKMMW88].

Research in lists and associative functions in algebraic specifications is not only inspired by the wish to improve the elegance of specifications and the possibilities to generate code for such specifications. In our case, it is also inspired by the wish to combine the Algebraic Specification Formalism ASF [BHK89] and the Syntax Definition Formalism SDF [Chapter 6 of BHK89, HHKR]. Both formalisms were developed as intermediate steps in the development of a language definition formalism. It is the goal of ESPRIT-project 348 (GIPE - Generation of Interactive Programming Environments) to make a system which can generate an interactive programming environment for a programming language from a formal definition of that language. Lists and associativity of binary functions are both features which occur naturally in SDF. Their semantical consequences have to be handled in the algebraic specification formalism and its implementation, however. As an example of an application of algebraic specifications with lists and associativity we present a specification in the combination of ASF and SDF in Section 5.

A general discussion of algebraic specifications, their semantics and a general scheme to generate implementations for them is given in Section 2. In Section 3 we describe the **assoc**-attribute, and the list operations are added to the formalism in Section 4. Finally, Section 6 contains conclusions and some remarks.

2. Algebraic specifications

2.1. Example

All examples in this paper are given in the Algebraic Specification Formalism ASF [BHK89] or extensions of it. However, the ideas and techniques in this paper are independent of ASF. We now explain aspects of ASF that will be used in the examples. Each specification in ASF is, ultimately, equivalent to a first order signature and a set of (conditional) equations (see next section). ASF has several features to support modularization of a specification:

- Exports:
  Each module may have an **exports** section consisting of a (possibly incomplete) signature. The sorts and functions declared in this section are visible outside the module.
- Hidden sorts and functions:
  Sorts and functions that are local to a module are declared in the **sorts** and **functions** sections.
- Imports:
  The **imports** section contains the names of modules that have to be incorporated in a module. While importing a module it is possible to bind its parameters, to rename its signature (see below) or to

perform a combination of these.

- Parameters:
  Parameters are declarations of (possibly incomplete) signatures which are formal parameters of the module. They are declared in the **parameters** section (see for example Section 4.3). They can be bound to actual sorts and functions of a module when the parameterized module is imported.
- Renamings:
  Upon import of a module parts of the signature of the module can be renamed if changes in names of sorts or functions are desirable to avoid, for instance, name clashes.

Throughout this paper we will use several algebraic specifications of natural numbers (NAT) and (finite) sets of natural numbers (SET) as examples. In the following algebraic specification 0 is represented as the constant `zero` and all natural numbers n greater than 0 are represented as `succ(succ(...succ(zero)))` with n repetitions of `succ`. In this specification addition (`plus`) and multiplication (`mult`) on natural numbers are defined. The constant `empty` stands for the empty set and all other sets are constructed by adding an element to a set using the function `add`. Finally, the union operator on sets (`union`) is specified.

```
module Natural-Numbers
begin
   exports
   begin
      sorts NAT, SET
      functions
         zero  :              -> NAT
         succ  : NAT          -> NAT
         plus  : NAT # NAT    -> NAT
         mult  : NAT # NAT    -> NAT
         empty :              -> SET
         add   : NAT # SET    -> SET
         union : SET # SET    -> SET
   end
   variables
      n, n1, n2 : -> NAT
      s, s1, s2 : -> SET
   equations
      [1]   plus(zero, n)       = n
      [2]   plus(succ(n1), n2)  = succ(plus(n1, n2))
      [3]   mult(n, zero)       = zero
      [4]   mult(n1, succ(n2))  = plus(mult(n1, n2), n1)
      [5]   add(n, add(n, s))   = add(n, s)
      [6]   add(n1, add(n2, s)) = add(n2, add(n1, s))
      [7]   union(empty, s)       = s
      [8]   union(add(n, s1), s2) = add(n, union(s1, s2))
end Natural-Numbers
```

Equation `[5]` says that adjacent identical elements in a set may be replaced by a single occurrence of the element. This has to be combined with equation `[6]` (which states the irrelevance of the order in which the elements are added to the set) to allow arbitrary occurrences of identical elements in a set.

2.2. Definitions

An *algebraic specification* $<\Sigma, E>$ consists of a signature Σ and a set of (possibly conditional) equations E. A *signature* $\Sigma \equiv <S_\Sigma, F_\Sigma>$ consists of a set of sort symbols S_Σ and a set of function symbols F_Σ. An implicit typing function of F_Σ to $S_\Sigma^* \times S_\Sigma$ exists which assigns to each element `f` of F_Σ an input type `S1 # S2 # ... # Sn` where $n \geq 0$ and an output type `S`. Such a function will be denoted by `f: S1 # S2 # ... # Sn -> S`. In most formalisms, overloading of function symbols is allowed, i.e., more than one typing of a function symbol $f \in F_\Sigma$ is possible. To assure unique typing of *each* term it is necessary that no functions with identical name and input type exists. To simplify the theoretical description we here forbid overloading because this can be remedied by encoding the type information in the function names.

Let a set of typed variables X be given, i.e., to each variable `x` a unique type `S` $\in S_\Sigma$ is attached and this will be denoted by `x: -> S`. Given a signature $\Sigma = <S_\Sigma, F_\Sigma>$ we can define the set $T^S_\Sigma(X)$ of terms of type `S` over Σ as the smallest set such that:

- `x` is an element of $T^S_\Sigma(X)$ for each variable `x: -> S`.
- `c` is an element of $T^S_\Sigma(X)$ for each (constant) function `c: -> S`.
- For each function `f: S1 # S2 # ... # Sn -> S` with `n` ≥ 1 and for all terms `t1` $\in T^{S1}_\Sigma(X)$, `t2` $\in T^{S2}_\Sigma(X)$, $\cdots$, `tn` $\in T^{Sn}_\Sigma(X)$ we define `f(t1, t2, ..., tn)` to be an element of $T^S_\Sigma(X)$.

The set of *closed terms* (terms without variables) of type `S` is denoted by T^S_Σ. The set of terms $T_\Sigma(X)$ over Σ is the union of $T^S_\Sigma(X)$ for all `S` $\in S_\Sigma$.

An *unconditional equation* of type `S` $\in S_\Sigma$ over a given signature $\Sigma = <S_\Sigma, F_\Sigma>$ is an element of $Eq^S \equiv T^S_\Sigma(X) \times T^S_\Sigma(X)$. It is denoted by `s = t` where `s, t` $\in T^S_\Sigma(X)$. The set of all (possibly conditional) equations E in an algebraic specification $<\Sigma, E>$ is a subset of $Eq \times Eq^*$, where Eq denotes the set of unconditional equations $Eq \equiv \bigcup_{S \in S_\Sigma} Eq^S$. Conditional equations with at least one condition are denoted by

```
s = t when s1 = t1, s2 = t2, ..., sn = tn
```

An *assignment* ρ is a function which assigns to each variable of type `S` a term of the same type. In short: it is a function $X \rightarrow T_\Sigma(X)$ such that $\rho(x) \in T^S_\Sigma(X)$ holds for all `x: -> S`. Each assignment ρ can be extended in a natural way to a function defined on the complete set of terms $T_\Sigma(X)$ such that it does not change the type of a term:

- For each (constant) function `c: -> S` we define $\rho(c) \equiv c$.
- For all functions `f: S1 # S2 # ... # Sn -> S` with `n` ≥ 1 ρ is defined by $\rho($`f(t1, t2, ..., tn)`$) \equiv$ `f(`ρ`(t1), `ρ`(t2), ..., `ρ`(tn))`.

Now, we can give the axioms and rules of (conditional) equational logic, which may be used to construct proof trees:

$$E \vdash t = t \tag{Eq1}$$

$$\frac{E \vdash t1 = t2}{E \vdash t2 = t1} \tag{Eq2}$$

$$\frac{E \vdash t1 = t2 \quad E \vdash t2 = t3}{E \vdash t1 = t3} \tag{Eq3}$$

$$\frac{E \vdash s1 = t1 \quad E \vdash s2 = t2 \quad \cdots \quad E \vdash sn = tn}{E \vdash f(s1, s2, \ldots, sn) = f(t1, t2, \ldots, tn)} \tag{Eq4}$$

$$\frac{E \vdash s = t}{E \vdash \rho(s) = \rho(t)} \tag{Eq5}$$

$$\frac{(s = t) \in E}{E \vdash s = t} \tag{Eq6}$$

$$\frac{E \vdash \rho(s1) = \rho(t1) \quad E \vdash \rho(s2) = \rho(t2) \quad \cdots \quad E \vdash \rho(sn) = \rho(tn) \qquad (s = t \textbf{ when } s1 = t1, s2 = t2, \ldots, sn = tn) \in E}{E \vdash \rho(s) = \rho(t)} \tag{C-Eq}$$

which holds for all terms `s, t, s1, s2, ` $\cdots$`, sn, t1, t2, ` $\cdots$`, tn` $\in T_\Sigma(X)$; for all functions `f` $\in F_\Sigma$ and for all assignments $\rho : X \rightarrow T_\Sigma(X)$. Axioms and rules (Eq1) through (Eq6) together constitute equational logic. Rule (C-Eq) handles conditional equations.

2.3. Semantics

The most natural semantics for algebraic specifications is the initial algebra semantics. For an extensive treatment of initial algebras we refer to [MG85]. In this section we only describe the main concepts.

Models of algebraic specifications are (many-sorted) algebras. A *many-sorted algebra* A is a structure $<S_A, F_A>$ which consists of a set of mutually disjoint non-empty sets S_A (the *carriers* of A) and a set of functions F_A. (For a discussion of empty carriers in algebras we refer to [GM87].) Each element of F_A is a total function from a tuple of carriers of A to a carrier of A. Given a signature Σ, a *many-sorted* Σ-*algebra* is an algebra of which S_A and F_A, respectively, consist of interpretations for the sorts and functions of the signature. A many-sorted Σ-algebra A is a *model* of a given algebraic specification $<\Sigma, E>$ if the interpretation which assigns to each sort symbol $s \in S_\Sigma$ an element of S and to each function symbol $f \in F_\Sigma$ an element of F is such that the interpretation of all equations E holds in the algebra. Some examples of algebras that are models of the algebraic specification given in Section 2.1 are:

- $A_1 \equiv <\{N, FS\}, \{0, S, +, *, \emptyset, \oplus, \cup\}>$ where N is the set of natural numbers and FS is the set of all finite sets of natural numbers and the set of functions corresponds to the obvious interpretations of the specified functions.
- A_2 is the algebra similar to A_1 where FS is replaced by the set of all sets of natural numbers S.
- $A_3 \equiv <\{\{N\}, \{S\}\}, \{z, s, p, m, e, a, u\}>$ where the natural numbers as well as the sets are interpreted as one element N and S respectively and all functions are trivial ones.

The algebra A_1 is one of the intended models of the given specification. It contains no *junk* in contrast to the algebra A_2 in which, for example, the set of *all* odd numbers exists which has no denotation in the specification. Also, A_1 contains no *confusion* in contrast to A_3 where the interpretation of two terms can be equal while their equality cannot be proved from the equations in the specification.

A model of a specification without junk and confusion is the *initial algebra* of the specification. In case of an algebraic specification in which each sort contains at least one closed term the initial algebra exists. The *term model* $<\{T_\Sigma^s/=\}_s, \{\overline{f} \mid f \in F_\Sigma\}>$ is an example of the initial algebra of the specification $<\Sigma, E>$. Its carriers are the sets of closed terms T_Σ^s over Σ in which terms are identified which belong to the same congruence class defined by:

$$t1 = t2 \iff E \vdash t1 = t2.$$

where $\vdash$ represents (conditional) equational provability as defined in the previous section. The functions in the term model are defined by

$$\overline{f}([t1], [t2], \cdots, [tn]) \equiv [f(t1, t2, \cdots, tn)]$$

where $[t]$ is the congruence class to which a term t belongs.

2.4. Implementation in Prolog

This section describes one of the possibilities to implement an algebraic specification. The implementation methods described here are used in the ASF system [Hen88]. It is a simple environment for compiling and testing ASF specifications. An algebraic specification is viewed as a term rewriting system by interpreting the conclusion of each equation as a rewrite rule from left to right which, in turn, is implemented in C-Prolog [PWBBP85]. For a more general overview of implementation strategies of algebraic specifications we refer to [Chapter 5 of BHK89].

In the next section we discuss specifications with unconditional equations and in Section 2.4.2 the implementation of equations with conditions is handled.

2.4.1. Implementation of equations without conditions

Before discussing the implementation we need some definitions. A *term rewriting system* $<\Sigma, R>$ consists of a signature Σ and a set of rewriting rules R, where $R \subset \bigcup_{s \in S_\Sigma} T_\Sigma^s(X) \times T_\Sigma^s(X)$. Rewriting rules are written as $s \rightarrow t$. A *context* (denoted by $C[x]$) is a term which contains a single occurrence of the variable x. The result of substituting the term $t \in T_\Sigma^s(X)$ for the variable `x: -> S` in the context $C[x]$ is denoted by $C[t]$. The rewriting rules of a term rewriting system $<\Sigma, R>$ give rise to *reduction steps*, as follows: $s \rightarrow_R t$ if

and only if a context $C[x]$ exists such that s equals $C[\rho(s1)]$ and t equals $C[\rho(t1)]$ for some assignment ρ and reduction rule $s1 \rightarrow t1$ of R. The subterm $\rho(s1)$ of s is called a *redex*. A term s is a *normal form* if there is no term t such that $s \rightarrow_R t$.

For a given term rewriting system $<\Sigma, R>$ several *reduction strategies* exist. They define for each term $t \in T_\Sigma^S(X)$ a set of subterms of t which are redexes for R, and which rule of R has to be applied to each of these redexes. An *outermost reduction strategy* only reduces redexes that are not subterms of another redex of the term. An *innermost reduction strategy* reduces redexes that do not contain any other redex. In both cases the order in which reductions are performed can be varied. The most common variants are *leftmost* strategies (which always choose the leftmost redex), and *parallel* strategies (in which all outermost or innermost redexes are reduced simultaneously).

Several ways of implementing a term rewriting system in Prolog are known. In [DE84, EY87, Wie87] several methods are described in which Prolog predicates model a certain reduction strategy: in [DE84] Drosten and Ehrich give predicates for leftmost innermost reduction; the interpretational approach of van Emden and Yukawa [EY87] yields a parallel outermost strategy and Wiedijk in [Chapter 5 of BHK89, Wie87] gives a schematic overview of the different possibilities to implement a term rewriting system in Prolog in this manner.

In contrast to the above-mentioned interpretational approaches, we generate faster code by using a variant of the compilational approach described in [EY87], see also [Chapter 9 of BHK89]. This method corresponds naturally to the way in which one would implement functions in Prolog. It regards each function of the specification as a relation of its input and output. One of the major deficits of the compilational approach is its leftmost innermost reduction strategy. This may cause non-termination even in case of weakly terminating term rewriting systems (i.e., systems in which each term has at least one normal form).

The compilational approach of [EY87] has been modified because the original method can only handle specifications in which each equation is of the form `f(t1, t2, ..., tm) = t` where `t1, t2, ···, tm` may not contain defined functions. *Defined functions* are functions which occur as main symbol in the left-hand side of an equation. In [EY87] Prolog predicates are generated only for defined functions. The implementation described here circumvents this constraint by generating predicates for all functions and adding a "catch all"-rule for each function.

As an example, we first show the code generated from the specification given in Section 2.1.

```
/* >>> Equations                                        <<< */

plus(N, zero, N).                                    /* [1] */
plus(Res, succ(N1), N2)                              /* [2] */
   :- plus(Temp1, N1, N2),
      succ(Res, Temp1).

mult(Res, N, zero)                                   /* [3] */
   :- zero(Res).
mult(Res, N1, succ(N2))                              /* [4] */
   :- mult(Temp1, N1, N2),
      plus(Res, Temp1, N1).

add(Res, N, add(N, S))                               /* [5] */
   :- add(Res, N, S).
add(Res, N1, add(N2, S))                             /* [6] */
   :- add(Temp1, N1, S),
      add(Res, N2, Temp1).

union(S, empty, S).                                  /* [7] */
union(Res, add(N, S1), S2)                           /* [8] */
   :- union(Temp1, S1, S2),
      add(Res, N, Temp1).

/* >>> Catch-all                                        <<< */

zero(zero).
```

```
succ(succ(X1), X1).
plus(plus(X1, X2), X1, X2).
mult(mult(X1, X2), X1, X2).
empty(empty).
add(add(X1, X2), X1, X2).
union(union(X1, X2), X1, X2).
```

For each n-ary function an $(n+1)$-ary Prolog predicate is generated whose first argument is the output after application of the function to the other n arguments. Each equation of the specification is translated to one Horn-clause in Prolog. A catch-all rule is added for each function returning the normal form if no equation is applicable.

A term that has to be reduced to normal form is translated into Prolog questions. For example, the term `mult(succ(zero), plus(succ(zero), zero))` is translated into:

```
?- zero(T1), succ(T2, T1),
   zero(T3), succ(T4, T3), zero(T5), plus(T6, T4, T5),
   mult(Res, T2, T6).
```

The normal form `succ(zero)` of the term to be reduced is the value of the variable `Res`. It is also possible to reduce *open terms* (i.e., terms with variables) as long as no values are given to the variables occurring in such terms. A term like `plus(succ(succ(n1), n2))` is translated into:

```
?- succ(T1, n1), succ(T2, T1), plus(Res, T2, n2).
```

The program will now return `succ(succ(plus(n1, n2)))`.

From the above example it is already evident that the generated code as well as the translation of input terms can be optimized. The fact that some functions do not occur as head symbol in the left-hand side of any equation may be exploited for optimizations. Other optimizations can be achieved by recognizing identical subterms in right-hand sides of equations, or by remembering the normal forms of terms. We will not discuss these optimizations here.

How can the above-mentioned code for an algebraic specification be generated? Each equation is typechecked before a Horn-clause is generated for it. During typechecking the use of variables in the equation is checked. It is impossible to generate code if the left-hand side of an equation is a variable or if the right-hand side contains variables which do not occur in the left-hand side. (See the next section for conditional equations in which case the latter might not be necessary). At the same time, we construct a list of variables and their corresponding Prolog variables which are to be used in the code.

The code generation process itself consists of two disjoint parts:

1. The right-hand side of an equation is changed into a *list of predicates* (which will become the conditions of the resulting Horn-clause) and a *translated term* (which will contain the result of the computation).
2. The conclusion of the Horn-clause is constructed from the left-hand side of the equation and the translated term generated in step 1.

We will now describe these steps in somewhat more detail.

The conditions of the clause in step 1 are constructed using induction on the complexity of the term `t` in the right-hand side of the equation:

- `t` ≡ `x`:
  The list of predicates to be generated is empty and the translated term is the Prolog variable that corresponds to the variable `x`.
- `t` ≡ `c`:
  The translated term of a constant `c` is a "fresh" Prolog variable `Var` (i.e., a Prolog variable which has not been assigned to any of the variables in the equation and which has not yet been used in the code generation process). The list of predicates contains just one element: the predicate `c(Var)`.
- `t` ≡ `f(t1, t2, ..., tn)` with `n` ≥ 1:
  The translated term of `t` is, once again, a "fresh" Prolog variable `Var`. Let $L_1, L_2, \cdots, L_n$ be the lists of predicates, respectively, generated for the subterms `t1`, `t2`, ···, `tn`, and, let `T1`, `T2`, ···, `Tn` be the translated terms corresponding to these subterms. The list of predicates for `t` is a

concatenation of the lists $L_1, L_2, \cdots, L_n$, and the predicate `f(Var, T1, T2, ..., Tn)` added at the end of it.

In step 2 the conclusion of the Horn-clause is generated: if the left-hand side of the equation is of the form `f(t1, t2, ..., tn)` with $n \geq 1$ the conclusion is `f(Res, T1, T2, ..., Tn)`, where `Res` is the translated term from the right-hand side of the equation and the `Ti` are constructed from `ti` by changing each variable into the corresponding Prolog variable. If the left-hand side of the equation is a constant `c` the conclusion is the predicate `c(Res)`.

Finally, for each function from the specification the catch-all rule is added. It consists for each n-ary function symbol `f` of the Horn-clause `f(f(X1, X2, ..., Xn), X1, X2, ..., Xn)`.

The translation of an input term to a Prolog question is done similarly to the above-mentioned method for decomposing the right-hand side of an equation. The only difference is the translation of a variable `x` which is now translated into the variable itself (as a Prolog atom). If the program terminates, the value of the translated term is one of the normal forms of the input term.

2.4.2. Implementation of conditions

In the ASF system [Hen88] the evaluation of conditions is determined by the way variables are used in the corresponding equation. Let V be the set of variables used in the left-hand side of the conclusion of the equation. The conditions are checked in the order in which they are specified. There are two kinds of conditions:

1. The condition contains only variables which are elements of V. Now, both sides of the condition will be reduced to normal form and the condition succeeds if these normal forms are identical.
2. One of the sides of the condition contains only variables which occur in V. Upon execution of the generated code this term is reduced to normal form and the other side of the condition has to match this normal form. The new variables in the other side are added to V.

Finally, it is checked that all variables in the right-hand side of the conclusion of the equation are members of the resulting set V. Hence, an error-message is given and no code is generated if in both sides of a condition or in the right-hand side of an equation variables are used which have not been introduced before.

How to generate the appropriate code for a conditional equation? For each condition a list of Prolog predicates is generated and these lists are concatenated in order in which the conditions are given in the equation. The list of predicates constructed in this way is added before the list constructed from the right-hand side of the conclusion of the equation as described in the previous section.

How to generate code for each of the conditions? This depends, of course, on the cases mentioned above:

1. Both sides of the condition `tl = tr` will be decomposed in a list of predicates and a translated term as described in Section 2.4.1. Let L_l with `Tl` and L_r with `Tr`, respectively, be the lists of predicates and the translated terms for `tl` and `tr`. The code for this condition is a concatenation of L_l and L_r (in an arbitrary order) followed by testing the literal equality of `Tl` and `Tr`: `Tl == Tr`.
2. Suppose the condition is `tc = tn`, where `tc` is the side of the condition which contains only variables which where known and `tn` contains some new variables. Now `tc` is decomposed into a list of predicates L_c and a translated term `Tc` as described in Section 2.4.1. All variables occurring in `tn` are changed into their corresponding Prolog variable resulting in a Prolog term `Tn`. The code is the list L_c followed by a unification of `Tc` with `Tn`: `Tc = Tn`.

2.4.3. Soundness and completeness

The generated code is *sound*, i.e. for all (possibly open) terms `t1` and `t2` the following holds: if the implementation I returns `t2` as the result of evaluating `t1` (notation: $eval_I(\texttt{t1}) = \texttt{t2}$), then the equality of `t1` and `t2` can be proved using the equations E of the specification. In short notation:

$$eval_I(\texttt{t1}) = \texttt{t2} \;\Rightarrow\; E \vdash \texttt{t1} = \texttt{t2}$$

The proof of this is similar to the proof of the correctness of the compilational approach in [EY87].

More interesting is the question whether the converse (the *completeness* of the implementation)

holds. Or, more precisely, if two terms `t1` and `t2` are given such that they can be proved equal using E, can we use the implementation to show them to be equal? In short:

$$E \vdash \texttt{t1} = \texttt{t2} \;\Rightarrow\; \exists \texttt{t}\ \mathit{eval}_I(\texttt{t1}) = \texttt{t} \wedge \mathit{eval}_I(\texttt{t2}) = \texttt{t}?$$

In general, this is too much to hope for, because it is undecidable whether an equation is derivable from a given set of equations. Incompleteness might be caused by non-termination of the implementation, non-confluence, and the inability to decide conditions completely. For an extensive treatment we refer to [Kap87].

3. Algebraic specifications with associativity

3.1. Example

To illustrate associativity in an algebraic specification we change the example given in Section 2.1 by using the **assoc**-attribute to declare the associativity of the addition and multiplication on natural numbers and the union operator on sets:

```
exports
begin
   sorts NAT, SET
   functions
      zero  :             -> NAT
      succ  : NAT         -> NAT
      plus  : NAT # NAT   -> NAT {assoc}
      mult  : NAT # NAT   -> NAT {assoc}
      empty :             -> SET
      add   : NAT # SET   -> SET
      union : SET # SET   -> SET {assoc}
end
```

The equations of the specification are not changed.

3.2. Definitions

An *algebraic specification with associativity* <Σ, *E*, *assoc*> consists of an algebraic specification <Σ, *E*> and a predicate *assoc* defined on the set of function symbols F_Σ. Here we will only describe associativity for functions of the form:

```
f: S # S -> S
```

For functions with other typings it is either impossible to give a semantics for associativity or it is unclear what its meaning should be. Given a function `f: S1 # S2 # S1 -> S1` (with `S1` ≠ `S2`) associativity could stand for the equation `f(x1, y1, f(x2, y2 , x3)) = f(f(x1, y1, x2), y2, x3)` where `x1, x2, x3: -> S1` and `y1, y2: -> S2`. However, for a function `g: S # S # S -> S` it is questionable whether it implies the equations: `g(z1, z2, g(z3, z4 , z5)) = g(z1, g(z2, z3, z4), z5) = g(g(z1, z2, z3), z4, z5)` where `z1, z2, z3, z4, z5: -> S`.

3.3. Semantics

The semantics of an algebraic specification with associativity <Σ, *E*, *assoc*> is defined as the semantics of the algebraic specification <Σ, *E'*>, where *E'* is constructed by adding the corresponding associative law to the set of equations *E* for each associative function. Hence, for each function `f: S # S -> S` for which *assoc*(`f`) holds, the equation

```
f(x, f(y, z)) = f(f(x, y), z)
```

is added (where `x, y, z: -> S`).

3.4. Implementation in Prolog

What is the advantage for code generation of the use of the **assoc**-attribute instead of the corresponding associative law? When implementing the associative law in the same way as other equations one has to choose a direction for it. In general a non-terminating term rewriting system results if the law is added as the two rewrite rules

```
f(x, f(y, z)) → f(f(x, y), z)
```

and

```
f(f(x, y), z) → f(x, f(y, z)).
```

As a consequence, the associative law can only be used in just one direction when terms are rewritten. In general both directions of the law are needed, however. By the way, in the example given above it does not make any difference as long as one only reduces closed terms. All three associative operators are defined here in such a way that all closed terms reduce to normal forms that do not contain them.

When generating code for a term rewriting system modulo associativity it is easier to handle an associative operator `f: S # S -> S` as a function `f'` which has two or more arguments of sort `S` and output `S`. All terms are *flattened* which means that terms like `f(a, f(b, c))` and `f(f(a, b), c)` are changed into `f'(a, b, c)`. Each occurrence of `f` is replaced by `f'` and all arguments of `f` whose head symbol is also `f` is replaced by its arguments. A term with head symbol `f'` has no arguments with `f'` as head symbol.

When rewriting modulo associativity we have to consider the following three complications:

1. Matching of terms is different from standard matching. The left-hand side of a rewrite rule of the form `f'(x, a)` must match terms like `f'(a, a)` and `f'(b, b, a)`. After matching, the value of `x` should be `a` in the first example and `f'(b, b)` in the second one.
2. We have to check whether a rewrite rule is applicable to the sublist of the arguments of an associative operator `f'`. Given a term `f'(a, b, c)` it may be that a rewrite rule for `f'(b, c)` exists but that there are no rewrite rules for `f'(a, b)` and `f'(a, b, c)` itself.
3. When constructing a term whose head symbol is an associative operator `f'` its arguments may not have `f'` as head symbol. Terms like `f'(a, f'(b, c))` are forbidden and must be replaced by their flattened variant `f'(a, b, c)`.

In the implementation only flattened terms are used, and we will no longer add an accent ' to the function name. Instead of the standard (n+1)-ary Prolog predicate which is generated for an n-ary function a binary predicate is generated for associative operators. The arguments of `f` are put into a Prolog list which is used as the second argument of the predicate. The first argument is still the output of the function after application of the function to its arguments. The normal form of a term in which an associative operator `f` occurs is represented by a unary function `f` whose argument is also the Prolog list containing the arguments of the associative operator.

For each of the above-mentioned three aspects of rewriting modulo associativity Prolog predicates are needed. These predicates are identical for all associative operators and for this reason the corresponding code does not have to be generated. The first argument `Name` of each of the predicates is the name of the associative operator. The code for these predicates is the following:

```
/* >>> General Predicates                                    <<< */

assoc_decomp(Name, Result, Term, Rest)
   :- append([Head| Tail], Rest, Result),
      assoc_arg(Name, Term, [Head| Tail]).

assoc_arg(Name, Result, [Arg1, Arg2| Args])
   :- Result =.. [Name, [Arg1, Arg2| Args]],
      !.
assoc_arg(_, Term, [Term]).

assoc_all(_, [Term], Term)
```

```
   :- !.
assoc_all(Name, Input, Result)
   :- split(L1, [L2_Arg1, L2_Arg2| L2_Tail], L3, Input),
      Pred =.. [Name, Res, [L2_Arg1, L2_Arg2| L2_Tail]],
      Pred,
      !,
      assoc_arg(Name, Res, L2_New),
      split(L1, L2_New, L3, Input_New),
      assoc_all(Name, Input_New, Result).
assoc_all(Name, Input, Result)
   :- Result =.. [Name, Input].

split([], L2, L3, List)
   :- append(L2, L3, List).
split([Head| Tail], L2, L3, [Head| Tail1])
   :- split(Tail, L2, L3, Tail1).

append([], List, List).
append([Head| Tail], List, [Head| Tail1])
   :- append(Tail, List, Tail1).

assoc_flat(_, [], []).
assoc_flat(Name, [Head| Tail], Result)
   :- Head =.. [Name, Args],
      !,
      assoc_flat(Name, Tail, Tail1),
      append(Args, Tail1, Result).
assoc_flat(Name, [Head| Tail], [Head| Tail1])
   :- assoc_flat(Name, Tail, Tail1).
```

For term matching (case 1) the predicates `assoc_arg` and `assoc_decomp` are used. The predicate `assoc_decomp` divides a list of arguments `Result` of an associative operator in a term `Term` and the rest of the list `Rest`. It uses `assoc_arg` to change an associative operator and its arguments into the corresponding term. If the list of arguments contains two or more elements the term returned is the associative operator applied to its arguments. If the list contains only one term this term is returned.

To compute a normal form of an associative operator `Name` applied to its arguments `Input`, the predicate `assoc_all` is defined. It successively tries to apply an equation to each sublist of the list of arguments (case 2). The first clause returns the argument itself if the input list contains just one argument. Next, Prolog backtracking is used to split the list of arguments in three sublists such that an equation can be applied to the middle one (which contains at least two elements). If this succeeds the result is converted to a list which is inserted between the two other lists after which application of the associative operator is retried. Finally, the last clause defines the catch-all rule for associative operators. Note that in case of a non-confluent specification of an associative operation the definition of `split` and `append` determine which of the normal forms of a term is returned by the generated code.

The `assoc_flat` predicate flattens the arguments of an associative operator (case 3).

The code generated for the specification of natural numbers and finite sets of natural numbers given in Section 3.1 is the following:

```
/* >>> Equations                                          <<< */

plus(N, [zero, N_List])                                  /* [1] */
   :- assoc_arg(plus, N, N_List).
plus(Res, [succ(N1), N2_List])                           /* [2] */
   :- assoc_arg(plus, N2, N2_List),
      assoc_flat(plus, [N1, N2], List1),
      assoc_all(plus, List1, Temp1),
      succ(Res, Temp1).

mult(Res, Input)                                         /* [3] */
   :- assoc_decomp(mult, Input, N, [zero]),
      zero(Res).
```

```
mult(Res, Input)                                          /* [4] */
   :- assoc_decomp(mult, Input, N1, [succ(N2)]),
      assoc_flat(mult, [N1, N2], List1),
      assoc_all(mult, List1, Temp1),
      assoc_flat(plus, [Temp1, N1], List2),
      assoc_all(plus, List2, Res).

add(Res, N, add(N, S))                                    /* [5] */
   :- add(Res, N, S).
add(Res, N1, add(N2, S))                                  /* [6] */
   :- add(Temp1, N1, S),
      add(Res, N2, Temp1).

union(S, [empty, S_List])                                 /* [7] */
   :- assoc_arg(union, S, S_List).
union(Res, [add(N, S1), S2_List])                         /* [8] */
   :- assoc_arg(union, S2, S2_List),
      assoc_flat(union, [S1, S2], List1),
      assoc_all(union, List1, Temp1),
      add(Res, N, Temp1).

/* >>> Catch-all                                            <<< */

zero(zero).
succ(succ(X1), X1).
empty(empty).
add(add(X1, X2), X1, X2).
```

The generation of code for an algebraic specification with associativity is an extension of the method described in Section 2.4 for standard algebraic specifications. The only difference as far as typechecking the specification is concerned is the flattening of terms which is done in this phase. Extensions of the two steps defined in Section 2.4.1 give the code generation of one Horn-clause for each equation:

1. The right-hand side of each equation is again decomposed in a list of predicates and a translated term. The predicates `assoc_flat` and `assoc_all` are used to reduce terms with associative operators to a normal form.
2. To obtain matching modulo associativity the left-hand side not only contributes to the conclusion of the Horn-clause, but it also gives predicates with `assoc_decomp` and `assoc_arg` in the conditions of the Horn-clause.

The changes in both steps are now described in more detail.

In the analysis of the right-hand side of the equation (step 1) the only change is the case of a term having an associative operator as head symbol:

- `t` ≡ `f(t1, t2, ..., tn)` with $n \geq 2$ and *assoc*(`f`):
  Let $L_1, L_2, \cdots, L_n$ be the lists of predicates generated for the subterms `t1, t2, ···, tn`, and let `T1, T2, ···, Tn` be the corresponding translated terms. The list of predicates for `t` is a concatenation of the lists $L_1, L_2, \cdots, L_n$, and the predicates `assoc_flat(f, [T1, T2, ..., Tn], List)` and `assoc_all(f, List, Var)` added at the end of it. The variables `List` and `Var` are both fresh Prolog variables, and `Var` is the translated term for `t`.

The treatment of the left-hand side of the equation is not as easy as in Section 2.4.1. We have to distinguish clearly between the handling of the head symbol of the left-hand side and the handling of its arguments:

2a. A corresponding Prolog term has to be generated for each argument. We call this Prolog term the *matching term* of the argument. The variables in the arguments will be represented by their corresponding Prolog variables. We have to take care that their value after using Prolog unification and resolution of generated `assoc_decomp` and `assoc_arg` predicates is the term which the original variable would have had after matching modulo associativity.

2b. The conclusion of the Horn-clause is constructed from the head symbol of the left-hand side and the matching terms of its arguments. If the head symbol is an associative operator the matching terms of

the arguments have to be put in a Prolog list as the second argument of the predicate.

In case 2a the matching term in the standard code generation process was simply created by changing all variables into their corresponding Prolog variable. Now, we define for each term the matching term and a list of `assoc_decomp` and `assoc_arg` predicates using induction on the complexity of the term `t`:

- `t` ≡ `x`:
  The list of predicates is empty and the matching term is the Prolog variable that corresponds to the variable `x`.
- `t` ≡ `c`:
  The matching term of a constant `c` is `c` and the list of predicates is empty.
- `t` ≡ `f(t1, t2, ..., tn)` with `n` ≥ 1 and not *assoc*(`f`):
  The matching term of `t` is `f(T1, T2, ..., Tn)`, where `T1, T2, ···, Tn` are the matching terms of `t1, t2, ···, tn`. The list of predicates for `t` is simply a concatenation of the lists for `t1, t2, ···, tn`.
- `t` ≡ `f(t1, t2, ..., tn)` with `n` ≥ 2 and *assoc*(`f`):
  Now we create the matching term and the list of predicates for the arguments `a` ≡ `[t1, t2, ..., tn]` and the associative operator `f` as follows:
  - `a` ≡ `[t1, t2, ..., tn]` with `n` ≥ 2:
    Let the matching term of `[t2, ..., tn]` be `Tr`, and let the list of predicates be L_r.
    - If `t1` is a variable `x` and the Prolog variable which corresponds to `x` is `X`, then the list of predicates for `a` is L_r with `assoc_decomp(f, Var, X, Tr)` added at the end. Here `Var` is a fresh Prolog variable which is also the matching term of `a`.
    - If `t1` is not a variable and the matching term for `t1` is `T1` and the list of predicates is L_1, then the list of predicates for `a` is a concatenation of L_r and L_1. The matching term for `a` is `[T1| Tr]`.
  - `a` ≡ `[t1]`:
    - If `t1` is a variable `x`, the list of predicates for `a` is `assoc_arg(f, X, Var)` where `X` is the Prolog variable which corresponds to `x`. The matching term of `a` is a fresh Prolog variable `Var`.
    - If `t1` is not a variable and the matching term for `t1` is `T1` and the list of predicates is L_1, then the list of predicates for `a` is L_1 and the matching term for `a` is `[T1]`.

If `Res` is the translated term from the right-hand side of the equation the conclusion of the Horn-clause (step 2b) is generated from the left-hand side `t` as follows:

- `t` ≡ `c`:
  If the left-hand side is a constant `c` the conclusion is `c(Res)`.
- `t` ≡ `f(t1, t2, ..., tn)` with `n` ≥ 1 and not *assoc*(`f`):
  The conclusion is `f(Res, T1, T2, ..., Tn)`, where the `Ti` are the terms which correspond to the arguments `ti` as defined in step 2a.
- `t` ≡ `f(t1, t2, ..., tn)` with `n` ≥ 2 and *assoc*(`f`):
  The conclusion is `f(Res, Tr)`, where the `Tr` is the term which corresponds to the list of arguments `[t1, t2, ..., tn]` as defined in step 2a.

As the catch-all rule for associative operators is already incorporated in the definition of `assoc_all` these rules only need to be generated for non-associative functions.

Finally, the decomposition of input terms to Prolog questions and the handling of conditional equations is similar to the way it was done in Section 2.4.

4. Algebraic specifications with lists

4.1. Example

As an example of a specification with lists, we present a specification in which natural numbers are modeled as non-empty lists of digits, and (finite) sets of natural numbers as lists of natural numbers. The number 3524 is, for instance, represented as `nat([3, 5, 2, 4])`. The set {12, 336} is represented as `set([nat([1, 2]), nat([3, 3, 6])])` and the empty set as `set([])`. Equation `[1]` serves to remove leading zeros of numbers. Identical elements in sets are removed in `[13]`, and the irrelevance of the order of elements is expressed in `[14]`.

```
module Natural-Numbers
begin
   exports
   begin
      sorts DIGIT, NAT, SET
      functions
         0     :            -> DIGIT
            ...
         9     :            -> DIGIT
         nat   : DIGIT+     -> NAT
         succ  : NAT        -> NAT
         set   : NAT*       -> SET
         union : SET # SET  -> SET
   end
   variables
      k, k1, k2  : -> DIGIT+
      m          : -> DIGIT*
      n          : -> NAT
      x1, x2     : -> NAT+
      y1, y2, y3 : -> NAT*
   equations
      [1]    nat([0, k]) = nat([k])
      [2]    succ(nat([m, 0]))  = nat([m, 1])
                ...
      [10]   succ(nat([m, 8]))  = nat([m, 9])
      [11]   succ(nat([9]))     = nat([1, 0])
      [12]   succ(nat([k1, 9])) = nat([k2, 0])
                when succ(nat(k1)) = nat(k2)
      [13]   set([y1, n, y2, n, y3]) = set([y1, n, y2, y3])
      [14]   set([x1, x2])           = set([x2, x1])
      [15]   union(set(y1), set(y2)) = set([y1, y2])
end Natural-Numbers
```

4.2. Definitions

An *algebraic specification with lists* $<\Sigma, E>$ consists of an extended signature Σ and a set of (possibly conditional) equations E over Σ. An *extended signature* $\Sigma \equiv <S_\Sigma, F_\Sigma>$ contains a set of sort symbols S_Σ and a set of function symbols F_Σ. Unlike the typing function in standard signatures, the typing function in extended signatures may also use "starred" and "plussed" sorts in its *input type*. Hence, the implicit typing function is now defined from F_Σ to $\{S, S^*, S+ \mid S \in S_\Sigma\}^* \times S_\Sigma$. The sorts of the form `S*` and `S+` are called *iterated sorts*.

To prevent the user of the specification formalism from changing the semantics of iterated sorts we forbid their use as the output sort of functions. It is also forbidden to use them as the sort of any equation in the specification. This is done because the names `S*` and `S+` suggest that these sorts contain only iterations of elements of sort `S` and none of these lists can be identified. To illustrate this, we could remove the sort `NAT` from the above specification and replace it by `DIGIT+`. The declaration of the function `nat` would

then disappear and equation [1] would become

[1'] [0, k] = [k]

meaning that a list of digits starting with a zero is identical to the same list without the zero at the beginning. The semantics of DIGIT+ would have changed and if one wants to use this plussed sort elsewhere one has to be aware of these changes. We believe that this is not desirable.

Given an extended signature $\Sigma = <S_\Sigma, F_\Sigma>$ and a set of typed variables X we can define the set of terms over such a signature. As can be seen from the above example variables are allowed to be of an iterated sort. In the sequel we will use S, S1, S2, ··· to denote the usual sorts of the specification (the elements of S_Σ) and T, T1, T2, ··· to denote possibly iterated sorts. We define the sets $T_\Sigma^S(X)$, $T_\Sigma^{S*}(X)$, $T_\Sigma^{S+}(X)$ of terms of respectively sort S, starred sort S*, and plussed sort S+. $T_\Sigma^S(X)$ is the smallest set such that:

- x is an element of $T_\Sigma^S(X)$ for each variable x: -> S.
- c is an element of $T_\Sigma^S(X)$ for each (constant) function c: -> S.
- For each function f: T1 # T2 # ... # Tn -> S with n ≥ 1 and for all terms t1 ∈ $T_\Sigma^{T1}(X)$, t2 ∈ $T_\Sigma^{T2}(X)$, ···, tn ∈ $T_\Sigma^{Tn}(X)$ we define f(t1, t2, ..., tn) to be an element of $T_\Sigma^S(X)$.

$T_\Sigma^{S*}(X)$ is the set such that:

- x is an element of $T_\Sigma^{S*}(X)$ for each variable x: -> S* or x: -> S+.
- The list [t1, t2, ..., tn] where n ≥ 0 is an element of $T_\Sigma^{S*}(X)$ if ti ∈ $T_\Sigma^S(X)$ holds for all 1 ≤ i ≤ n, or ti is a variable of type S* or S+.

$T_\Sigma^{S+}(X)$ is the set such that:

- x is an element of $T_\Sigma^{S+}(X)$ for each variable x: -> S+.
- [t1, t2, ..., tn] with n ≥ 1 is an element of $T_\Sigma^{S+}(X)$ if ti ∈ $T_\Sigma^S(X)$ holds for all 1 ≤ i ≤ n, or ti is a variable of type S* or S+. At least one of the ti should not be a variable of type S*.

The set of all terms over regular sorts (i.e., excluding terms of iterated sorts) is denoted by $T_\Sigma(X) \equiv \bigcup_{S \in S_\Sigma} T_\Sigma^S(X)$ and the set of all terms (including lists) is denoted by $T_\Sigma^{*+}(X)$.

Note that it is no longer possible to assign a unique type to each term. For each sort S $T_\Sigma^{S+}(X) \subset T_\Sigma^{S*}(X)$ and the empty list [] is an element of $T_\Sigma^{S*}(X)$ for any S*. On the other hand, all lists can only occur within a context which can be used to disambiguate the type of a term. In algebraic specifications with lists we could also allow overloading of function symbols and still assure unique typing of terms which are not lists. Now, functions with identical names and overlapping input types should be forbidden. Input types are *overlapping* if they consist of the same number of (regular or iterated) sorts and for each pair of corresponding positions the following holds:

- identical sorts S appear at both positions, or
- a type S+ in one position corresponds to S* or S+ at the other position, or
- a type S* corresponds to S+ or S* or another starred sort S1*.

The set of all *(possibly conditional) equations E* consists, once again, of equations of which the types of left-hand side and right-hand side are identical. As stated before, it is forbidden to construct equations over iterated sorts. In short, the set of unconditional equations of type S ∈ S_Σ is $Eq^S \equiv T_\Sigma^S(X) \times T_\Sigma^S(X)$ and the set of all (possibly conditional) equations E is a subset of $Eq \times Eq^*$, where $Eq \equiv \bigcup_{S \in S_\Sigma} Eq^S$.

An assignment $\rho : X \rightarrow T_\Sigma^{*+}(X)$ is a function which assigns to each variable a term over the given extended signature Σ. The type of ρ(x) has to be equal to the type of x if x is of type S or S+. For x: -> S* the type of ρ(x) should be S* or S+. The extension of $\rho : X \rightarrow T_\Sigma^{*+}(X)$ to the complete set of terms $(\rho : T_\Sigma^{*+}(X) \rightarrow T_\Sigma^{*+}(X))$ is defined by:

- For each (constant) function c: -> S we define ρ(c) ≡ c.
- For all functions f: T1 # T2 # ... # Tn -> S with n ≥ 1 ρ is defined by ρ(f(t1, t2, ..., tn)) ≡ f(ρ(t1), ρ(t2), ..., ρ(tn)).
- For the empty list we define: ρ([]) ≡ [].
- Finally, for non-empty lists [t1, t2, ..., tn] with n ≥ 1 which are an element of $T_\Sigma^{S*}(X)$ or $T_\Sigma^{S+}(X)$ suppose ρ([t2, ..., tn]) = [s1, s2, ..., sm] with m ≥ n−1. There are two possibilities:

- If ρ(t1) is an element of $T^{S}_{\Sigma}(X)$, or a variable of type S* or S+ then ρ([t1, t2, ..., tn]) ≡ [ρ(t1), s1, s2, ..., sm]
- If ρ(t1) is an element of $T^{S*}_{\Sigma}(X)$ or $T^{S+}_{\Sigma}(X)$ of the form [u1, u2, ..., uk] with k ≥ 0 we define ρ([t1, t2, ..., tn]) ≡ [u1, u2, ..., uk, s1, s2, ..., sm]

4.3. Semantics

The semantics of an algebraic specification with lists <Σ, *E*> is defined by giving a translation of the specification to an algebraic specification with associativity <Σ', *E'*, *assoc*>. For each sort S of which an iterated variant occurs in the original specification, new sorts S-star and S-plus are added. Furthermore, for all such sorts S we add standard functions for the empty list (empty-S), injections from sort S into S-plus and from S-plus into S-star, and concatenation functions for lists. To define the semantics of these functions some extra equations are necessary. The following (parameterized) specification shows how this is done:

```
module Lists
begin
   parameters
   Sort begin
      sorts S
   end Sort
   exports
   begin
      sorts S-plus, S-star
      functions
         inj     : S                -> S-plus
         c-pp    : S-plus # S-plus -> S-plus {assoc}
         empty-S :                  -> S-star
         inj     : S-plus           -> S-star
         c-ps    : S-plus # S-star -> S-plus
         c-sp    : S-star # S-plus -> S-plus
         c-ss    : S-star # S-star -> S-star {assoc}
   end
   variables
      sp, sp1, sp2 : -> S-plus
      ss           : -> S-star
   equations
      [1]      c-ps(sp, empty-S)    = sp
      [2]      c-ps(sp1, inj(sp2)) = c-pp(sp1, sp2)
      [3]      c-sp(empty-S, sp)    = sp
      [4]      c-sp(inj(sp1), sp2) = c-pp(sp1, sp2)
      [5]      c-ss(ss, empty-S)          = ss
      [6]      c-ss(empty-S, ss)          = ss
      [7]      c-ss(inj(sp1), inj(sp2)) = inj(c-pp(sp1, sp2))
end Lists
```

The typing of function symbols and variables has to be changed such that all occurrences of S* and S+ are, respectively, replaced by S-star and S-plus. Finally, all terms which occur in the equations *E* of the original specification have to be translated to terms over the new specification with associativity. The translation $\tau: T^{*}_{\Sigma}(X) \rightarrow T_{\Sigma'}(X)$. is given by defining the projections τ_T for all sorts T. The translation τ_S is defined such that for all terms t ∈ $T^{S}_{\Sigma}(X)$: $\tau_S(t) \in T^{S}_{\Sigma'}(X)$:

- $\tau_S(x) \equiv x$ for each variable x: -> S.
- $\tau_S(c) \equiv c$ for each (constant) function c: -> S.
- $\tau_S(f(t1, t2, ..., tn)) \equiv f(\tau_{T1}(t1), \tau_{T2}(t2), ..., \tau_{Tn}(tn))$ for each function f: T1 # T2 # ... # Tn -> S with n ≥ 1.

For all terms t ∈ $T^{S*}_{\Sigma}(X)$ the translation τ_{S*} is defined such that $\tau_{S*}(t) \in T^{S\text{-}star}_{\Sigma'}(X)$:

- $\tau_{S*}(x) \equiv x$ for each variable x: -> S*.

- τ_{S*}(x) ≡ inj(x) for each variable x: -> S+.
- τ_{S*}([]) ≡ empty-S.
- τ_{S*}([t1, t2, ..., tn]) ≡ inj(τ_{S+}([t1, t2, ..., tn])) if at least one of the ti ($1 \le i \le n$) is not a variable of type S*.
- τ_{S*}([x1, x2, ..., xn]) ≡ c-ss(x1, τ_{S*}([x2, ..., xn])) if for all $1 \le i \le n$ Xi: -> S*.

For all terms $t \in T_\Sigma^{S+}(X)$ the translation τ_{S+} is defined such that $\tau_{S+}(t) \in T_{\Sigma'}^{S\text{-plus}}(X)$:

- τ_{S+}(x) ≡ x for each variable x: -> S+.
- τ_{S+}([t1, t2, ..., tn]) ≡ c-ps(inj(τ_S(t1)), τ_{S*}([t2, ..., tn])) if $t1 \in T_\Sigma^S(X)$ and $n \ge 1$.
- τ_{S+}([x, t2, ..., tn]) ≡ c-ps(x, τ_{S*}([t2, ..., tn])) if x: -> S+ and $n \ge 1$.
- τ_{S+}([x, t2, ..., tn]) ≡ c-sp(x, τ_{S+}([t2, ..., tn])) if x: -> S* and $n \ge 2$.

4.4. Implementation in Prolog

It turns to be impossible to use the translation semantics for * and + given in the previous section directly in an implementation. Problems occur in equations in which variables of starred sorts occur. The translation of equation

```
[2]     succ(nat([m, 0])) = nat([m, 1])
```

of the example of Section 4.1 would give:

```
[2]     succ(nat(c-sp(m, c-ps(inj(0), empty-DIGIT))))
         = nat(c-sp(m, c-ps(inj(1), empty-DIGIT))).
```

The translation of the term succ(nat([0])) which is succ(nat(c-ps(inj(0), empty-DIGIT))) cannot match the left-hand side of the translated equation. The same holds for the translation of succ(nat([1, 0])) which is succ(nat(c-ps(inj(1), inj(c-ps(inj(0), empty-DIGIT))))). Even reducing both sides of the translated equation [2] using the equations of module Lists as given in Section 4.3 gives no solution:

```
[2]     succ(nat(c-sp(m, inj(0)))) = nat(c-sp(m, inj(1))).
```

A possible solution would be to double each equation in which a variable of a starred sort occurs, into an equation for the empty case and an equation with the variable of the corresponding plussed sort. In this example this would give:

```
[2a]    succ(nat([0]))     = nat([1])
[2b]    succ(nat([m, 0])) = nat([m, 1])
```

where m: -> DIGIT+.

It is much easier to translate lists into Prolog lists. The only problem is the head-tail-like decomposition of lists in Prolog which makes it necessary to use the append predicate in the implementation of the more general lists as defined here. When rewriting with lists the following changes are relevant:

1. In the construction of legal terms given in Section 4.2 we forbid lists as arguments of a list. As a consequence, we have to be careful that no lists as arguments of lists occur during list construction. Hence, in the decomposition of the right-hand side we have to generate append predicates to join lists.
2. To match a given list with the left-hand side of an equation we must be able to split the given list in arbitrary parts. We also use the append predicate for this.

As an example we present the code generated for the example of Section 4.1:

```
/* >>> Equations                                          <<< */

nat(Res, ['0', K_Head| K_Tail])                        /* [1] */
   :- nat(Res, [K_Head| K_Tail]).

succ(Res, nat(Input))                                   /* [2] */
   :- append(M, ['0'], Input),
      '1'(Temp1),
      append(M, [Temp1], Temp2),
```

```
      nat(Res, Temp2).
   ...
succ(Res, nat(Input))                                    /* [10] */
   :- append(M, ['8'], Input),
      '9'(Temp1),
      append(M, [Temp1], Temp2),
      nat(Res, Temp2).
succ(Res, nat(['9']))                                    /* [11] */
   :- '0'(Temp1),
      '1'(Temp2),
      nat(Res, [Temp2, Temp1]).
succ(Res, nat(Input))                                    /* [12] */
   :- append([K1_Head| K1_Tail], ['9'], Input),
      nat(Temp1, [K1_Head| K1_Tail]),
      succ(Temp2, Temp1),
      Temp2 = nat([K2_Head| K2_Tail]),
      '0'(Temp3),
      append([K2_Head| K2_Tail], [Temp3], Temp4),
      nat(Res, Temp4).

set(Res, Input)                                          /* [13] */
   :- append(Y1, [N| I1], Input),
      append(Y2, [N| Y3], I1),
      append(Y2, Y3, Temp1),
      append(Y1, [N| Temp1], Temp2),
      set(Res, Temp2).
set(Res, Input)                                          /* [14] */
   :- append([X1_Head| X1_Tail], [X2_Head| X2_Tail], Input),
      append([X2_Head| X2_Tail], [X1_Head| X1_Tail], Temp1),
      set(Res, Temp1).

union(Res, set(Y1), set(Y2))                             /* [15] */
   :- append(Y1, Y2, Temp1),
      set(Res, Temp1).

/* >>> Catch-all                                            <<< */

'0'('0').
   ...
'9'('9').
nat(nat(X1), X1).
succ(succ(X1), X1).
set(set(X1), X1).
union(union(X1, X2), X1, X2).
```

In general, the code generation process is again an extension of the two steps described in Sections 2.4.1 and 3.4. So far, we added to each variable occurring in an equation a corresponding Prolog variable in the typechecking phase. To prohibit variables of a plussed sort from matching an empty list we add an expression `[Head| Tail]` to each such variable. Of course, the Prolog variables `Head` and `Tail` are different for each variable. So, instead of a list of variables with their corresponding Prolog variables we now generate a list of corresponding Prolog expressions.

In the decomposition of the right-hand side of the equation (step 1) the list of predicates and the translated term need to be defined only in case the term is a list:

- `t ≡ []`:
  The list of predicates to be generated is empty and the translated term is the empty list `[]`.
- `t ≡ [t1]`:
  - If `t1` is a variable `x` of an iterated sort then the translated term of `t1` is the expression which is associated to it in the typechecking phase. Hence, if `x` is of a starred sort it is the Prolog variable associated to `x`, and if `x` is of a plussed sort it is an expression of the form `[Head| Tail]`. In this case the generated list of predicates is empty.

- If `t1` is not a variable of an iterated sort and the translated term for `t1` is `T1` and the list of predicates is L_1 then the list of predicates for `[t1]` is L_1 and the translated term for `[t1]` is `[T1]`.

- `t` ≡ `[t1, t2, ..., tn]` with `n` ≥ 1:
  Let the translated term of `[t2, ..., tn]` be `Tr`, and let the list of predicates be L_r.
  - If `t1` is a variable `x` of an iterated sort and the Prolog expression which corresponds to `x` is `Tx` then the list of predicates for `t` is L_r with `append(Tx, Tr, Var)` added at the end. Here `Var` is a fresh Prolog variable which is also the translated term of `t`.
  - If `t1` is not a variable of an iterated sort and the translated term for `t1` is `T1` and the list of predicates is L_1 then the list of predicates for `t` is a concatenation of L_r and L_1. The translated term for `t` is `[T1| Tr]`.

In handling the left-hand side of an equation (step 2) we only need to describe what has to be done if lists occur in the arguments of the left-hand side. Remember, that it is forbidden to construct equations over iterated sorts and therefore lists can never occur as the left-hand side of any equation. This should, by the way, be checked while typechecking the specification. The construction of the matching term and the list of predicates which take care of matching modulo lists is identical to the construction of the translated term given above and the list of predicates for terms in the right-hand side of equations.

The handling of conditional equations is similar to what is done in Section 2.4.2. The only difference in the treatment of input is that we cannot handle terms in which variables of iterated sorts occur. These terms are simply forbidden in the input.

5. Applications

As mentioned in the introduction adding lists and associative functions to an algebraic specification formalism is a necessary step in combining the formalisms ASF and SDF. Several specifications have been written in (preliminary versions of) the combination of these formalisms:

- The typechecker for a sublanguage of ML (Mini-ML) in [Hen89].
- The static and dynamic semantics of the toy language PICO [Chapter 9 of BHK89].
- The typechecker and interpreter for a simple programming language ASPLE, the dynamic semantics of the machine language SML, and a compiler from ASPLE to SML [Meu88].

An SDF-specification is a combination of the abstract syntax (in the form of a signature) and the concrete syntax (in the form of BNF-rules, read in reverse order) of a language. Hence, each SDF-specification implicitly defines a lexical analyzer and a parser for the language it defines. A specification in the combined ASF/SDF formalism can be reduced to an algebraic specification in ASF as follows:

- replace each SDF-definition by its underlying signature;
- parse all equations using the grammar defined by the SDF-definitions and replace each equation by the result of this parse (the result is an equation containing terms in prefix form instead of arbitrary strings).

As an example of the combination of both formalisms we give, once again, a specification of the natural numbers and (finite) sets of natural numbers:

```
module Natural-Numbers
begin
   exports
   begin
      sorts NAT, SET
      lexical syntax
         [ \t\n\r]            -> LAYOUT
         [0-9]+               -> NAT
      context-free syntax
         succ "(" NAT ")"     -> NAT
         "{" {NAT ","}* "}"   -> SET
         SET "+" SET          -> SET {assoc}
         "(" SET ")"          -> SET {bracket}
   end
   variables
      k, k1, k2  -> CHAR+
```

```
    m           -> CHAR*
    n           -> NAT
    x1, x2      -> NAT+
    y1, y2, y3 -> NAT*

equations

    [1]       nat(0 k) = nat(k)

    [2]       succ(nat(m 0))  = nat(m 1)
    [3]       succ(nat(m 1))  = nat(m 2)
    [4]       succ(nat(m 2))  = nat(m 3)
    [5]       succ(nat(m 3))  = nat(m 4)
    [6]       succ(nat(m 4))  = nat(m 5)
    [7]       succ(nat(m 5))  = nat(m 6)
    [8]       succ(nat(m 6))  = nat(m 7)
    [9]       succ(nat(m 7))  = nat(m 8)
    [10]      succ(nat(m 8))  = nat(m 9)
    [11]      succ(9)         = 10
    [12]      succ(nat(k1 9)) = nat(k2 0)
                 when succ(nat(k1)) = nat(k2)

    [13]      {y1, n, y2, n, y3} = {y1, n, y2, y3}
    [14]      {x1, x2}           = {x2, x1}

    [15]      {y1} + {y2} = {y1, y2}

end Natural-Numbers
```

An SDF-specification consists of at most five components (four of which can be found in the above example):

- The **sorts** section contains the names of the non-terminals of the grammar which can be derived from an SDF-specification. These names are also the names of the sorts in the derived signature.
- The **lexical syntax** section incorporates the specification of a regular grammar which is used to generate a lexical analyzer. It contains one or more function declarations each consisting of a regular expression and a result sort. The sort `LAYOUT` is predefined and functions with output sort `LAYOUT` do not contribute to the derived regular grammar or the derived signature. Character classes like `[0-9]` and `[a-zA-Z]` are used to abbreviate the lexical definition. A sort or character class followed by a * stands for zero or more repetitions of the sort. A + stands for one ore more repetitions.
- The context-free grammar can be extracted from the **context-free syntax** section. Each rule in this section (except from the functions which are furnished with the **bracket**-attribute) adds information to the derived signature. The notations `{SORT "t")*` and `{SORT "t"}+` are used to denote lists of elements of `SORT` separated by the symbol `"t"`. By extending signatures with * and + as described in Section 4 each rule will correspond to exactly one function in the derived signature.
- In the **priorities** section the precedence of the rules in the **context-free syntax** section can be specified in order to disambiguate ambiguous sentences. In the above examples this section is absent.
- The **variables** section defines the variables which may be used in the **equations** section.

6. Conclusions

As mentioned in the introduction, lists and associative functions do not add expressive power to an algebraic specification formalism, but especially the use of lists gives more elegant specifications which are easier to read. Both features are implemented in the ASF system [Hen88] using the given algorithms. To declare a function to be associative evidently causes an increase in execution time as well as in generation time. However, rewriting modulo associativity gives a more powerful implementation for specifications with associative functions. The implementation of lists is reasonably fast as long as the head-tail-like decomposition of lists in Prolog can be used. From the specification point of view other decompositions of lists are desirable and it is very useful to have an implementation for them. An elegant combination of term rewriting and string rewriting as proposed here is desirable.

Acknowledgements

Jan Heering, Paul Klint, and Emma van der Meulen commented on earlier versions of this paper. Their remarks improved the readability considerably. Research in this topic was encouraged by discussions with the members of the GIPE-project: Hans van Dijk, Jan Heering, Paul Klint, Wilco Koorn, Emma van der Meulen, Jan Rekers, and Pum Walters.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint (eds.), *Algebraic Specification*, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley (1989).

[BGS88] H. Bertling, H. Ganzinger, and R. Schäfers, "CEC: a system for the completion of conditional equational specifications," pp. 249-250 in *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan, and J.-P. Jouannaud, Lecture Notes in Computer Science **308**, Springer-Verlag (1988).

[DE84] K. Drosten and H.-D. Ehrich, "Translating algebraic specifications to Prolog programs," Informatik-Bericht Nr. 84-08, Technische Universität Braunschweig (1984).

[EY87] M.H. van Emden and K. Yukawa, "Logic programming with equations," *Journal of Logic Programming* **4**, pp. 265-288 (1987).

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," pp. 52-66 in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM (1985).

[GKKMMW88] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler, "An introduction to OBJ3," pp. 258-263 in *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan, and J.-P. Jouannaud, Lecture Notes in Computer Science **308**, Springer-Verlag (1988).

[GM87] J.A. Goguen and J. Meseguer, "Remarks on remarks on many-sorted equational logic," *Sigplan Notices* **22**(4), pp. 41-48 (1987).

[HHKR] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism SDF – reference manual," Centre for Mathematics and Computer Science, Amsterdam, to appear.

[Hen88] P.R.H. Hendriks, "Automatic implementation of algebraic specifications," pp. 83-94 in *Conference Proceedings of Computing Science in the Netherlands, CSN'88* **1**, SION (1988).

[Hen89] P.R.H. Hendriks, *Typechecking Mini-ML* (1989), Chapter 7 in [BHK89]. Short version: "Type-checking mini-ML: an experience with user-defined syntax in an algebraic specification," *Conference Proceedings of Computing Science in the Netherlands, CSN'87*, pp. 21-38, SION (1987).

[Jan86] M. Jantzen, "Confluent string rewriting and congruences," *Bulletin of the European Association for Theoretical Science EATCS*(28), pp. 52-72 (1986).

[Kap87] S. Kaplan, "Simplifying conditional term rewriting systems: unification, termination and confluence," *Journal of Symbolic Computation* **4**, pp. 295-334 (1987).

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

[Meu88] E.A. van der Meulen, "Algebraic specification of a compiler for a language with pointers," Report CS-R8848, Centre for Mathematics and Computer Science, Amsterdam (1988).

[PWBBP85] F. Pereira, D. Warren, D. Bowen, L. Byrd, and L. Pereira, *C-Prolog User's Manual, Version 1.5*, SRI International, Menlo Park, California (1985).

[RC88] T. Rush and D. Coleman, "Architecture for conditional term rewriting," pp. 266-278 in *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan, and J.-P. Jouannaud, Lecture Notes in Computer Science **308**, Springer-Verlag (1988).

[Wie87] F. Wiedijk, "Termherschrijfsystemen in Prolog," Rapport P8704, University of Amsterdam (1987), [in Dutch].