# On Equal Terms

Implementing Algebraic Specifications

H.R. Walters

On Equal Terms

# On Equal Terms

## Implementing Algebraic Specifications

Academisch Proefschrift
ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van Rector Magnificus
Prof. dr. P.W.M. de Meijer,
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse kerk, ingang Singel 411, hoek Spui),
op 14 juni 1991, 15.00 uur,

door

Humphrey Robert Walters
geboren te Hilversum

Promotor: prof. dr. P. Klint.

Faculteit: Wiskunde en Informatica.

## Acknowledgments

First and foremost I would like to thank my promotor Paul Klint, who over the years has supported me with advice and countless suggestions. Also, he has spent a considerable amount of time proofreading all manuscripts of this thesis.

Then I would like to thank my colleagues, who have listened to me patiently while my ideas took form, supplying me with their welcome insight. They are (in no particular order): Jan Rekers, Paul Hendriks, Emma van der Meulen, Wilco Koorn, Sjouke Mauw, Freek Wiedijk, Hans Mulder and Gert Veltink.

In particular, I would like to thank (in arbitrary order): Jan Heering and Wiet Bouma, who helped with Chapter two; Karin Vos, who helped with Chapter one; Casper Dik, who is co-author of a part of Chapter One, and who did most of the implementation of the system described in that chapter, and Piet Rodenburg, who helped with Chapter four.

Finally I would like to thank the referees for their willingness to review this thesis: Prof. Dr. H.A. Partsch, Prof. Dr. J.A. Bergstra and Dr. J.C.M. Baeten.

This thesis was written while I was employed at the Programming Research Group of the University of Amsterdam. I am grateful for the support and the use of facilities that have been given to me.

"Enten:
Het uit een boom verwijderen van delen, en deze door andere boomdelen met een ontwikkeld oog vervangen."

Vrij naar van Dale

Notes:

Partial support for this research was received from the European Communities under ESPRIT project 2177 (GIPE II - Generation of Interactive Programming Environments II)

Chapter 2 has been published (as part of Chapter 5: 'Implementing algebraic specifications', by L.G. Bouma and H.R. Walters,) in *Algebraic Specification*, J.A. Bergstra, J. Heering and P. Klint (eds.), pp. 199—282, The ACM press in cooperation with Addison-Wesley, 1989, and is reprinted with kind permission of Addison-Wesley.

Chapter 4 has been published in abbreviated form (under the title 'Hybrid implementations of algebraic specifications') in *Algebraic and Logic programming*, H. Kirchner and W. Wechler (eds.), pp. 40—54, LNCS 46 3, Springer Verlag, 1990.

Chapters 1 and 3 will be submitted for publication elsewhere.

# Contents

# Introduction

In recent years algebraic specifications have been playing an increasingly important role in software design. The key feature leading to this development is *abstraction*.

In algebraic specifications several levels of abstraction can be distinguished, each providing its specific contribution to software design.

1. The first abstraction level is the division of a problem in parts. Defining a modular structure can be a first step towards solving a problem.

2. A second level of abstraction consists of defining a signature for the problem at hand. This means that the distinct classes of objects, and the functions, with corresponding argument and result classes, are identified. Software tools are available which can verify the internal consistency of such signatures.

3. The third level of abstraction is the total algebraic specification. A meaning is given to functions in the signature by supplying equalities between those functions. This can be looked upon as formulating desired properties. Note that, in general, a high degree of abstraction is still achieved:

    - The actual objects concerned are not uniquely identified. Any class of objects that has the indicated properties (according to a certain interpretation), can be used as a model of the specification. In particular, one does not indicate whether, and if so, how objects can be represented in a computer.
    - How the indicated properties are achieved is not described. This is in sharp contrast with, for instance, algorithmic programming languages, where one describes how a result is achieved, but not what properties this result has. Note that this particular advantage of algebraic specifications is achieved only to a small degree, when restrictions are being imposed on them for the benefit of efficient prototyping.

    It is not necessary to represent the objects before calculations pertaining to them can be made on a computer. It is possible to consider consequences of the given equations, which will apply to all models. This is the domain of 'Theorem Proving'.

Unfortunately it is impossible – or not practical – to perform all calculations concerning a specification in this manner. Sometimes a fact does not follow from the given equations, but is nevertheless true in a particular relevant model; sometimes verification of a fact, merely using the given equations, takes much time, whereas it follows immediately in a given model.

4. In accordance with the aforementioned, it is necessary to impose restrictions on the class of models one wants to consider, and on the equations one wants to allow. Obviously many approaches exist, all leading to distinct areas of research.

   This final level of abstraction can not be seen as the implementation level. We are still at a formal level, but now we create the possibility to perform calculations, using suitable software.

In this thesis we will mainly be looking at the last level of abstraction. We will restrict ourselves to equational theories (the equations are universally quantified, possibly conditional equalities) and we only consider initial models. Such models can be characterized by two special properties:

- The model only contains objects which can be expressed in the given signature.
- All equational properties of the model can be deduced from the given equations.

Note that the disadvantages mentioned under 3, above, have been dealt with to a large degree. Since all objects can be expressed in the given signature, they can be stored in a computer using trees, and since all properties are deducible, in an equational theory they can be deduced automatically by way of tree replacement.

A problem still remains with this second point. Although equational properties can be deduced automatically, an arbitrary assumption can not necessarily be verified automatically. In other words, in order to verify some fact we can try to deduce it, but as long as we have not yet succeeded, we do not know whether we will ever succeed, or whether it is, in fact, not true.

In order to solve this last problem we will restrict ourselves even further. We will interpret algebraic specifications as term rewriting systems, which means that equations can only be applied in one direction, and we will only consider term rewriting systems which are **confluent** and **terminating**, such that of all possible deductions only one has to be considered, and such that any deduction is finite.

Furthermore we will restrict ourselves in the class of facts we want to verify. We will only look at the semantic equality between objects, or, in the context of confluent, terminating term rewriting systems equivalent with this, finding the normal form of objects. The normal form is a unique object in a class of semantically equal object which uniquely identifies that class.

With these restrictions any 'fact' can be established or refuted in finite time.

Note that it is by no means obvious whether the imposed restrictions still allow for acceptable expressiveness. Fortunately this proves to be the case. For instance, in the framework sketched above one has the same expressive power as in most programming languages, where only input and output, and 'real-time' aspects are essentially different.

Note that, strictly speaking, termination is too strong a restriction here, since it can not be guaranteed in most programming languages. Partly for this reason, termination and confluence will not be enforced in practice, even though these properties are theoretically needed to establish certain results. In general it is impossible to decide automatically whether a specification is confluent or terminating.

When using term rewriting systems in the development of software, one can basically have two applications in mind:

- The desired functionality is tested with the help of a specification and possibly using a prototype. When the specification is acceptable, an implementation is coded (by hand) in some programming language, where the defined behavior is approached as close as possible. The specification is used as a tool, but there is no formal relation between the specification and the implementation.

- Using the specification the eventual implementation is generated automatically. Now, the specification is a building block instead of a tool. Clearly this approach is a special case of the former, but now two aspects become relevant: automatic generation and the efficiency of the implementation, because for the final product the efficiency of a building block is of much greater importance than that of the tools used.

In our research systems are being developed which mainly aim at the second approach. Based on a specification, integrated implementations are generated automatically.

Unfortunately, efficiency is a weak point of software based on term rewriting. If the speed of such software is compared to that of directly implemented software, a factor of hundreds to thousands should be expected.

Accordingly, in this thesis we will pay attention to techniques for increasing the efficiency of term rewriting implementations. This efficiency is determined by two quantities: the number of rewrite steps needed and the time spent on each rewrite step. The technique of hybrid specifications, discussed in chapters 1, 4 and 5, aims at reducing the number of reduction steps, while 'tree-scanners', discussed in chapters 1 and 3 reduce the time spent on each reduction step.

This research has been carried out in the context of the project GIPE (Generation of Interactive Programming Environments), which is sponsored by the EEC (ESPRIT). In this project the ASF+SDF system has been developed: a programming environment generator, which, given a formal specification, generates several integrated, interactive tools, such as, for instance, syntax-oriented editors.

In Chapter 1 of this thesis the Equation Manager is discussed. This is the component of the ASF+SDF system which generates term rewriting implementations. In this chapter we present a formal definition of conditional term rewriting, and we sketch the implementation as used in the Equation Manager. Then we look at three techniques used in the Equation Manager, which are not – or not in this fashion – used in other term rewriting implementations. Two of these three techniques will be discussed in detail in later chapters.

In Chapter 2 we present, in some more detail, the theory of term rewriting. Then we look at term rewriting implementations in Prolog, reviewing some existing literature. This results in a compact overview of various reduction strategies. Finally, we look at the efficiency of Prolog implementations.

The repeated analyses required to determine where, in a given term, a rule can be applied form the most time consuming part of term rewriting implementations. In Chapter 3 we will look at a technique to avoid a lot of this duplicate work by performing these analyses in a quasi-parallel fashion for all rules at once.

In Chapter 4, a technique is described which allows one to use programs written in conventional programming languages from within the term rewriting framework. In this fashion basic datatypes can be implemented using the underlying computer hardware, thereby skipping many rewrite steps, while still operating within the formal framework of algebraic specifications.

In Chapter 5 we will present an extended example of the method discussed in Chapter 4. Using this example, various measurements will be discussed.

Various chapters in this thesis have been published separately. As a consequence, a certain degree of overlap between chapters is unavoidable. Whenever overlap occurs in a section, its title is marked with the symbol $\bowtie$ to indicate that (part of) the contents already appeared in some form in an earlier chapter.

# Chapter 1

# An Equation Manager

## Incremental processing of conditional equations

We introduce the notion of conditional term rewriting and sketch the implementation of the Equation Manager (EQM), the component of the ASF+SDF system dealing with term rewriting. We look at three particular aspects of the EQM in more detail: the implementation of list sorts, the values of which are arbitrary-length lists of terms; tree matching automata, in which string scanning techniques are applied to tree pattern matching; and hybrid specifications, a technique for the integration of term rewriting systems with traditionally implemented functions.

## 1.1 Introduction

The GIPE/ASF+SDF system [28, 17] is a generator for interactive programming environments. Given the formal specification of some language, the system creates a programming environment consisting of a syntax-sensitive text editor, incremental scanners and parsers and possibly other tools such as type-checkers and interpreters.

The system is self-contained. Formal language specifications are also edited using generated tools. This particular application of the system is called the **meta environment**. It differs from normal environments only in the sense that special dependencies exist: changes in a language definition imply changes to all tools generated from that definition.

The most important feature of the system is **incrementality**. All changes are processed immediately, and every time the system is brought up to date. All work done previously, is reused as much as possible. This approach shortens and effectively cuts out the time consuming edit/compile cycle.

The specification formalism used is ASF + SDF. This is a combination of an algebraic specification formalism (ASF) and a syntax definition formalism (SDF).

- In ASF ([3]) modular, conditional equational specifications can be written. An ASF specification defines a number of sorts and a collection of functions over those sorts. Every function has a signature: the argument sorts, and the result sort. With these functions the collection of terms is defined, which corresponds one to one with the set of abstract syntax trees. An ASF specification defines equations over the algebra of terms.

  The major non-standard feature of ASF is the existence of list-sorts. Algebraically, a list-sort is a sort the values of which are lists of some parameter sort. These lists are associative (neither left- nor right-recursive). A list expression can be any string of parameter values, parameter variables or list variables. In Section 1.4.1 this is discussed in more detail.

- In SDF ([18]) arbitrary lexical and context-free grammars can be defined. The definition of lexical and context-free grammars, and implicitly that of the abstract syntax appear together in the same, algebraically oriented context. The notion of phyla (non-terminals) coincides with that of sorts and there is a one to one correspondence between grammar rules and function declarations. SDF allows for the definition of arbitrary context-free languages.

As mentioned, a key concept in ASF + SDF is modularity. A specification is a set of modules, each defining its own syntax and equations. A module can use the syntax and equations of another module by importing it. Observe that for a specification many different scanners, parsers and other tools need to be generated. There is no 'top' module, every module may be selected and then only syntax rules and equations visible in that module are applicable. In such a system, it is necessary to have a large degree of sharing. If a module imports another module, no copies should be made of the source text or of any of the tools generated for it.

Every module has two sections, a syntax section and an equations section. In the syntax section the lexical and context-free grammar are defined, variables are declared and imported modules are indicated. Lexical and context-free grammar rules, and variable declarations may be declared as hidden (i.e., local, invisible to importing modules). The effective grammar of a module is the grammar defined by all its rules (both the hidden and visible ones) plus the visible grammar of all imported modules. This grammar defines what terms in that module may look like. The grammar of (conditional) equations cannot be defined by the user. It is generated as an extension of the grammar of terms. The syntax of this syntax section is fixed. It is identical for all modules and it does not depend on the grammars being defined. The equation section may contain a number of tagged conditional equations. The tag is used for reference only (debugging and tracing). Every equation may have a list of conditions, each appearing as an equality or as an inequality. The equation is applicable if all conditions are met. In the system, specifications are implemented as term rewriting systems.

The syntax of the equation section is generated by adding a skeleton grammar to the grammar of terms. For example, for every sort `S` in a module the function declaration `S "=" S` $\rightarrow$ `Simple-Equ` defines what an untagged unconditional equation looks like.

To summarize, to every module three syntaxes are associated:

- The syntax of the first (syntax defining) section. This syntax is identical for all modules.
- The syntax of terms, which is defined by the first section.
- The syntax of the equation section. This is an automatically generated extension of the term syntax.

In Figure 1.1 an example specification is shown. The specification consists of three modules:

- In the module `Layout` the sort `LAYOUT` is defined. This sort has a special status in the system (strings produced by instances of this sort are ignored by the parser, and do not appear in the abstract syntax). The sort itself is pre-declared, but its grammar can be defined by the user. In our example all whitespace will be ignored.
- In the module `Items` arbitrary items are defined (identifiers starting with a capital, in this case). Even though instances of sort `ITEM` are lexical tokens, the system allows us to manipulate their contents. For this purpose a function is automatically generated for that sort (its name is the lower case version of the sort name). This function regards an item as a list of characters. Variables are declared of sort `CHAR` (pre-declared in the system) and sort `CHAR+` (lists of characters of at least one character long). For the purpose of illustration, a single equation is given, which states that items containing more than four symbols, are truncated.
- The module `Sets` defines finite sets of items. It defines the union and intersection operators. The predicate '`assoc`' indicates that those operators are associative. The intersection operator is given higher priority than the union operator. The predicate '`bracket`' indicates that the brackets are used only to indicate precedence, but that they will not appear in the abstract syntax. The module defines variables of sort `ITEM` (an `i` followed by arbitrarily many ones and twos) and of sort 'list of items' (an `x` followed by ones, twos and threes). Then six equations are given. Equation [1] states that a set contains no duplicate elements. Equation [2] defines set union as the concatenation of all elements (duplicates will be thrown away by equation [1]). The remaining equations define the intersection. The empty set always yields an empty intersection; a shared element is present in the intersection; otherwise the

elements still have to be checked against the rest of the set. Equation [6] has a condition, which states that the items `i1` and `i2` should not be equal, as indicated by the inequality. Clearly, this specification is not very efficient but it is only intended to serve as an example.

```
Module Layout
  exports
    lexical syntax
         [ \t\n] -> LAYOUT
Module Items
  imports Layout
  exports
    sorts ITEM
    lexical syntax
      [A-Z][A-Za-z0-9]* -> ITEM
    variables
      "[" [a-z] "]"  -> CHAR
      "[*]"          -> CHAR+
  equations
    [1] item([a] [b] [c] [d] [*]) = item([a] [b] [c] [d])
Module Sets
  imports Layout Items
  exports
    sorts SET
    context-free syntax
      "{" {ITEM ","}* "}" -> SET
      SET "∪" SET         -> SET {assoc}
      SET "∩" SET         -> SET {assoc}
      "(" SET ")"         -> SET {bracket}
    variables
      i [12]*  -> ITEM
      x [1-3]* -> {ITEM ","}*
  priorities
    "∪" < "∩"
  equations
    [1] {x1,i,x2,i,x3} = {i,x1,x2,x3}
    [2] {x1} ∪ {x2} = {x1,x2}
    [3] {} ∩ {x} = {}
    [4] {x} ∩ {} = {}
    [5] {i,x1} ∩ {i,x2} = {i} ∪ {x1}∩{x2}
    [6] i1 != i2
        ============================================
        {i1,x1}∩{i2,x2} = {i1,x1}∩{x2} ∪ {x1}∩{i2,x2}
```

*Figure 1.1.*

The term

```
{Tinker, Tailor, Soldier}
  ∩({Soldier,Spy,Spy}∪{Tinker,Soldier,Tinker,Spy})
```

is correctly normalized to

```
{ Tink, Sold }
```

The different syntaxes can clearly be seen in this example. The SDF part of each module – between '`Module`' and '`equations`' – has the same syntax. The ASF part of module `Items` consists of the skeleton syntax – in which for instance tags (`[1]`) and the '=' symbol are defined – and the syntax of Items. The syntax of the ASF part of module `Sets` consists of the same skeleton syntax, extended with that of sets (which also uses items).

In the ASF+SDF meta-environment, an editor may be associated with each syntax (or, to be more precise, an instantiation of the editor, because editors are parameterized with the current grammar). The syntax and equation editors are used to manipulate the specification. The term editor is used to create and modify terms in the defined syntax. If this syntax is the grammar of some language being defined (as opposed to for instance auxiliary notions) such an editor is precisely a syntax oriented text editor for that language. Consider, for example, a Pascal definition. A complete specification includes

- auxiliary modules such as `Booleans` and `Integers`;
- the module defining the Pascal syntax, which typically has sorts such as `PROGRAM`, `STATEMENT` and `DECLARATIONS`;
- modules defining tools such as a typechecker, an interpreter or a pretty printer.

The meta-environment has editors for all three parts of all modules (not all editors are always active, but conceptually they are present). The (normal) Pascal environment only has a Pascal editor. This editor is identical to the term editor for the Pascal syntax module. That is, a 'normal' environment corresponds to one component (for one module) in the meta-environment. The various tools defined in the specification are made available in a term editor. For instance, the term editor may have a button 'typecheck', which, when activated, will typecheck a term according to the specification of the typecheck function. Every term editor at least has an evaluate button which causes the system to calculate the normal form of a term.

The ASF+SDF meta-environment ([28]) has four major components.

- The module manager (MM) is the only component aware of the modular structure of specifications. It acts as an overseer: all components (in particular editors) report changes to the MM, which then determines what other components need to be informed. The MM also manages the top level of the user interface. This top level is extremely simple. It tells one which modules are in the specification and allows one to open an editor of any of the three kinds mentioned earlier (syntax, equations, terms) on any of those modules.

- The editors have already been discussed. Apart from the top level they constitute the entire user interface. Editors are 'unaware' of each other, they only communicate with the MM;
- The syntax manager (SM) incrementally maintains a database for all lexical and context free grammars. As mentioned earlier, the amount of sharing is maximal. This is done by creating a single scanner and parser which can accept all possible language constructs in the entire specification. When a text must be parsed given one particular grammar (say the grammar of a term in the Pascal syntax module) then a so called **selection** is performed on the scanner and parser, in which that module and all modules it imports are indicated. After that, only the subset of language constructs legal in terms of that module can be recognized (remember that only the MM has this information). In Section 1.3 we will consider this selection mechanism in some more detail.
- The Equation manager (EQM) similarly maintains a database of all conditional rewrite rules in the specification. Here the selection mechanism is also used to avoid copying rewrite rules. The basic implementation of term rewriting systems is straightforward. There are some particularities though.
  - ▷ As mentioned earlier ASF+SDF supports list-sorts. The use of list sorts makes backtracking necessary because list variables can match with lists of any length. For instance, in order to match $XY$ (a list expression containing two list variables which indicates a list separated into two sub-lists) with the list $ab$ (containing two elements) 3 alternatives are legal and have to be considered:
    - · $X =$ (empty), $Y = ab$;
    - · $X = a, Y = b$;
    - · $X = ab, Y =$ (empty).

    Ordinarily, this backtracking only occurs during matching. When conditions are considered this is no longer true. Suppose one of the alternatives constitutes a match, then the conditions have to be checked. This may imply normalization of many terms and the application of many rules (including the one we were considering). If any of the conditions fails we still have to consider the remaining alternatives of the original match. List sorts will be discussed in Section 1.4.1.
  - ▷ In a term rewriting implementation where the set of rewrite rules is potentially very large, even when this set frequently undergoes

small changes, efficiency can be gained by applying text scanning techniques to term matching. In the EQM most rewrite rules are put in a special database from which a term scanner is generated and incrementally maintained. The tool used for this purpose is the same lexical scanner generator used by the SM to create its scanners. In Section 1.4.2 this will be discussed in some detail.

▷ In a term rewriting environment it may sometimes be useful to use low level implementations of basic datatypes such as integers or tables. A large amount of execution time is spent in rewritings on these low level datatypes, whereas the underlying programming language can perform the required calculations almost instantaneously. In our system a method is used which allows one to exploit the underlying programming languages directly. This method is completely transparent; such so-called 'hybrid' datatypes are indistinguishable from ordinary datatypes except in the defining module. Hybrid specifications are discussed in Section 1.4.3.

## 1.2 Implementing conditional rewriting

Formally, the conditional rewrite relation [32, 4] is the smallest relation satisfying:

$$
\begin{aligned}
s \to t \overset{\text{def}}{\Longleftrightarrow}\; & \\
& \exists C[\,] \\
& \exists\, \langle s_0 \to t_0 \ \mathbf{when}\ s_1 = t_1, \ldots, s_n = t_n \rangle \in R \\
& \exists \sigma \\
& :\ \ s = C[s_0^\sigma] \wedge \\
& \quad\ \bigwedge_{i=1}^{n} s_i^\sigma \downarrow t_i^\sigma \wedge \\
& \quad\ t = C[t_0^\sigma]
\end{aligned}
$$

where

- $C[\,]$ is some context (a term with a hole);
- $\langle s_0 \to t_0 \ \mathbf{when}\ s_1 = t_1, \ldots, s_n = t_n \rangle$ is a conditional rewrite rule (equation) in the specification. In accordance with the common notation for term rewriting systems, in this chapter we will use the symbol $\to$ in rules. In ASF+SDF the symbol $=$ is used in order to reflect the fact that, strictly speaking, equational specifications are being considered. The term rewriting implementation is only one particular model. We have also used the **when**-notation, instead of the more common notation
$$\frac{s_1 = t_1, \ldots, s_n = t_n}{s_0 \to t_0} \in R.$$

In ASF+SDF either notation can be used, but the **when**-notation looks better, embedded in text;

- $\sigma$ is a substitution assigning arbitrary terms to all variables in $s_0, \ldots, s_n$ and $t_0, \ldots, t_n$;

- the symbol $\downarrow$ means: have a common normal form. Actually several meanings could be taken for this symbol (see [4]):

  - ▷ $s$ is equal to $t$, in equational logic, where rewrite rules are regarded as equations;
  - ▷ $s$ and $t$ share a common reduct, i.e., there is a $u$ such that $s \twoheadrightarrow u$ and $t \twoheadrightarrow u$, where $\twoheadrightarrow$ is as defined below;
  - ▷ $s$ can be rewritten to $t$ ($s \twoheadrightarrow t$).

  A term rewriting implementation usually poses constraints on the term rewriting system, in order to guarantee soundness and completeness. Under these constraints all meanings listed above are identical. Our definition of $\downarrow$ is given below.

The terms $s$ and $t$ may contain variables. These variables are not instantiated however, and they are considered to be distinct from all variables occurring in rewrite rules. That is, for all practical purposes these variables are indistinguishable from constants which occur nowhere else.

Also of importance is the transitive, reflexive closure $\twoheadrightarrow$ of the relation $\rightarrow$, recursively defined by

$$
\begin{aligned}
&s \twoheadrightarrow t \stackrel{\text{def}}{\Longleftrightarrow} \\
&\qquad s = t \vee \\
&\qquad \exists u : s \rightarrow u \wedge u \twoheadrightarrow t
\end{aligned}
$$

Using this relation we can now define $\downarrow$ as the smallest relation satisfying:

$$
\begin{aligned}
&s \downarrow t \stackrel{\text{def}}{\Longleftrightarrow} \\
&\qquad \exists u : s \twoheadrightarrow u \wedge t \twoheadrightarrow u
\end{aligned}
$$

In fact, more than one definition could be used here (see [32]). Our definition is usually referred to as a **Join system**. For our purposes the particular choice will prove to be irrelevant.

Observe that the 'smallest relation' constraints are essential. For instance, the rewrite rule

$$a \rightarrow b \textbf{ when } a = b,$$

where $a$ and $b$ are constants, is consistent both with the empty relation and with the relation containing $a \rightarrow b$. By definition the former is preferred.

In a broad sense an implementation is any decision procedure which, given two terms $s$ and $t$ determines if a sequence of terms $u_1, \ldots, u_n$ can be found, such that $s \downarrow u_1$, $u_1 \downarrow u_2$, ..., $u_{n-1} \downarrow u_n$ and $u_n \downarrow t$. That is, it determines if $s$ and $t$ are equal in equational logic.

Unfortunately this problem is often not decidable, in which case no such decision procedure can exist. If some restrictions are imposed on the term rewriting system however, a decision procedure does exist. We will proceed to formulate such restrictions.

A term rewriting system is **confluent** if for any $s$, $t_1$ and $t_2$, for which $s \twoheadrightarrow t_1$ and $s \twoheadrightarrow t_2$ a term $u$ exists such that $t_1 \twoheadrightarrow u$ and $t_2 \twoheadrightarrow u$. In words: all diverging reductions can converge again.

A term rewriting system is called (strongly) **normalizing** if no infinite reduction sequences exist.

In the unconditional case this definition is sufficient, but in the conditional case we have to be more careful, as the following two examples show.

1.
$$f(x) \to x \textbf{ when } f(f(x)) = x$$

2.
$$\begin{cases} f(x) \to g(x) \\ g(x) \to x \textbf{ when } f(x) = x \end{cases}$$

In the first example infinitely many conditions must be considered without a single rewrite step ever being taken. In the second example this happens again, but now a reduction step is taken each time.

It is not sufficient to merely exclude infinite reduction sequences. Now, we also have to assume that no infinite recursion exists on the conditions and that evaluation of a condition does end at some point.

For this reason we use the notion of a **decreasing** term rewriting system [32, 11].

A **decreasing ordering** $\prec$, is a well-founded ordering on terms (there is no ever-decreasing chain of terms), such that if $s$ is a subterm of $t$ we have $s \prec t$, and for every rule $s_0 \to t_0$ **when** $s_1 = t_1, \ldots, s_n = t_n$, and every substitution $\sigma$ we have
$$t_0^\sigma \prec s_0^\sigma \bigwedge \forall i > 0 : s_i^\sigma \prec s_0^\sigma \wedge t_i^\sigma \prec s_0^\sigma.$$

A term rewriting system is decreasing if there exists a decreasing ordering.

It is easy to see that a decreasing term rewriting system is normalizing and that such a system can not display the behavior described in the examples above. Furthermore, in a decreasing term rewriting system the notions $\to, \twoheadrightarrow$, $\downarrow$ and normal form are decidable ([11]).

If a term rewriting system is confluent and decreasing, a decision procedure as described earlier does exist. To see this, first observe that under these conditions every term has a normal form, and that this normal form is unique. This is easily verified.

Now, given two terms $s$ and $t$, a sequence $u_1, \ldots, u_n$ exists, such that $s \downarrow u_1$, $u_1 \downarrow u_2$, ..., $u_{n-1} \downarrow u_n$ and $u_n \downarrow t$ if and only if $s$ and $t$ have the same normal form. If they do, that normal form can be taken as $u$ (where $n = 1$). If such a sequence exists, $s$ and $t$ can be shown to have the same normal form by repeatedly showing that $u_{i-1} \downarrow u_i$ and $u_i \downarrow u_{i+1}$ implies that $u_{i-1} \downarrow u_{i+1}$ (using confluence).

The normal forms of $s$ and $t$ can be found by repeatedly applying rewrite rules to the terms. Confluence guarantees that when more than one subterm exists which can be reduced, any may be taken. The fact that the term rewriting system is decreasing guarantees that the process will eventually terminate.

In the context of confluent, decreasing term rewriting systems the unique normal form of a term is of great importance. It uniquely indicates all terms to which that term is equal. Accordingly, instead of a decision procedure which checks equality of terms, we will regard as an implementation the function which determines the normal form of a term.

Before we continue to describe our implementation one final word about confluent and decreasing systems. Both properties cannot be enforced, because they cannot be checked. In general it is not decidable whether a term rewriting system is confluent and/or decreasing. This means that termination of the implementation and correctness of its answers is only guaranteed if the term rewriting system under consideration is confluent and decreasing.

### 1.2.1 The implementation

Our implementation of conditional term rewriting is based on the functions `normalize`, `reduce`, `match`, `validate` and `inst`.

`normalize` puts an arbitrary term into normal form by repeatedly applying rewrite rules to the term and all its subterms;

`reduce` does the same, though under the assumption that the immediate subterms (if any) of the term are already in normal form. Hence, it only needs to check for possible reductions at the toplevel of a term;

`match` matches a subject term against an open term. The open term is the left-hand side of the unconditional part of a rewrite rule, or either side of a condition. The result of `match` is a boolean value indicating whether a match was established. The function uses, and possibly extends a substitution, which is maintained in a global variable. Initially this substitution is empty, but when a new variable is encountered it is stored in this substitution, together with the corresponding subterm. If a variable occurs more than once, all occurrences must match identical subterms of the subject term. Variables in the subject term are handled as constants.

Before explaining `validate` we should look at the problem of supplementary variables. A **supplementary variable** is a variable occurring in the right-hand side of an equation, or in the conditions, but not in the left-hand side of the rule. Consider for example the variable $z$ in the equation $f(x,y) \rightarrow z$ **when** $g(x) = h(z)$. Suppose, for the moment, that the term $h(z)$ is a normal form. Then, this equation means that the term $f(t_1, t_2)$ may be rewritten if the normal form of $g(t_1)$ matches with $h(z)$. If so, the result of the rewriting is whatever the variable $z$ was matched against.

In general, different degrees of freedom could be allowed for supplementary variables. One extreme is to disallow them all together. This is not as extreme as it may seem. In fact most methods described here are identical in expressive power. For instance, the equation above could be expressed without supplementary variables by introducing two auxiliary functions $f_1$ and $f_2$ with the following equations:

$$f(x, y) \rightarrow f_1(g(x)) \textbf{ when } f_2(g(x)) = true$$
$$f_2(h(x)) \rightarrow true$$
$$f_1(h(x)) \rightarrow x$$

A second approach is to allow introduction of supplementary variables, but only as 'local' names of common subexpressions. That is, a condition either contains no supplementary variables; or one side is a supplementary variable only, and the other side is the (sub)expression being 'assigned' to that variable. These variables could be disposed of, by replacing them with the associated expression.

The third approach is to allow either side of a condition (but not both) to be an arbitrary term containing supplementary variables. The side not containing supplementary variables is instantiated and normalized. Then it is matched against the side that does contain supplementary variables, and all supplementary variables are instantiated. This only makes sense if the latter side is itself in normal form. Otherwise, matching would fail, whereas the condition might hold according to the definition of $\downarrow$. For this reason, the side of a condition which contains supplementary variables is assumed to be in normal form. This approach was used in the example above and it is the method implemented in the current EQM.

Even more liberal would be to allow either side to contain supplementary variables. Both sides are instantiated and normalized. The free (supplementary) variables are handled as constants during rewriting. Then the two sides are matched in a mirror-matching fashion. Again a choice exists here. Either simple matching is used, where every occurrence of one variable matches the same term in all places, or full unification is used.

We can now present validate.

`validate` gets a list of conditions. It uses and possibly extends the substitution so-far, which is maintained in a global variable. For each condition it checks if either side (but at most one) contains supplementary variables. If so, it is kept as it is. Otherwise it is instantiated (using the current substitution) and normalized. Then the two sides are compared, or the side containing supplementary variables is matched starting with the current substitution. This either fails, in which case the rule is not applicable, or the (possibly extended) substitution is returned. This is then used to evaluate any further conditions.

So far we have not looked at reduction strategies. The reduction strategy determines which subterm of the term to be reduced is done first, if there are more then one. Suppose for example that the TRS contains the rule $or(false, x) \rightarrow x$ and suppose we want to normalize the term $or(false, not(or(false, y)))$. Two possible reduction sequences exist:

$$or(false, not(or(false, y))) \rightarrow not(or(false, y)) \rightarrow not(y)$$
$$or(false, not(or(false, y))) \rightarrow or(false, not(y)) \rightarrow not(y)$$

That the two normal forms are the same is not surprising. As a consequence of the confluence of the TRS, the normal form of any term is uniquely defined. Reduction strategies may differ, however, in number of reduction steps, memory usage or ease of implementation. There is no reduction strategy which in general uses a minimal number of reduction steps, nor, given a particular TRS, can such a strategy easily be found (if one exists at all) [21]. Consequently, we will concentrate on the implementational issues.

Basically there are two reduction strategies: innermost and outermost. Innermost strategies put all subterms of a term in normal form before applying rules to its top level. Outermost strategies apply rules to the top level until no rule is applicable. Then the immediate subterms are considered, and so on. Two circumstances are in favor of innermost reduction strategies:

- When a term is being considered, all its subterms are already in normal form. Hence when the term is matched against a left-hand side, all variables are associated with values which are in normal form. After a reduction this information can be used to cut out a substantial amount of useless matching.

  No comparable situation exists in the outermost case. After a reduction a reducible subterm many occur anywhere below or above the reduced part, up to the root, or at least as may levels higher as the deepest left-hand side. For an example of this, consider the following TRS.

  (1) $s(s(s(x))) \rightarrow x$
  (2) $plus(s(x), y) \rightarrow s(plus(x, y))$

  Now consider the term $s(s(plus(s(0), 0)))$. First rule (2) is applicable, but then rule (1) becomes applicable at the top level. (Although we will not investigate this here, one can define bottom up matching [22] which probably renders similar properties to outermost strategies).

- When a rule is applied to a term, the right-hand side of the rule is instantiated. This involves the creation of a number of cells in storage representing the new term. These cells are located in some form of heap, upon which garbage collection has to be applied occasionally, at run-time. When one is using an innermost reduction strategy, the creation of these cells can be largely avoided. Cells are only created for normal forms, otherwise they are immediately rewritten again. This is possible because a lot of information is quite naturally kept on the system stack, and no garbage collection is needed for this stack.

It would be unfair not to list some advantages of outermost strategies as well.

- For regular term rewriting systems the parallel outermost reduction strategy (which continuously reduces all outermost redexes in parallel) always terminates, even if the TRS is only weakly normalizing (i.e., if every term has a normal form, but infinite reduction sequences may exist) [35]. See also Chapter 2.

- Outermost strategies behave superior for some often occurring rewrite rules. For example, the if-function could be defined by

  (1) $if(true,\ x,\ y)\ \rightarrow\ x$
  (2) $if(false,\ x,\ y)\ \rightarrow\ y$

  Given some term $if(t_1, t_2, t_3)$ an innermost strategy would compute the boolean value (the normal form of $t_1$) and both the true-case ($t_2$) and the false-case ($t_3$) and only then would it select the required case. The leftmost-outermost strategy normalizes the boolean and then immediately continues with the proper case, not considering the other case at all.

- Tracing information generated during term rewriting seems to make more sense for outermost strategies than for innermost strategies. In an environment such as the ASF+SDF system this is of importance.

The EQM implements the leftmost innermost reduction strategy, although it currently does not exploit the second advantage mentioned for innermost strategies.

Finally, we can present the instantiation function `inst`.

`inst` gets the right-hand side of a rule and uses the current substitution, and computes the normal form of the required instantiation by scanning the open term bottom up and applying `reduce` at each level. An intuitive approach to `inst` would be to instantiate the open term and to apply `normalize` to the result. That approach is not efficient, however, because it fails to use the advantages mentioned earlier. By applying reduce at each level we do use the fact that instantiated variables are in normal form.

After a few final comment, we can now present our pseudo code implementation of conditional term rewriting.

- The variable `trs` represents the global set of conditional rewrite rules, and the variable `sub` is the substitution being constructed;

- The following functions and predicates operate on the syntactic representation of terms, substitutions and conditional rewrite rules: `is-variable`, `lhs`, `rhs`, `conds`, `first`, `rest`, `lookup`, `occurs`, `term`, `ofs`, `args`, `is-empty` and `has-supp-vars`. They all have an obvious meaning, and will not be discussed in detail;
- The higher order functions `map` and `each` get a function name and a number of value-lists as arguments. Then they apply the indicated function to the first values in the lists, and to the second, and so fourth. In the case op `map`, this continues until the lists are exhausted. In the case of `each`, the indicated function should be a predicate, and it is evaluated as long as `true` is returned, until the lists are exhausted. `Each` the returns a boolean value indicating whether all evaluations succeeded.

```
global trs, sub
inst(pat)
  case
    is-variable(pat):
      return lookup(pat, sub)
    otherwise:
      return reduce(term(ofs(pat),map('inst',args(pat))))
validate(conds)
  while not is-empty(conds)
    cond := first(conds); conds := rest(conds)
    case
      has-supp-vars(lhs(cond), sub):
        if not match(lhs(cond), inst(rhs(cond)))
          then return false
      has-supp-vars(rhs(cond), sub):
        if not match(rhs(cond), inst(lhs(cond)))
          then return false
      otherwise:
        if inst(lhs(cond)) ≠ inst(rhs(cond))
          then return false
  return true
match(pat, term)
  case
    is-variable(pat):
      if occurs(pat, sub)
        then return lookup(pat,sub) = term
        else sub := sub + ⟨pat ↦ term⟩
             return true
    otherwise:
      return ofs(sub) = ofs(term) and
             each('match',args(pat),args(term))
```

```
reduce(term)
  for rule ∈ trs
    sub := ∅
    if match(lhs(rule), term)
      then if validate(conds(rule), sub)
             then return inst(rhs(rule))
  return term
normalize(term)
  if is-empty(args(term))
    then return reduce(term)
    else return reduce(term(ofs(term),
                        map('normalize',args(term))))
```

## 1.3 Lazy, incremental maintenance

Recall that the Equation Manager is being used in an interactive environment in which the equations in specifications can be modified interactively.

In the EQM incremental maintenance is straightforward. The granularity of changes is at the level of entire equations. That is, changes to a subpart of an equation (from the head or the conditions down to, say, a variable name) are effectuated by removing the old equation and inserting the new one. Changes affecting a number of equations are handled as a sequence of changes to individual equations. This means that the interface of the EQM with the rest of the system, as far as maintenance is concerned, consists of two functions: `add-equation` and `delete-equation`. Both functions have the same arguments:

- the EQM (technically more than one could exist);
- the equation (which has a tag, a conclusion and zero or more conditions);
- the module in which the equation is defined.

Propagating changes to the level of one or more equations, is not a part of the EQM. In fact it might be considered as a separate component: the Change Manager. Whenever a change is reported by one of the editors, the Change Manager analyses the change and breaks it down in one or more edit operations for the EQM, the SM and possibly some alerts to other editors. Clearly the Module Manager is needed in order to determine if the change has implications for other modules. This Change Manager is currently a part of the Module Manager.

As mentioned earlier, when a module imports another module, no copy is made of that other module, nor of any information pertaining to it. For the EQM this means that only a single copy exists of every equation. Both in the context of the original module and in that of any importing module the same rule is used.

The next question is: how does the EQM know which rules are applicable and which may not be used when a particular term is being normalized. This depends on the current module and the import graph (the graph representing import relations between modules). As mentioned, this information is not available to the EQM. The basic notion used throughout the system is the **selection**.

A selection is a restricted view on a manager, in which only active rules of that manager can be used. We can create more than one selection for one particular manager. In each selection individual rules of the manager can be selected or deselected. For example, we can create a new selection for the SM. Initially, no grammar rules are selected and consequently no string can be scanned or parsed using that selection. Next, some rules occurring in the SM may be selected. Now, if a string is scanned and parsed using this selection, only the selected rules will be used. It is, as if only those rules have been entered in the SM. It is important to note that only a single scanner and a single parser exists in the SM. Each selection, when used, modifies this scanner and parser to reflect its particular subset of rules.

In the EQM, the selection mechanism is very straightforward. Selections simply maintain a sublist of the entire list of equations. This sublist is then used during rewriting. An EQM selection is created by the EQM function `select`, which gets a list of module names. Initially only rules in the indicated modules are used.

The final interface function is of course the function `normalize`, which gets a term and a selection and which returns the normal form of that term using that selection. This function is roughly the function `normalize` from the implementation described in the previous section. The variable `trs` used there corresponds to the list of active rules in the current selection.

## 1.4 Implementation Issues

In this section we will investigate in some detail, three features of the EQM which are considered to be less common. These features are

- list sorts;
- tree matching automata;
- hybrid specifications.

In ASF + SDF so-called list sorts exist, the values of which are lists of values of some other sort. In equations over list sorts an open term may match with a closed term in more than one way. The consequences of this problem are discussed in Section 1.4.1. Section 1.4.1 is co-authored by C.H.S. Dik.

Tree matching automata are an implementation technique which applies (lexical) string-matching algorithms based on automata to term matching. This technique is discussed in Section 1.4.2.

A hybrid specification is a specification in which low level datatypes or i/o related functions are imported into the algebraic framework. This is discussed in Section 1.4.3.

### 1.4.1 List Sorts[1]

In ASF+SDF so called **list sorts** are defined. A list sort is a sort the values of which are lists of a parameter sort. List sorts do not have to be declared as separate sorts. Instead, in any position where a sort name may occur, one may use `SORT*` or `SORT+`, where `SORT` is the sort (parameter) of the list elements, and where the `*` and `+` symbols indicate that the lists may be empty, or should contain at least one element, respectively. The two variants are compatible, the only constraint being that anything (e.g., variable or argument sort) declared with a `+`, should at run-time only have values with at least one element.

Syntactically, lists often have a separator, which does not occur if the list is empty, or contains precisely one element, but which does occur between elements if there are more than one. In ASF+SDF a list separator can for instance be defined as follows.

```
{ STATEMENT ";" } * -> STATLIST
```

This defines value of the sort `STATLIST` to be lists containing zero or more values of the sort `STATEMENT`, syntactically separated by semicolons, if there are more than one.

Note that the separators concern syntax only. They do not appear in the abstract syntax.

A special list sort is `CHAR*` (`CHAR+`). Apart from its normal use, as sketched above, terms of this sort can be matched against any lexical construct. For this purpose an injection function is automatically generated for every sort `SORT` with the following SDF rule.

```
"sort" "(" CHAR* ")" -> SORT
```

The name is the lower case version of the sortname.

When such a function is used, the lexical construct is viewed as a list of characters. This feature allows one to manipulate even single characters from a lexical token. Apart from the conversion from lexical token to list of characters, which is done by the EQM whenever it recognizes such a special function, the treatment of such constructs is as any other list. Consequently we will not discuss this feature in any more detail.

Variables of list sorts introduce a particular problem in the matching process. In the ordinary case, discussed in Section 1.2.1, a straightforward procedure is used, which recursively visits each node in the tree representation of a term, according to the structure of the term. In that situation, a term can match with a rule in at most one way and the instantiating substitution is unique.

[1] This section is written in cooperation with C.H.S. Dik

With list sorts there can be more than one binding. Consider for example equation [1] from module `Sets` in Section 1.1, where variable `i` is of the parameter sort, and where variables `x1`, `x2` and `x3` are of the list sort.

```
[1] {x1,i,x2,i,x3} = {i,x1,x2,x3}
```

Now look at the following term.

```
{Tinker, Tailor, Soldier, Soldier, Soldier}
```

There are two distinct substitutions under which this term is an instance of equation [1].

```
x1 = {Tinker, Tailor}, i = Soldier, x2 = {}, x3 = {Soldier}
x1 = {Tinker, Tailor}, i = Soldier, x2 = {Soldier}, x3 = {}
```

Since there are two valid, satisfactory substitutions, there cannot be a matching procedure which yields the unique instantiation, and the entire construction of our term rewriting implementation must be redesigned. Note that we can not simply choose one of the substitutions, because it may later on be rejected in one of the conditions. Suppose for instance that the condition '`x2!={}`' is added to equation [1]. Then the first alternative is no longer valid.

Basically the problem is caused by the fact that a list variable can match lists of arbitrary length, and the first time the variable is encountered during matching, a range of possibilities could exist. If we were using a parallel language, this observation could be exploited, and a recursive algorithm could still be provided: when a list variable is encountered for the first time – when it is not yet present in the substitution, – a number of parallel processes is started. One where the variable matches the empty list, one where it matches the list containing only the first item of the list occurring in the subject term, and so forth, up to the last alternative, where the variable matches the entire remaining list in the subject term. The set of valid substitutions would then be the union of all sets returned by those processes.

Note that this parallel approach could be taken even further. When a substitution is valid, the conditions still have to be checked. In principle this could also be done in parallel, but in that case parallel rewriting has to be considered. We will not go into this. It is important however, to be aware of the parallel nature of list matching, and of the fact that choosing an acceptable substitution requires checking the conditions, which could involve many rewritings, possible using the very rule being considered.

In our sequential implementation we will implement parallel matching with backtracking.

We will now summarize two key observations mentioned above.

- Traversal of the (inductively defined) pattern and subject term should not be done recursively.
- Flow of control in the list matching algorithm should offer a construction which can support backtracking, in order to handle all possibilities for every list variable.

It is not difficult to transform the recursive matching algorithm to an iterative algorithm. Below are two skeletons for a tree traversal algorithms.

```
recursive(tree)
  visit(tree)
  foreach arg ∈ args(tree)
    recursive(arg)

iterative(tree)
  repeat
    visit(tree)
    tree := advance(tree)
  until done
```

The function `advance` used above implements a pre-order tree traversal by advancing its argument until the root is again reached, in which case the global variable done is set. This function can not be shown directly, because its definition depends on the functionality of the datastructures used to represent terms. For the moment we will not go into details, and we will regard the function as a primitive.

Since recursion is no longer used for tree traversal it becomes a suitable vehicle for the implementation of backtracking. Note however, that any other means could also have been used.

An implementation of backtracking should have two properties:

- When an attempt is made, which is one out of a number of possibilities, flow of control should return eventually, indicating whether the attempt and all its consequences were successful. If so, control is passed on, and success is indicated. Otherwise another alternative is tried, or, if all alternatives have been exhausted, control is passed on and failure is indicated.

- When an attempt has been been made, but failed, the entire state (that is, the values of all variables) should be restored as it was immediately before the attempt (except of course the variable indicating which attempt comes next).

From these two properties we can see that backtracking can be implemented with a function call which, as its result, indicates whether an attempt and all its consequences have been successful, and which, by its nature, does not alter the state in which it is evaluated.

As mentioned, an attempt has to check all consequences before success is apparent. Consequently, the functions `match` and `validate`, defined in our implementation in Section 1.2.1, will be integrated into a single function `m&v`. This function has four arguments: the pattern (open term), the subject term, the list of conditions and the current substitution. The substitution is added as an argument only in order to have it restored each time an attempt fails, as are the other three arguments.

So far, we have not yet encountered list-sort values in our code. We will now present the notation used, and we will look in some more detail at the function `advance` mentioned earlier.

- In our pseudo language, list values are 'stored' as ordinary terms. The outermost function symbol indicates the sort, and the fact that it is a list operator. The arguments are the elements of the list, and there may be arbitrarily many (we will not go into the difference between '`*`'- and '`+`'-lists). Since all terms are well-typed, the list operators will always match and need not concern us.

- If `t` is an element in a list, then the notation `t[...n]` indicates the sublist of that list containing precisely `n` elements and starting with `t`. For instance, if `t` is the item '5' in the list '1 2 3 4 5 6 7 8 9', then `t[...3]` is the list '5 6 7'. Note that `t[...0]` indicates the empty list, for any `t`.

- The function `length`, when given an element of a list, yields the length of the sublist starting with that element up to the last element in the list. In the example above, `length(t)` yields 5.

- There are two versions of `advance`. The one-argument version gets a location in a term as its single argument. It returns the next location in the term, according to a pre-order traversal. If the root is encountered, the global variable `done` is set. For instance, suppose `t` is the root location of the term `f(a,b)`. Then `advance(t)` is the subterm `a`, `advance(advance(t))` is `b` and `advance(advance(advance(t)))` sets the variable `done`.

  Note that in order to implement `advance`, some features should be available. In particular, given a term, its parent (if present) and its right siblings should be accessible. We will not go into these details, but we will mention that if this functionality is not supported by the datastructures used, it could still be achieved by extending the concept of 'a location' in a term. This could for instance be implemented as a path starting from the root.

  Given the basic tree movement functions mentioned above, `advance` could be implemented as follows.

```
advance(tree)
  if not is-empty(args(tree))
    then return first(args(tree))
    else while 'tree has no right sibling'
           if 'tree has no parent'
             then done := true
             else tree := parent(tree)
         return right-sibling(tree)
```

- The two-argument version of `advance` gets a location in a term, and a count, and advances the first argument as many times as indicated by the second argument.

We can now present the pseudo implementation of conditional term rewriting with list sorts. We will only show the functions `m&v` and `reduce`. All other function remain as shown earlier.

```
m&v(pat,term,conds,sub)
  local tmp cond count
  repeat
    case
      is-normal-var(pat):
        if occurs(pat, sub)
          then if lookup(pat,sub) ≠ term
                 then return false
          else sub := sub + ⟨pat ↦ term⟩
        pat := advance(pat)
        term := advance(term)
      is-list-var(pat):
        if occurs(pat, sub)
          then tmp := lookup(pat,sub)
               if length(term) < length(tmp) or
                  tmp ≠ term[...length(tmp)]
                 then return false
                 else return m&v(advance(pat),
                                 advance(term,length(tmp)),
                                 conds, sub)
          else for count := 0 to length(term)
                 if m&v(advance(pat),
                        advance(term, count),
                        conds,
                        sub + ⟨pat↦ term[...count]⟩)
                   then true
               return false
      otherwise:
        if ofs(pat) ≠ ofs(term)
          then return false
        pat := advance(pat)
        term := advance(term)
  until done
  while not is-empty(conds)
    cond := first(conds)
    case
      has-supp-vars(lhs(cond), sub):
        return m&v(lhs(cond), inst(rhs(cond)),
```

```
                  rest(conds), sub)
        has-supp-vars(rhs(cond), sub):
          return m&v(rhs(cond), inst(lhs(cond)),
                  rest(conds), sub)
        otherwise:
          if inst(lhs(cond)) ≠ inst(rhs(cond))
            then return false
            else conds := rest(conds)
    return true

  reduce(term)
    for rule ∈ trs
      if m&v(lhs(rule), term, conds(rule), ∅)
        then return inst(rhs(rule))
    return term
```

Tests have shown that an algorithm as shown above has an efficiency comparable to that of the recursive tree matching algorithm without list matching.

One final word about list sorts is in order. Although the implementation of list matching shown above is reasonably efficient, one should be aware that any implementation always has to check the entire search space, until a solution is found. Currently there is no way to direct this search by skipping unpromising areas. Compare this situation with Prolog, where the '!' (cut) operator serves precisely this purpose. Without this operator Prolog probably would no longer be used. However, list sorts have only recently been implemented in the EQM, so experience must still be gathered before optimalizations can be designed.

To end this section on an optimistic note we could observe the following. The equation defining the unicity of elements in a set, as shown in this and in earlier sections, is algorithmically optimal. That is, no more comparisons are made than when using a normal specification. Furthermore, fewer reductions are needed than in the normal case, namely one per pair of identical elements.

### 1.4.2 Tree matching automata

The most time consuming operation in a term rewriting implementation is, given a term and a set of rules, to find those left-hand sides of rules (if any) of which the term is an instance. For every rewrite step this operation has to be performed at least once, and even more if it fails in one part of the term and has to be applied elsewhere.

This operation resembles string scanning where, given a subject string and a set of regular expressions, the regular expression is sought of which the subject (or a prefix thereof) is an instance, if such an expression exists at all.

This similarity can be exploited: tree matching can be viewed as a special kind of string scanning. Indeed, several efficiency increasing techniques developed for string scanning have already been applied to term matching ([22] [36] [37]). These techniques are less suitable for the EQM, since they are usually based on the assumption that a given set of rules will be applied to many terms. Under this assumption efficiency can be gained by preprocessing the set of rules. Unfortunately this preprocessing is difficult to implement when making incremental changes to the set of rules.

In the ASF+SDF system, the similarities between tree matching and text scanning are exploited to the fullest. The Syntax Manager contains a generator for lexical scanners (Modular Scanner Generator [19, 29]) which has been extended such that now it can also create term matching automata. The reuse of this scanner generator has been advantageous. It was already lazy, modular (using the selection mechanism) and incremental. The loss in speed, in the text case, both of the generation phase and of the generated scanners is marginal. The gain in overall execution speed of the term rewriting implementation has been measured at a factor five, in representative cases.

We will now present a brief description of the technical issues. A more detailed account can be found in Chapter 3.

The key idea when applying string scanning techniques to term matching is to flatten the terms. Instead of considering a term, we regard, loosely speaking, the string of function symbols without parentheses and commas. For instance, the term $p(z, s(z))$ appears as $pzsz$. This flattening coincides with a pre-order traversal of the tree representing the term.

If we disregard variables (for the moment), this process can be applied to the entire set of open terms (that is, the set of left-hand sides of all rewrite rules). The result is a set of string constants. Using standard techniques, an automaton can be created which accepts precisely all constants in that set. Any subject term can then be flattened and supplied to the automaton which will accept it if and only if it occurs in the set. For example: consider the pattern set $\{z, s(z), p(z, z), p(s(z), z), p(z, s(z))\}$. This set is recognized by the automaton in Figure 1.2.

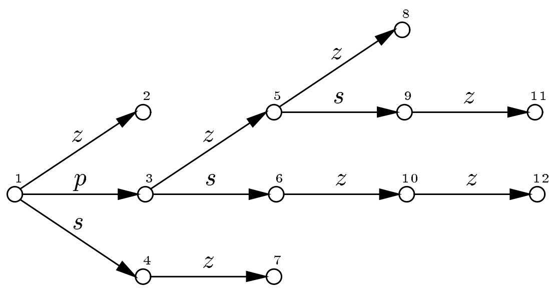
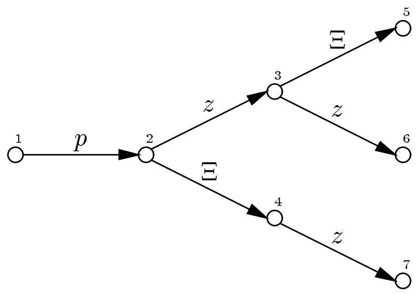



*Figure 1.2.*

So far, we have ignored variables, because they constitute a major problem. Consider the pattern set $\{p(z, \alpha), p(\beta, z)\}$, where $\alpha$, $\beta$, ... are variables. An automaton accepting instances of this set is sketched in Figure 1.3.




The intended meaning of edges marked $\Xi$, is that transitions along such edges can only be taken if no other transition is possible. Consequently, the automaton is deterministic.

*Figure 1.3.*

The edges marked $\Xi$, which accept the string corresponding to an arbitrary term, cannot be defined by any finite automaton. Moreover, even if such an automaton existed its use would be questionable, as is shown in the following example. Consider the term $p(s(s(s(s(z)))), z)$. To recognize it as an instance of $p(\beta, z)$ we do not have to look at a single symbol of the subterm $s(s(s(s(z))))$. We only have to check the outermost function symbol, and that of the second immediate subterm. Looking at the string *pssssz z* however, it is not immediately clear where the second immediate subterm starts. This is because too much has been flattened. It turns out that the subterm $s(s(s(s(z))))$ was not needed at all.

We would like an automaton to be able to skip parts of the tree being scanned. The term is not flattened beforehand, but the traversal is done while the automaton is scanning and the traversal process can be directed by the automaton.

In our automata all edges will be labeled with the tree move to be performed when the transition corresponding to that edge is taken. Normally, this tree move coincides with pre-order traversal. When a subterm is not needed, because it is matched entirely by a variable, it is skipped.

These tree moves are 'coded' into the automaton by the generator, at a time when the tree structure is still available.

When a scan has been successful, we are not only interested in which patterns have been recognized. We are also interested in the subterms corresponding to the variables in those patterns. For instance, if the automaton reports that some term is an instance of $p(\beta, z)$, we would like to know for which value of $\beta$ this is the case. This information is available when the automaton is scanning the subject tree. Consider for example state 2 in the automaton shown in Figure 1.3. In this state the outermost function symbol corresponding to the variable $\beta$ will be read. When the automaton enters this state, while scanning some term, the 'current' subterm (pointed to by the traversal process) is precisely the value which should be associated with $\beta$.

In our automata we will annotate all states with the list of variables which match the current subterm at that point. When the scanner enters such a state, these variables, together with the associated value, will be maintained in a list. This list is eventually used to construct the substitution corresponding to a successful match. Note that since the automata are deterministic, and consequently all matching rules are found at once, this list may eventually contain information pertaining to several successful matches, and even to matches which succeeded only partially, but failed at some point. In reality this information is distributed while the automaton is scanning. That is, every variable 'belongs' to a rule, and its corresponding value is stored with the rule, but it leads too far to go into these details.

We will now present the automaton for a larger pattern set. The pattern set is the set of left-hand sides of the following rewrite rules defining addition on natural numbers.

1. $p(z, \alpha) \rightarrow \alpha$
2. $p(\beta, z) \rightarrow \beta$
3. $p(s(\gamma), \delta) \rightarrow s(p(\gamma, \delta))$
4. $p(\varepsilon, s(\zeta)) \rightarrow s(p(\varepsilon, \zeta))$

In Figure 1.4 tree moves are represented by expressions such as $u$ k $d$ n. This means: move $k$ levels up in the tree, and then move down to the $n$-th child (where $u$ k is abbreviated to $u$).

Furthermore, to every final (i.e. accepting) state the patterns which have been recognized are attached in the form of the corresponding rule number. For instance, in the state labeled '$1, 4$', an instance of the term $p(z, s(\alpha))$ has been recognized, which matches both patterns one and four.

Observe that given some term, all patterns which that term is an instance of are found in a single scan. Also observe that the time needed for a single scan does not depend directly on the size of the pattern set (it depends on the degree of overlapping in the pattern set).

In Figure 1.4 states have been annotated with the variables which occur in that state for the first time. This means that the state has the same tree location as that variable.

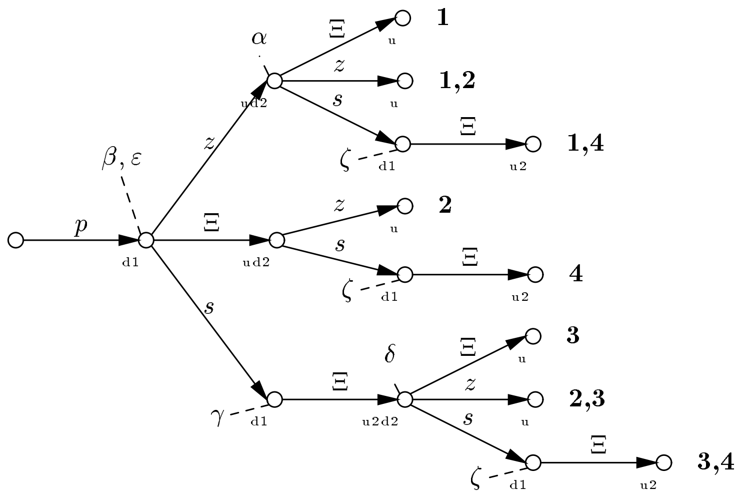



*Figure 1.4.*

As mentioned, the techniques described above have been added to the scanner generator of the SM. This extension is conservative. That is, the generator still generates all text scanners, but it is also used to create term scanners. The overhead due to this added functionality in the text scanning case is less than 10% in execution time, both of the generation process and of the generated scanners.

Using this extension, the EQM supplies the generator with a list of the left-hand sides of all equations. This list is processed as described, and a term scanning automaton is created (lazily). It is not necessary to show the rewriting process in this new context, since it is largely unchanged. Instead of applying the function `match` (Section 1.2.1) to all rules, until it succeeds, we now apply the scanner to a subject term. This yields the set of applicable rules together with the respective instantiating substitutions.

For technical reasons, which we will not investigate here (see Chapter 3) our scanners do not support non-linear patterns (i.e, where some variable occurs more than once). Instead, the scanner 'remembers' all occurrences of variables whereas the function `match` compares the associated value of all occurrences of a variable on the fly. When using a scanner, all resulting substitutions have to be validated by performing this comparison afterwards. Substitutions which do not satisfy this constraint are thrown away, together with the corresponding rules. Rules and substitutions which do apply are handled as usual. That is, the conditions are checked, and, if successful, the right-hand side is instantiated.

Since the scanner generated already supported incrementality and modularity, the list of patterns can be maintained incrementally, and parts can be disabled using the selection mechanism. A representative benchmark using 85 equations showed an increase in overall execution speed with a factor 5, compared to straightforward matching and linear search through the list of patterns. This means that the actual matching process relatively gains more than that, but no precise measurements are available.

Finally, we should mention that the method described here imposes no constraints on the pattern set (apart from the fact that no single variable may occur, but this constraint applies to all term rewriting systems.)

As it is, the method cannot be used to support list-matching. In the current EQM rules involving list-matching are handled in the standard manner. All other rules are recognized using the term scanning technique. We have not yet investigated if this method can be used in conjunction with hybrid specifications, but no essential problems seem to exist.

### 1.4.3 Hybrid Specifications

Two major disadvantages of using term rewriting for the specification of semantics are:

- Speed. Every datatype has to be specified formally, and all computations, including those concerning basic datatypes are performed with term rewriting. This means that trivial computations may require many reduction steps. For instance, the common specification of integers, with *zero* and a successor function, takes in the order of $n \cdot m$ steps to multiply $n$ and $m$.

- I/O. Strictly speaking a term rewriting implementation does not produce side effects (apart perhaps from trace information). Sometimes however, this would be very useful:

    - ▷ Consider a specification concerning graphics. Imagine how to debug such a specification using trace information only.

    - ▷ Consider a specification of the dynamic semantics of a programming language with keyboard input and textual output. Separately the I/O could be modeled with lists, but an interactive dialogue would be hard to simulate.

In Chapter 4 a theoretical framework and an implementation will be presented of a method which combines implementations of functions in a conventional programming language, such as C, Lisp, or Prolog, with a standard term rewriting implementation.

With such a system, basic, often used datatypes can be given an efficient implementation. Furthermore, some functions can have (I/O-related) side effects.

In this section we will sketch this method, we will indicate how it is implemented in the EQM, and we will introduce some notation for hybrid specifications in ASF+SDF.

**Introduction**

In order to understand this method we will first look at a simple example: the natural numbers. Most computers and programming languages have numbers as a standard datatype. All numbers (within a certain range) can be represented. The addition and multiplication operators are available and are executed very efficiently. In algebraic specifications the natural numbers are often defined using a constant $z$ (for zero), and a monadic function $s$ (mapping each number to its successor). Addition and multiplication can be defined as follows:

$$\begin{array}{lcllr} a(z,x) & \rightarrow & x & \qquad & 0+x=x \\ a(s(x),y) & \rightarrow & a(x,s(y)) & & s(x)+y=x+s(y) \\ m(z,x) & \rightarrow & z & & 0*x=0 \\ m(s(x),y) & \rightarrow & a(m(x,y),y) & & s(x)*y=x*y+y \end{array}$$

In our hybrid method we will keep the function symbols $z$, $s$, $a$ and $m$, but we will use the machine implementation for addition and multiplication. In order to avoid confusion between the language in which the term rewriting implementation is made, and the language used for hybrid implementations, we will refer to the latter as **cpl**, to indicate that any conventional programming language can, in principle, be used. Note that an obvious choice for a cpl would be the implementation language itself, but this is not mandatory.

For this purpose, we conceptually extend the signature with a new constant for every number to be implemented at cpl level. In this section such values, and the corresponding added constants are denoted by $c_t$, where $t$ is the term in the original signature which corresponds to that value. For instance, the term $s(s(z))$ corresponds to the value 2. Consequently $c_{s(s(z))}$ is, depending on context, the value 2 or the newly added constant formally representing 2.

This extension is only conceptual because no actual representation of the entire signature is maintained. That is, such a constant is created only for those numbers which actually occur.

Also, rewrite rules are added which allow terms in the $s$-$z$ notation to be transformed to their corresponding new constants.

$$z \rightarrow c_z$$
$$s(c_t) \rightarrow c_{s(t)}$$

Furthermore rules are added to map functions such as $a$ and $m$ to the corresponding implementation-level functions.

$$a(c_s,c_t) \rightarrow < c_s + c_t >$$
$$m(c_s,c_t) \rightarrow < c_s * c_t >$$

The latter two rules clearly show a problem: the right-hand side can not properly be expressed in the algebraic framework. For this reason, our method has to be formulated in a slightly more complicated fashion. In theory we add large, potentially infinite numbers of fixed rules to the term rewriting system. The implementation of our hybrid system uses rule schemes, each capable of mimicking large sets of fixed rewrite rules as suggested in the notation above. Formally the rules include:

$$\begin{aligned} s(c_z) &\rightarrow \quad c_{s(z)} \\ s(c_{s(z)}) &\rightarrow \quad c_{s(s(z))} \\ a(c_{s(s(z))}, c_{s(z)}) &\rightarrow \quad c_{s(s(s(z)))} \end{aligned}$$

The final particularity of our hybrid system appears when a term is being matched with the left-hand side of a rule (in order to determine whether the rule may be applied). If the term is one of the special constants $c_t$ it must be 'unwrapped' temporarily in order to compare its intended outermost function symbol with the corresponding symbol in the left-hand side of the rule. The term is not unwrapped entirely (i.e., to its original form in the original signature), but only one step at a time. For instance, 5 would be unwrapped to $s(4)$.

This unwrapping is necessary in order to guarantee the correctness and transparency of our system. This unwrapping is somewhat like applying rules such as $s(c_t) \rightarrow c_{s(t)}$ in reverse order. In practice, the implementor provides functions which perform this unwrapping. In ASF+SDF special unpacking 'equations' must be defined.

We will now give a step by step example of the entire rewrite process. We will show the added constants such as $c_{s(z)}$ as their cpl representation (i.e. 1).

For the purpose of illustration we will add a new function $d$ which is defined in the $s$-$z$ notation and which is 'unaware' of the existence of cpl implemented functions. The cut-off-difference $d$ is defined by:

$$\begin{aligned} d(x,z) &\rightarrow \quad x \\ d(z,x) &\rightarrow \quad z \\ d(s(x),s(y)) &\rightarrow \quad d(x,y) \end{aligned}$$

We will now normalize the term representing $((2 \times 3) - (1+1)) \times 3$.

$$m(d(m(s(s(z)), s(s(s(z)))), a(s(z), s(z))), s(s(s(z))))$$

(five rewritings, transforming $z$ into 0)

$$\begin{aligned} &m(d(m(s(s(0)), s(s(s(0)))), a(s(0), s(0))), s(s(s(0)))) \\ &m(d(m(s(1), s(s(1))), a(1,1)), s(s(1))) \\ &m(d(m(2, s(2)), a(1,1)), s(2)) \\ &m(d(m(2,3), a(1,1)), 3) \end{aligned}$$

(calling the cpl operators m and a)

$$m(d(6,2),3)$$

(unwrapping 6 and 2 to $s(5)$ and $s(1)$ in order to match $d(s(x), s(y))$)

$$\begin{aligned} &m(d(5,1),3) \\ &m(d(4,0),3) \\ &m(4,3) \\ &12 \end{aligned}$$

**Theory**

After this brief introduction to our method we can now sketch how its correctness and completeness is established. This is done by showing the equivalence of a hybrid implementation, where some functions have been implemented correctly in a programming language, with the corresponding plain implementation, where those functions have been defined with rewrite rules. This equivalence is only established for confluent, decreasing term rewriting systems. Otherwise formal equivalence may be meaningless (for instance if the hybrid functions have side effects), but the system always behaves as one would intuitively expect.

We start by selecting a set $N$ of terms which have a cpl representation. In our earlier example this set looked like $\{z,\ s(z),\ \ldots,\ s(\ldots s(z)\ldots)\}$, where the last term corresponds to the largest cpl representable integer. There is a constraint on $N$: it should be closed under subterm selection. That is, all subterms of terms in $N$ should also be in $N$. This constraint is no problem for basic datatypes, such as integers, but it seems to pose a problem for parameterized datatypes such as tables, because the subterms of terms of such datatypes include the terms of the parameter types, and for reasons of transparency and extendibility we cannot assume all possible parameters to have a cpl implementation. This problem can be solved by allowing $N$ to contain open terms. The resulting theory is essentially the same, but notationally more complicated. In this section we will only look at the simpler – non-parameterized – case. In Chapter 4 both methods are discussed in detail.

For every ground term $t$ in $N$ a new constant $c_t$ is introduced in the signature and equations will be added to the specification such that $\vdash t = c_t$. In order to avoid a proliferation of notations we will assume that $c_t$ may, depending on the context, indicate that constant or term, or the cpl value we associate with that term. For instance, given that the integer 2 is associated with the term $s(s(z))$, then $c_{s(s(z))}$ may indicate either a new constant, or the integer 2. The term $c_t$ is called an **internal representation** of $t$. The cpl value $c_t$ is called the cpl representation of $t$ (and $c_t$). Given some term $t$, **alternative representations** can be constructed by replacing some subterms of $t$ which occur in $N$, say $t_1, \ldots, t_n$, with $c_{t_1}, \ldots, c_{t_n}$ respectively. Terms entirely of the form $c_t$ are called (totally) internal representations and the term $t$ is referred to as the external representation. Let $\Sigma^+$ be the set of those constants. Let $\Sigma^* = \Sigma \cup \Sigma^+$ be the union of the old signature with these constants, and let $T_{\Sigma^*}$ be the term space over $\Sigma^*$.

In an implementation every term has a representation in the language in which that implementation was made. This representation is a datastructure in which the tree-like construction of the term is reflected. In this section we will indicate such a datastructure simply with the term it represents. That is, in the context of a programming language '$s(s(z))$' refers to the datastructure representing the term $s(s(z))$. Note that notations such as $c_t$ may now have three meanings:

- a newly added constant;
- the cpl value associated with the term $t$;
- the datastructure representing the term $c_t$. This is quite separate from the datastructure representing $t$.

We will refer to the second meaning as the **value**, and to the third meaning as the **term representation**.

For example, in an implementation the following three values may occur: the value '$s(s(z))$', which is a datastructure of type 'term'; the value 2, which is of type 'integer'; and the datastructure representing $c_{s(s(z))}$, which is of type 'term', but but from which the integer 2 can easily be derived. More on this in the section concerning the implementation.

A partial order $\preceq$ can be defined on $T_{\Sigma^*}$, which expresses that one term 'is a more internal representation of' another term. That is, $s \preceq t$ precisely if $s$ can be derived from $t$ by replacing some subterms with equivalent internal representations. For instance, in the signature of our earlier example, we have that $m(1,2) \prec m(1,s(s(0))) \prec m(s(z),s(s(z)))$, and also $m(s(z),2) \prec m(s(z),s(s(0)))$.

Every term $s$ has a $\preceq$-infimum ($\lfloor s \rfloor$), which is unique, and a $\preceq$-supremum ($\lceil s \rceil$), which is also unique and which is in $T_\Sigma$.

The importance of this partial order is that it defines the equivalence of alternative representations: two terms refer to the same conceptual value if they have the same $\preceq$-supremum.

Now, given a function symbol $f$, suppose a cpl function `f` exists such that if evaluation of `f`$(c_{t_1},\ldots,c_{t_n})$ yields $c_t$, then a reduction sequence $f(t_1,\ldots,t_n) \twoheadrightarrow t$ exists. Such a function `f` is said to **implement** $f$, and it can be used to define the new reduction $f(c_{t_1},\ldots,c_{t_n}) \rightarrow c_t$. This new reduction has the effect of a number of reduction steps in the original rewrite system. For instance, the `+` operator implements the function $a$, in our example earlier, because when `x+y` yields `z`, where `x`, `y` and `z` integers, then the normal form of $a(s(\ldots s(z)\ldots),s(\ldots s(z)\ldots))$ is $s(\ldots s(z)\ldots)$ (`x`, `y` and `z` times $s$, respectively).

Finally, we can extend the rewrite relation. In Figure 1.5 the old and extended relations are shown. Terms in this picture are ordered in accordance with $\prec$. The upper part contains all terms in $T_\Sigma$, highest in the $\prec$-ordering. Lower down are terms containing constants from $\Sigma^+$. The set $\Sigma^+$ is of course at the bottom, but terms not in $\Sigma^+$ may still be most internal representation (such as $m(c_z,c_z)$).

The plain arrows in the upper part of the picture are rewritings according to the original rewrite relation $\rightarrow$. The dotted arrows pointing down transform terms to more internal representations (e.g. $s(c_z) \rightarrow c_{s(z)}$). These arrows correspond to rules of the form $f(c_{s_1},\ldots,c_{s_n}) \rightarrow c_{f(s_1,\ldots,s_n)}$, which we will refer to as $\prec$-rules. The dashed arrows are parallel to the $\rightarrow$-arrows. They lead from more internal representations of terms to an alternative representation of whatever the corresponding $\rightarrow$-arrow is pointing to. They are a consequence of a special kind of matching used during rewriting, so-called $\prec$-matching. Informally, $\prec$-matching matches modulo the $\prec$-equivalence defined earlier, by temporarily 'unwrapping' internal representations. Clearly, $\prec$-matching should not be used when considering $\prec$-rules, because circularities would then be introduced. Usually terms need not to be unwrapped entirely, as is indicated by the fact that most dashed arrows do not end in $T_\Sigma$. Finally, the bent arrows represent the application of cpl functions. They are the result of so-called cpl-rules, which are of the form $f(c_{s_1}, \ldots, c_{s_n}) \rightarrow c_t$, where `f` implements $f$ and where $t$ can be computed by evaluating $\mathtt{f}(c_{s_1}, \ldots, c_{s_n})$.

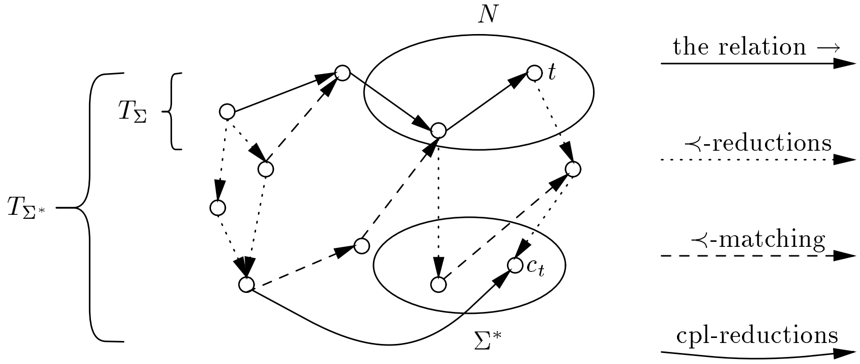



*Figure 1.5.*

With these adaptations a new rewrite relation $\searrow$ can be defined which will replace the old rewrite relation $\rightarrow$.

The correctness of our implementation is expressed in a number of theorems. The central theorem states that given an arbitrary term $s$, if the normal form of $s$ with respect to the rewrite relation $\rightarrow$ is $t$, then the normal form of $s$ with respect to the relation $\searrow$ is $\lfloor t \rfloor$. Given the fact that there is a one-to-one correspondence between a term $u \in T_\Sigma$ and its $\prec$ infimum $\lfloor u \rfloor$, this implies that the two relations are equivalent.

**Implementation**

In this section we will present an implementation scheme of algebraic specifications using our algorithmic notation. In Appendix A, Chapter 4, a detailed example of a particular TRS, together with all TRS-dependent algorithms, and a Lisp implementation are presented.

The hybrid implementation again consists of the functions `normalize`, `reduce`, `match`, `validate` and `inst`, with the same meaning as in our general implementation in Section 1.2.1. In the hybrid situation, only the functions `match` and `inst` are essentially different, although `reduce` has been adapted slightly. This will be explained. For clarity we will show all functions again.

There are some particularities and special auxiliary functions specific to the hybrid implementation, which we will now list.

- Earlier we have mentioned rule schemes: a method to implement a number of rules at once. Consider for example the rule scheme $s(c_t) \to c_{s(t)}$, where $t$ ranges over all natural numbers. The left-hand side of such a rule acts as the open term $s(x)$, with one exception: the variable $x$ can only match internal representations such as $c_z$. For this purpose so-called **limited** variables are introduced. A limited variable will only match an internal constant.

  Note that in general the left-hand side may contain both limited and normal variables.

  A recognizing function `is-limited` exists to recognize limited variables.

- The right-hand side of a rule scheme is also special. It is called a **generator**, and it is an expression which uses a substitution, defined by matching the left-hand side against some term. When the generator is evaluated, it may retrieve values from this substitution. These values can be cpl values (of any type) which were matched against limited variables, or they can be arbitrary terms matched against ordinary variables. Consider for example the rule scheme $< x > + < y > \to < x + y >$. The generator is an expression which applies the $+$ function to the integer values found for $x$ and $y$ in the the left-hand side. Note that in the notation used here the substitution does not actually appear. In the section concerning ASF+SDF, below, more detailed examples will be given.

  A generator either returns an ordinary term (for instance, of a parameter sort), or a cpl value, which is then transformed into the term representation of the corresponding constant. In the 'body' of a generator it may be necessary to normalize arbitrary terms, for instance, in order to check conditions on parameter values. For this purpose the definition of the cpl in ASF+SDF has been extended with notations which, upon evaluation, result in the proper calls to the SM and the EQM. We postpone the presentation of specific examples, until we have introduced the notation for hybrid functions in ASF+SDF.

  Note that the contents of a generator depends on the particular application. In our algorithmic implementation we will access generators with the function `evaluate`, which evaluates a generator, using the current substitution. Furthermore there is a recognizing function `is-generator`.

- As mentioned earlier, $\prec$-matching should not be applied when considering rules in $R^{\prec}$. For this reason rules have an extra field, retrieved with the function `hm` (for 'hybrid matching'). This value is passed to the function `match`, which uses it accordingly.

- The function `internal` gets a single argument (a term), and yields *true* precisely if this argument is one of the newly added constants $c_t$. Otherwise, i.e., if the outermost function symbol of the argument occurs in the original signature, it yields *false*.

- The function `val` gets a single term as argument, and yields the corresponding cpl value. For instance, if the argument is a constant such as $c_{s(s(s(0)))}$, then `val` returns the integer 3.
- The function `make` is the inverse of `val`; it gets a cpl value and returns the term representing the new constant belonging to that value.
- The function `unpack` gets a constant $c_{f(t_1,\ldots,t_n)}$. It reconstructs precisely one function symbol of the original signature. That is, it yields the term $f(c_{t_1},\ldots,c_{t_n})$. For instance, `unpack`(`make`(3)) yields a term $s(t)$, where $t$ is `make`(2).

  Note that this can be defined uniquely, because for each $t$ a unique $c_t$ has been chosen. Also note that this function depends on the TRS under consideration. For the moment we simply assume its existence. In examples at the end of this chapter we will see how such a function is actually defined.

Now we can present the pseudo code for hybrid implementations.

```
global trs, sub

inst(pat)
  case
    is-variable(pat):
      return lookup(pat, sub)
    is-generator(pat):
      return evaluate(pat, sub)
    otherwise:
      return reduce(term(ofs(pat),map('inst',args(pat))))
validate(conds)
  while not is-empty(conds)
    cond := first(conds); conds := rest(conds)
    case
      has-supp-vars(lhs(cond), sub):
        if not match(lhs(cond), inst(rhs(cond)), true)
          then return false
      has-supp-vars(rhs(cond), sub):
        if not match(rhs(cond), inst(lhs(cond)), true)
          then return false
      otherwise:
        if inst(lhs(cond)) ≠ inst(rhs(cond))
          then return false
  return true
```

```
match(pat, term, hm)
  case
    is-variable(pat):
      if occurs(pat, sub)
        then return lookup(pat,sub) = term
        else sub := sub + ⟨pat ↦ term⟩
             return true
    is-limited(pat):
      if internal(term)
        then sub := sub + ⟨pat ↦ term⟩
             return true
        else return false
    internal(term) and hm:
      match(pat, unpack(term),hm)
    otherwise:
      return ofs(sub) = ofs(term) and
             each('match',args(pat),args(term),hm)
reduce(term)
  for rule ∈ trs
    sub := ∅
    if match(lhs(rule), term, hm(rule))
      then if validate(conds(rule), sub)
             then return inst(rhs(rule))
  return term
normalize(term)
  if is-empty(args(term))
    then return reduce(term)
    else return reduce(term(ofs(term),
                              map('normalize',args(term))))
```

A remark on efficiency is in order. As can be seen, the function `match` 'unwraps' internal representations always, when they do not match a limited variable, and when a rule not in $R^{\prec}$ is being considered. In the lisp implementation in Appendix A, Chapter 4, this inefficiency is somewhat reduced by annotating tree representations of internal representations with the outermost function symbol of the unpacked version. The result is that a term is only actually unpacked when its outermost function symbol is as required.

### Hybrid specifications in ASF+SDF

In this final section we will present the notation for hybrid specifications in ASF+SDF, and we will present an example (specifying integers and tables) using this notation. We will use the language Lisp as our implementation language (cpl), although other languages could, in principle, be used with ASF+SDF.

For every such implementation language a module defining its syntax must be defined. For instance, there is a module Lisp which exports the syntax of Lisp programs as a sort LISP. Furthermore, the ASF+SDF system should provide some 'hooks' for interfacing with implementation languages. A conversion must exist which transforms the datastructure representing a program in that language into, for instance, the textual representation of that program. Also there must be a connection back to the system allowing one to parse and reduce terms from within this implementation language.

Clearly, these modules defining the syntax, and the corresponding hooks cannot be made by the end user, because the ASF+SDF system is involved at a deep level. Remember however, that this needs to be done once only, for a particular implementation language. The end user then merely imports this module from the library with available languages.

For every sort which now occurs in the context of a hybrid specification the user has to define injections from that sort to the implementation language and vice versa. These functions must be defined for all sorts which have a hybrid implementation, but also for all parameter sorts (the formal parameter that is, otherwise transparency would be at risk). For each sort, these functions are defined with a single declaration, the special status of which is indicated with the pragma `alien`. For example, a user could define the following SDF rule.

```
NAT     -> LISP    {alien}
```

From this rule the following three function declarations will be created:

```
(1)           "LISP" LISP -> NAT
(2)           "NAT" LISP -> NAT
(3)           "<<" NAT ">>" -> LISP
```

The numbers have been added for reference only. The meaning and the use of these functions will be commented upon shortly.

Furthermore, the kernel syntax of the equation sections of modules (that is the syntax in which equations and conditions etcetera have been defined) is used in the following definition, which occurs in the module Lisp.

```
(4)           "<<" Condition ">>" -> LISP
```

As far as syntax is concerned, these are the only additions for hybrid specifications.

Hybrid equations – i.e., equations using any of the four functions mentioned above – fall into two categories: plain (hybrid) equations and unpacking equations.

- Plain (hybrid) equations define the added rules ($\prec$-rules and cpl-rules) of a hybrid specification.

  A specification can contain as many of these equations as the user may seem to need. In particular more than one may exist for one function symbol, and hybrid equations may occur in conjunction with plain equations.

- Unpacking equations define the function `unpack` needed to convert internal constants to the original signature ($\prec$-matching). This means that unpacking equations are never used as rewrite rules; they are used only when the EQM encounters an internal constant while it is matching.

  For every sort for which internal representations occur, exactly one unpacking equation must be present, and it should not have conditions. Technically these constraints are not essential. The main reason is one of efficiency. Unpacking equations are used relatively often, during rewriting. With this restriction they can be implemented efficiently.

  In ASF+SDF unpacking equations are recognized by the fact that their left-hand side has an alien function symbol of the first variety listed above as outermost symbol.

We will now discuss the use and meaning of each of the four varieties of alien function symbol mentioned above.

1. An alien function of the first variety may occur in either side of an equation or in the conditions. If it occurs in the right-hand side or in the conditions, its use is similar to that of the second variety, and it is discussed under 2 below.

   If an alien function of the first variety occurs in the left-hand side of an equation, its cpl-valued argument should be a single variable. This variable is not an ASF+SDF variable, so it should not be declared as such. Note that these restrictions are not actually checked by the ASF+SDF system.

   If the alien function symbol occurs at root level, then the equation is an unpacking equation, and it is only used during matching, when the outermost function symbol of an internal value is needed. Otherwise the equation is a plain hybrid equation.

   In both cases the meaning of this occurrence is the same. It can only match with an internal representation. The cpl variable is bound to (temporarily assigned to) the cpl value found in that internal representation.

2. An alien function of the first or second variety may occur in the right-hand side of any equation (both unpacking and plain) or in the conditions of a plain equation. Its argument can be any cpl expression. It may occur anywhere in the right-hand side and in the conditions, and more than one such subterms may occur.

   The meaning of such an occurrence is straightforward. The cpl expression is evaluated (it may use the variables mentioned under 1, above). If the outermost function symbol was of the first variety, then the result is an arbitrary cpl value, which will then be packed into an internal representation of the proper sort. If the outermost function symbol was of the second variety, it must already be a valid term of the proper sort (retrieved from some datastructure for instance). In this case it is passed as it is.

3. Sometimes it is necessary to consult the ASF+SDF system from within a cpl expression as mentioned under 2 above. This happens for example, if a value of a parameter sort must be returned. That value is not present in the cpl, so it must supplied by the ASF+SDF system.

   For this purpose functions of the third variety are available. The argument of these functions is an arbitrary open term (of the proper sort). The variables in it are ASF+SDF variables, which should already have been matched against a value, either when the left-hand side was checked, or during validation of a condition. This restriction is identical to that imposed on normal (non-hybrid) right-hand sides. Note that the open term may not introduce supplementary variables (even if the variable occurs in a condition).

   The use of the third variety will become apparent in the examples below.

4. In a context similar to that of the third variety, we sometimes need to check conditions on parameter sorts. For instance, suppose the equality of two terms of a parameter sort must be checked. It is not sufficient merely to compare their term representations, because this would only constitute syntactic equality. Again, the ASF+SDF system must be consulted.

   The function of the fourth variety gets two open terms as arguments (as components of a single `Condition`). Both terms are instantiated and normalized and then compared, quite similar to conditions of normal equations. If the comparison succeeds, the entire expression yield the cpl value `true` (Lisp `t`), otherwise `false` (Lisp `nil`) is returned.

   As with the third variety, no supplementary variables may be introduced in these conditions. All variables occurring should appear in the surrounding equation.

As can be seen from the function definitions, hybrid terms may occur embedded in each other. This is allowed, and sometimes quite necessary. Below we will see an example of embedded hybrid terms.

We have now presented the notation and meaning of hybrid ASF+SDF specifications. We will conclude this section with a specification in which natural numbers and association tables are defined. The natural numbers are defined in the $s$-$z$ notation. The functions $+$ and $*$ are defined with hybrid rules. To demonstrate the transparency of our method, an equation has been added over the symbol $s$, and a separate function $fac$ has been defined using the $s$-$z$ notation without any knowledge of the hybrid aspects.

The specification for tables has two parameters: keys and values. It defines association tables where values may be stored under some key in a table. Then, given some key, the value stored under that key can be retrieved from a table. In our specification all entries for one key are maintained, but the one returned is always the last one entered.

A default value should exist which can be returned when a non-existent entry is requested.

Note that we do not claim this specification to be efficient. Due to equation [3] we have specified natural numbers modulo 10, but possibly a more efficient specification could be made in another fashion. Similarly, the specification of tables is general, but not efficient. With this functionality plain equations would probably give better performance. One should however imagine the gain in efficiency if hash tables, or B-trees would be used in the implementation.

In the specification below, line numbers are for reference purposes only, they are not part of the specification.

```
imports Booleans Lisp
exports
  sorts
    NAT TAB VAL KEY
lexical syntax
  [ \t\n]      -> LAYOUT
context-free syntax
  VAL                          -> LISP    {alien}
  KEY                          -> LISP    {alien}
  NAT                          -> LISP    {alien}
  TAB                          -> LISP    {alien}
  "0"                          -> NAT
  "s" NAT                      -> NAT
  NAT "+" NAT                  -> NAT
  NAT "*" NAT                  -> NAT
  "fac(" NAT ")"               -> NAT
  "mt-tab"                     -> TAB
  "add" VAL "on" KEY "in" TAB  -> TAB
  "lookup" KEY "in" TAB        -> VAL
  "eq(" KEY "," KEY ")"        -> BOOL
  "default"                    -> VAL
variables
  x      -> NAT
  key    -> KEY
  val    -> VAL
equations
1    [1]  0 = LISP 0
2    [2]  s LISP x = LISP (1+ x)
3    [3]  s s s s s s s s s s 0 = 0
4    [4]  LISP x + LISP y = LISP (+ x y)
5    [5]  LISP x * LISP y = LISP (* x y)
6    [6]  fac(0) = s 0
```

```
7    [7]  fac(s x) = fac(x) * s x
8    [8]  mt-tab = LISP nil
9    [9]  add val on key in LISP tab
10        = LISP (acons << key >> << val >> tab)
11   [10] lookup key in LISP tab
12        = VAL (block ()
13                (while tab
14                  (when << eq(key, KEY (caar tab)) = true >>
15                    (return (cdar tab)))
16                  (nextl tab))
17                (return <<default>>))
18   [11] LISP x = NAT (if (zerop x)
19                      << 0 >>
20                      << s LISP (- x 1) >>)
21   [12] LISP tab
22        = TAB (if tab
23                << add VAL (cdar tab) on KEY (caar tab)
24                       in LISP (cdr tab) >>
25                << mt-tab >>)
```

Some comments about this specification are in order.

- The Lisp function `acons` (line 10) gets two values and a list of pairs, and puts the pair of the two values at the head of the list.
- In lines 10, 14, 17 and others, we see examples of embedded hybrid terms. In line 14, for instance, the equality of two keys is determined using the function of the fourth variety. One of the keys however, is in a cpl datastructure, so it must be fetched by the EQM.
- In line 17 the search for the required entry has failed. Now the default value must be returned. This value is not accessible directly from the generator, so a function of the third variety is used in order to consult the ASF+SDF system.
- Equation [11] defines unpacking for numbers. Observe that only a single symbol is unwrapped. More symbols will be unwrapped only if the (recursive) matching encounters the new subterm again.

# Chapter 2

# Term Rewriting Systems

## Implementations in Prolog,

## Reduction Strategies

Algebraic specifications can be implemented by compiling them into Prolog programs. We give an overview of some existing (semi-) automatic implementation methods based on Prolog, and pay particular attention to the implementation of reduction strategies.

## 2.1 Introduction

Implementing algebraic specifications is interesting from two completely different points of view. One point of view is that algebraic specifications should be a means for the development of production software. One is eventually interested in the efficient execution of specifications. Since the efficiency of software development using algebraic specifications is high, compared to that using, say, conventional programming languages, one is particularly interested in the efficiency of the execution of specifications.

The second point of view is that algebraic specifications are merely a tool in software practice, helpful to annotate or clarify software, or to construct and check prototypes. In this case one is only interested in the execution of an algebraic specification in as much as this is necessary to validate it. The emphasis lies on the efficiency of the compilation process rather than on execution speed. Furthermore, proper debugging facilities must exist and the "compiled code" must resemble the algebraic specification closely enough to allow unhampered debugging. Observe that all these properties are fulfilled if the specification is interpreted directly (for instance as a term rewriting system, see below).

We will refer to this second point of view as "prototyping" as opposed to the first, which one might call "constructing production software". Now we will look at an example from each of these two points of view.

The semantics of programming languages can be defined by means of algebraic specifications (see for instance [3]).

- The execution of such a specification would then boil down to the evaluation of programs written in the specified language. Hence, the execution of the algebraic specification would be equivalent to having an interpreter or compiler for the specified language. If algebraic specifications can be executed efficiently, then the formal specification of the semantics of a programming language (i.e., its defining document) would constitute an interpreter or compiler for that language.

- Slightly less far reaching would be our second point of view, prototyping. We would merely want to execute the specification in order to validate it (see whether it actually specifies the functionality we wanted to specify).

What does implementing a specification mean? An algebraic specification defines an equivalence relation on every algebra which has the same signature as the specification, but we will restrict ourselves almost always to the algebra of terms over the given signature. Now there are several notions of implementations. To name a few:

- given a term, compute its value;

- given two ground terms (terms without variables), decide whether they have the same value (i.e., whether one can prove using the equations that they are equal);

- given a set of open equations (equations between terms containing variables), find substitutions for the variables that solve these equations;

- decide whether two open terms are equal in the initial algebra of the specification.

There is no algorithm that implements algebraic specifications in the most general sense described above: equality, in equational logic, is often not decidable. To be more precise, algebraic specifications such as ASF-specifications define *semi-computable* but not computable algebras (in initial algebra semantics) [5].

A central concept is the equality of terms. Algorithms which implement term equality fall into two broad categories.

- The first approach is to use the algebra of terms in the implementation. Terms can be represented in software by trees, which can then be transformed in a stepwise process, where, associated with each step, is one equation in the specification. The generated transformation process constitutes a proof of equality of terms in equational logic.

  This way to implement specifications could be called **interpretational**, and in this chapter we will mainly look at such implementations.

- The second approach is to consider other algebras than the algebra of terms. For some algebras an efficient model may be available. Consider for instance the ring of integers modulo some natural number. This algebra is a model for most specifications of integers. On the other hand, any computer can efficiently evaluate terms in this model.

  In this approach, a representation of a term in one particular model is used. That representation, an object in the model, is manipulated, yielding some other object. The term which is represented by this last object is then reconstructed. The manipulation of representations does not have to be a stepwise process, where each step is validated by a number of equations. The manipulations just happen to implement the model (in the same way as bit string manipulations can implement the algebra of integers).

  This approach to the implementation of algebraic specifications is called **compilational** because a term is compiled to a representation in some model.

  We will not consider this topic in great detail in this chapter. The 'compilational' method described in Section 2.2.1 is, strictly speaking, compilational in the sense described above. Its execution model however, is the data domain of Prolog. Within the limited use described in Section 2.2.1, this domain is isomorphic to the algebra of terms.

  In Chapter 4 a method will be described which allows one to create hybrid implementations. A hybrid implementation combines compilational and interpretational methods in an interpretational framework.

When considering interpretational approaches to the implementation of algebraic specifications one can still vary the degree of preprocessing of the specification. In some cases, this preprocessing introduces compilational aspects in an essentially interpretational implementation.

Restricting our attention to interpretational methods, one can, roughly speaking, distinguish three methods for their implementation:

- Interpreting the specification directly. Interpreting a specification means: consider the specification as a term rewriting system (by giving a direction to equations). The remainder of this chapter will mainly be devoted to this topic, so we will not go into detail here. For now we only mention that this process of orienting the equations can be done manually, or semi-automatically as in the REVE Term Rewriting System Generator [33, 14] (REVE changes the set of rewrite rules, and this could be regarded as a compilational aspect). The advantages of this approach are the absence of preprocessing overhead and the similarity of the text of the specification and the interpreted code. The major disadvantage is the inefficiency of the resulting execution.

- Translating the specification to some logic programming language such as Prolog. Some confusion may arise at this point. In a compilational approach, the specification is analyzed and then mapped onto the semantics of, say, Prolog. In the interpretational approach, the specification appears as a number of Prolog facts, used by a small Prolog program which executes the stepwise interpretation discussed earlier. This Prolog program is an interpreter. An advantage of this approach is the slightly increased efficiency of the interpreter. A disadvantage is that Prolog does not support modularity. Hence debugging at Prolog code level will be unnecessarily complicated.

- Translating the specification to some general purpose programming language such as Modula-2 [39]. This approach is analogous to the preceding one with a different target language. The main advantage is another gain in efficiency of the interpretation. A disadvantage is the complexity of the translation process. This obviously depends on the degree of preprocessing, and any level between purely interpretational and purely compilational could, in principle, be achieved. Another problem is the correctness of the translation process which is hard to establish, especially since formal semantics of the target languages are rarely available. Problems in this area are addressed in for instance [9]. Remarkable in this case is that the modular structure of the object code still resembles that of the source code very closely; most contemporary algorithmic languages (Modula-2 [39], Ada [2]) support modularity.

As we have seen, algebraic specifications define semi-computable datatypes. If we want to implement them, it is of importance to consider subclasses, for which there exists a more or less straightforward implementation.

The class of algebraic specifications that can (and subsequently will) be interpreted as **term rewriting systems** (TRSs), is an important example. To be able to interpret a specification as a TRS, it must obey the following two rules:

- No variable may occur on the right-hand side (of the equation) which does not occur on the left-hand side.
- The left-hand side of an equation may not be a sole variable (it must contain at least one function or constant symbol).

Such a specification can be interpreted as a (generally incomplete) TRS, where an equation $s = t$ is interpreted as a rewrite rule $s \rightarrow t$, meaning that $s$ can be rewritten to $t$, but not the other way around (this will be made precise shortly).

Sometimes, each equivalence class contains exactly one term that cannot be rewritten, a so-called normal form. Term equivalence can then be determined by repeatedly applying rewrite rules to two terms, until both are irreducible (note that this does not need to happen, rewriting could go on indefinitely). The original terms were in the same equivalence class precisely if their normal forms are syntactically equal.

Equational logic proper is left at this point. The relation $\twoheadrightarrow$ which is the reflexive transitive closure of $\rightarrow$ is not symmetric (as is $=$). The gain of this "sidetrack" is that a decision procedure for equality does exist under these constraints.

Actually, moving our attention to the theory of TRSs is not entirely a sidetrack. The Knuth-Bendix algorithm automatically transforms an algebraic specification into a complete TRS under certain conditions. It may stop with success, loop forever, or stop with failure. In the first case it gives a decision procedure for the equality problem in the original specification, in the second case a semi-decision procedure. For more details see [26] or [24].

From here on, we will only look at algebraic specifications that can be viewed as TRSs and we will interpret any equation $s = t$ as if it were a rewrite rule $s \rightarrow t$.

### 2.1.1 Term rewriting systems $\bowtie$[1]

We will now look at the theory of TRSs to indicate classes of algebraic specifications for which equivalence is decidable or can be computed with some guaranteed efficiency. We will first give a brief overview of the terminology; a full introduction to TRSs can be found in [21] and [32]. In this section we will only consider TRSs over a single sort. The introduction of many-sorted TRSs involves technical details rather than theoretical problems.

We will use the letters $a$, $b$, $c$, etc. for constants (function without arguments); the letters $x$, $y$, $z$, etc., for variables; the letters $f$, $g$ etc. for non-constant functions and the letters $s$, $t$, $r$, etc., for terms. In particular we will not "declare" variables and function symbols for every example TRS.

A TRS consists of a **signature** (specifying function symbols with arity, constant symbols and variables) and a set of **reduction** or **rewrite rules**, such as $s \rightarrow t$ (from now on we will write this as $s = t$), where $s$ and $t$ are terms. A term is a constant, a variable or a function symbol with the proper number of arguments, which must again be terms. A term containing variables is called **open**, otherwise it is called a **ground term**. As mentioned before, reduction rules $s = t$ must meet two requirements: $s$ should not be a single variable and any variable occurring in $t$ should already have occurred in $s$. A **substitution** "assigns" values (terms) to the variables in a term (it is a function from the set of variables to the set of terms). An **instantiation** of an open term is the term one obtains by substituting, for each variable, its assigned value (notation: $s^\phi$, where $s$ is an open term, and $\phi$ is a substitution).

If $s = t$ is a reduction rule and if $s^\phi$ is an instantiation of $s$ using a substitution $\phi$ then $s^\phi \rightarrow t^\phi$ is a basic reduction step. In this reduction step $s^\phi$ is called the **redex** (reducible expression). A **context** $C[\,]$ is a term with a hole, i.e., an unknown subterm, in it. If $s^\phi \rightarrow t^\phi$ is a basic reduction step and if $C[s^\phi]$ is the term obtained by putting $s^\phi$ in the hole in $C[\,]$ then $C[s^\phi] \rightarrow C[t^\phi]$ is a reduction step, with the indicated occurrence of $s^\phi$ as redex. Occasionally

[1] Only the first part of this section has already been discussed.

we will also need terms with more holes. We still call those terms contexts. For instance, when $s$ is an open term we denote the context which is obtained by replacing every variable in $s$ with a hole by $s[\ ,..., \ ]$.

If $s = t$ is a reduction rule, the context $s[\ ,..., \ ]$ is called a **redex pattern**. For instance, if $p(s(x), y) = s(p(x, y))$ is a reduction rule, then $p(s(\ ),\ )$ is a redex pattern. This redex pattern occurs twice in the term $p(p(s(s(z)), s(z)), p(z, p(s(z), z)))$.

*Figure 2.1. Term with redex patterns*

The relation $\twoheadrightarrow$ is the transitive reflexive closure of the reduction step relation $\rightarrow$. That is, $s \twoheadrightarrow t$ means $s = t$ or $s \rightarrow s'$ and $s' \twoheadrightarrow t$. A term is said to be a **normal form** if it cannot be reduced (i.e., if it does not contain a redex). A term $t$ *has* a normal form if there is some $s$ which *is* a normal form and for which $t \twoheadrightarrow s$.

A TRS is **Noetherian** or **strongly terminating** if no infinite sequence $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow ...$ of reduction steps exists. It is **weakly normalizing** or **weakly terminating** if there exists at least one finite, non-extendible reduction sequence for each term (each term has a normal form). A TRS has the **unique normal form property** when $s \twoheadrightarrow t$ and $s \twoheadrightarrow t'$, where both $t$ and $t'$ are normal forms, implies that $t$ and $t'$ are syntactically equal.

A TRS is **confluent** or **Church-Rosser** if for all reductions $s \twoheadrightarrow s'$ and $s \twoheadrightarrow s''$, there exists a term $t$ and reductions $s' \twoheadrightarrow t$ and $s'' \twoheadrightarrow t$. A TRS is weakly confluent if for all *one-step* reductions $s \rightarrow s'$ and $s \rightarrow s''$ there exist a term $t$ and (multi-step) reductions $s' \twoheadrightarrow t$ and $s'' \twoheadrightarrow t$. Confluence is of importance for implementing TRSs because it says something like “all roads lead to Rome”.

Three facts of importance are:

- A confluent TRS has the unique normal form property.
- A weakly normalizing TRS which has the unique normal form property is confluent.
- A Noetherian TRS which is weakly confluent is also confluent (Newman’s lemma).

A Noetherian confluent TRS is called **complete**.

Yet another important fact is the following: for a complete TRS, the equality of terms is decidable.

A TRS is **left linear** if no variable occurs twice in the left-hand side of a rule (e.g., $eq(x, x) = true$ is not allowed).

A TRS is **non-ambiguous** if no two redex patterns can ever overlap (e.g., $f(g(x)) = x, g(h(x)) = x$ is ambiguous, because the term $f(g(h(c)))$ contains two overlapping redex patterns). Note that redexes also should not overlap at the root, as in the term $or(true, true)$ with the rules $or(true, x) = true$ and $or(x, true) = true$.

TRSs which are both left linear and non-ambiguous are called **regular**. One of the virtues of a regular TRS is that the notion of **descendant of a redex** after the reduction of another redex is well defined. This observation is the basis for the proof of the theorem that regular TRSs are confluent. This is of great importance because regularity can be tested syntactically, whereas confluence is undecidable, in general ([32]).

A term which is to be rewritten may contain many redexes. A **reduction strategy** tells us for any term, which redex, or which set of redexes, is/are to be rewritten. Strategies which indicate more than one redex to be rewritten at one time are called parallel reduction strategies.

We will now look at the most common reduction strategies, using the following TRS as an example:

```
[1] and(true, x)  = x
[2] and(false, x) = false
[3] or(true, x)   = true
[4] or(false, x)   = x
```

- Leftmost outermost: the redex to be rewritten is the one encountered first during a preorder traversal of the term (in its tree representation).

*Figure 2.2. Leftmost-outermost reduction strategy*

- Leftmost innermost: the redex to be rewritten is the leftmost redex that does not contain another redex.

*Figure 2.3. Leftmost-innermost reduction strategy*

- Parallel innermost: all redexes that do not contain another redex are rewritten simultaneously.

*Figure 2.4. Parallel innermost reduction strategy*

- Parallel outermost: all redexes that are not contained in another redex are rewritten simultaneously.

*Figure 2.5. Parallel outermost reduction strategy*

- Full substitution (also called Gross-Knuth): all redexes are rewritten simultaneously.

*Figure 2.6. Full substitution reduction strategy*

The parallel outermost reduction strategy is normalizing (will eventually reach a normal form, if there is one at all) for regular TRSs [35]. The full substitution strategy is of course only applicable to regular TRSs (otherwise redex patterns could overlap). Like the parallel outermost strategy it is normalizing for regular TRSs.

Given a regular TRS, some redex in a term $s$ is called **needed**, if, in any reduction sequence ending in a normal form, that redex or one of its descendants is rewritten.

The importance of needed redexes from the viewpoint of efficiency is clear: whatever the reduction strategy, needed redexes always have to be rewritten at some point, so time spent in rewriting them is never wasted.

Let $C[\ ,\ldots,\ ]$ be a context (with $n$ holes) which itself does not contain a redex. Suppose that for any substitution of the redexes $s_1$ to $s_n$ in $C$, some $s_i$ ($i$ fixed!) happens to be needed, then $i$ is called an **index**. A TRS is called **sequential** if every context which itself contains no redex has an index. It is called **strongly sequential** if this index is independent of the right-hand sides of the TRS; this index is then called a **strong index**. Note that if one could determine an index for any term, then one would immediately have an optimal reduction strategy.

Huet and Lévy [20] showed for regular TRSs that any term which has a normal form contains a needed redex, and that repeated contraction of a needed redex leads to a normal form, if one exists. Also, the property of strong sequentiality is decidable and there exist efficient (implementable) algorithms to determine whether a TRS is strongly sequential (see [20]).

## 2.2 The translation to Prolog

In Section 2.2.1 we discuss three papers in which a particular implementation method for the translation of algebraic specifications to Prolog is described. Then we give a unifying framework in which these (and other) methods can be compared. Finally, we describe a compiler which translates ASF specifications into Prolog programs.

In Section 2.2.2 we discuss the efficiency of various Prolog implementation schemes.

There is currently little or no literature on the implementation of many-sorted, conditional specifications in Prolog. In Section 2.2.3 we present some suggestions on this topic, without attempting to be complete.

In the remainder of Section 2.2 we will denote the Prolog translation of terms $s$ and $t$ by, respectively, $\sigma$ and $\tau$, and we will use the letters $\beta$, $\gamma$, $\nu$ and $\pi$ to denote arbitrary Prolog clauses.

### 2.2.1 A review of some of the literature

In this entire section we only consider confluent Noetherian TRSs unless stated otherwise. These TRSs have the unique normal form property (every term has a unique normal form), and every reduction sequence is finite, ending in a normal form. Algebraic specifications that translate to confluent Noetherian TRSs admit a simple algorithm to compute the value (= normal form) of a term, and to decide whether two terms are equal (by reducing them to their respective normal forms and checking these for syntactic equality).

We use one example to illustrate the different implementation schemes. The example is a specification of binary coded non-negative integers. The constants 0 and 1 represent the binary digits. The infix operator $\wr$ represents the concatenation of digits into numbers. That is, the number 1001 is represented by the term $1 \wr 0 \wr 0 \wr 1$. Note that ASF operators are left associative. Hence $1 \wr 0 \wr 0 \wr 1$ is parsed as $((1 \wr 0) \wr 0) \wr 1$ as indeed it should be. The intended interpretation of the $\wr$ operator is as follows: if `s` is a term whose value is $n_s$ (some natural number) and if `t` is a term with value $n_t$ then the value of the term `s≀t` is $2n_s + n_t$.

In order to keep the example reasonably small, we only define addition.

```
module Binary-numbers
begin
  exports
  begin
    sorts  BIN
    functions
      0     :            -> BIN
      1     :            -> BIN
      _ ≀ _ : BIN # BIN -> BIN
      _ + _ : BIN # BIN -> BIN
  end
  variables
    x,y,z  :                  -> BIN
  equations
    [1]  x ≀ (y ≀ z) = (x + y) ≀ z
```

```
      [2]  0 ≀ x = x
      [3]  0 + x = x
      [4]  1 + 0 = 1
      [5]  1 + 1 = 1 ≀ 0
      [6]  1 + (x ≀ y) = x ≀ (1 + y)
      [7]  (x ≀ y) + z = x ≀ (y + z)
  end Binary-numbers
```

Some comments about this specification, interpreted as a TRS, are in order.

- Since the second argument to the operator $\wr$ is `BIN`, other values than 0 and 1 may occur, resulting in "weird" `BIN`s such as $1 \wr (1 \wr 1)$. Due to rule `[1]` such values are not in normal form and need not concern us.
- Weird numbers starting with 0 are likewise dealt with in rule `[2]`.
- To prove that this TRS is confluent, we first observe that the natural numbers are a model for this specification (this means that all equations are true for natural numbers). The normal forms of this TRS are: the digits 0 and 1, and strings of 0 and 1 constructed with the operator $\wr$, not containing parentheses (using the left-associativity of the $\wr$ operator) and not beginning with a 0. This can be verified easily. We may conclude that no two distinct normal forms have the same value in the model (every natural number has a unique binary representation). From this

we may conclude that the TRS has the unique normal form property and hence that it is confluent if it is Noetherian.

- It is not easy to see that this specification (interpreted as a TRS) is Noetherian. To prove strong termination, note that no reduction rule increases the number of function symbols in a term. As only finitely many terms with no more than a given number of function symbols exist, any infinite reduction sequence must contain some term more than once. Let $s$ be a smallest term (in the sense of number of function symbols) for which a non-trivial circular reduction sequence $s = s_0 \rightarrow \ldots \rightarrow s_n = s$ exists. In view of the foregoing, there can be no reduction step in the sequence which decreases the number of function symbols. Hence, the rules `[2]`, `[3]` and `[4]` are not used. Furthermore, the outermost function symbol of all terms in the sequence must be a $\wr$, because this is the outermost function symbol of all right-hand sides of the rules `[1]`, `[5]`, `[6]` and `[7]` and one of the terms must be rewritten once at its root level (otherwise $s$ would not be a "smallest" recurring term). We conclude that the only rule that can be applied at root level is `[1]`. Each time it is applied however, the number of function symbols in the right argument of the outermost $\wr$ decreases. Consequently the reduction sequence cannot be circular. So the TRS is Noetherian and therefore confluent.

### The interpretational, innermost method of Drosten and Ehrich

In the paper by Drosten and Ehrich [8] the following algorithm to translate an algebraic specification to a Prolog program is described (actually the procedure is slightly more general, allowing for open terms as well. This minor change was made in order to compare this approach to other ones). In order for the method to be applicable, one has to restrict oneself to specifications that give a confluent, Noetherian TRS.

1. For each function symbol $f$ with arity $n$ define a Prolog clause

```
analyze(f(X1, ..., Xn), F) :-
   analyze(X1, Y1),
   ...
   analyze(Xn, Yn),
   normalize(f(Y1, ..., Yn), F).
```

2. For each rewrite rule $s = t$ add a fact

```
rule(σ, τ).
```

   Here $\sigma$ and $\tau$ stand for translations of the terms $s$ and $t$, adhering to the Prolog syntax, which requires variables to begin with a capital and constants with a lower case letter.

3. Finally add the the following two clauses to the program:

```
normalize(X, Z) :- rule(X, Y), analyze(Y, Z).
normalize(X, X).
```

   Obviously these two clauses must appear in the order shown here.

The correctness of the translation above is established by the following theorem:

**Theorem:** If **T** is a confluent Noetherian TRS, and if **P** is the Prolog program resulting from the translation process described above, then a Prolog system executing **P** will terminate for every query `analyze(`$\sigma$`, X)` where $\sigma$ is the Prolog representation of a term over the signature of **T**. After termination, `X` will be instantiated to the normal form of $\sigma$.

That the theorem holds is mildly surprising because almost no constraints exist on the order of the Prolog clauses.

Its validity is due to the fact that only Noetherian TRSs are considered. For such TRSs any reduction strategy is strongly normalizing.

The strategy implemented by Drosten and Ehrich's scheme is leftmost innermost. For each term $f(s_1, ..., s_n)$ to be analyzed (put in normal form), its leftmost argument (possibly containing the leftmost redex) is first put in normal form, then its second argument (which would now contain the leftmost innermost redex), and so on. Only when all arguments are in normal form will a rule be applied to the entire term.

This description immediately suggests the proof method for the theorem. It is based on the observation that, given a term whose subterms are in normal form, the predicate `normalize` will perform a reduction step unless the entire term is in normal form; the proof is based on induction on the length of the leftmost innermost reduction sequence of a term. Since all reduction sequences are finite for Noetherian TRSs, this process terminates.

Now we apply this implementation method to the example of binary numbers. There are four function symbols: 0, 1, + and ≀ with arities 0, 0, 2 and 2 respectively. Since the functions ≀ and + appear as operators in the ASF specification, we choose to declare them as left-associative infix operators in the Prolog program as well. Hence the program contains the clause

```
:- op(500, yfx, ['+', '≀']).
```

According to step 1, we must now add the clauses

```
analyze(0, F) :- normalize(0, F).
analyze(1, F) :- normalize(1, F).
analyze(X≀Y, F) :-
   analyze(X, X1),
   analyze(Y, Y1),
   normalize(X1≀Y1, F).
```

```
analyze(X+Y, F) :-
   analyze(X, X1),
   analyze(Y, Y1),
   normalize(X1+Y1, F).
```

Now, we must add the reduction rules as facts (step 2):

```
rule(X≀(Y≀Z), (X+Y)≀Z).
rule(0≀X, X).
rule(0+X, X).
rule(1+0, 1).
rule(1+1, 1≀0).
rule(1+(X≀Y), X≀(1+Y)).
rule((X≀Y)+Z, X≀(Y+Z)).
```

Finally (step 3) we must add the clauses

```
normalize(X, Z) :- rule(X, Y), analyze(Y, Z).
normalize(X, X).
```

This Prolog program terminates for all queries

```
?- analyze(σ, X).
```

after which `X` is instantiated to the normal form of $\sigma$, where $\sigma$ is a ground term (in Prolog syntax) over the signature of the module `Binary-numbers`.

**The interpretational, outermost method of van Emden and Yukawa**

The paper by van Emden and Yukawa [13] does not specifically aim at Prolog, but rather at logic programming in general. We formulate their method using Prolog. The following translation process is defined:

1. The Prolog program begins with the clauses (in the order shown):

   ```
   analyze(X, X):-canonical(X).
   analyze(X, Z):-step(X, Y), analyze(Y, Z).
   ```

2. For each rule $s = t$ in the TRS add the fact

   ```
   step(σ, τ).
   ```

3. For each function symbol $f$ of arity $n$ ($n > 0$) add the clause

   ```
   step(f(X1, ..., Xn), f(Y1, ..., Yn)) :-
      analyze(X1, Y1), ...., analyze(Xn, Yn).
   ```

   The facts resulting from step 2 should precede the clauses resulting from step 3.

4. Clauses must be added to define the predicate `canonical` which should succeed if and only if its argument is a normal form. Since the TRSs under consideration are Noetherian, such a predicate is decidable. (As a

matter of fact, the procedure described in the previous section specifies exactly such a predicate.) In general the predicate can be evaluated in a more economical fashion than by trying every rule on every subterm. For instance, the canonical predicate for binary numbers might look like

```
canonical(0).
canonical(1).
canonical(X≀0) :- canonical(X), X \== 0.
canonical(X≀1) :- canonical(X), X \== 0.
```

Note that whereas the implementation in the previous section automatically guarantees normal forms to be proper, that is to have every right-hand side argument of the operator ≀ to be either 0 or 1 and the left-hand side of ≀ unequal to 0, we must now specify this explicitly in the `canonical` predicate. In particular, the straightforward approach

```
canonical(0).
canonical(1).
canonical(X≀Y) :-
   canonical(X)
   canonical(Y).
```

would fail, even though the relevant reduction rules are present, because these rules are not tried if the canonical predicate succeeds.

Van Emden and Yukawa prove exactly the same theorem as Drosten and Ehrich about their algorithm. The proof again uses induction.

Van Emden and Yukawa's scheme implements another reduction strategy than that of Drosten and Ehrich. Since rules resulting from step 2 occur before those from step 3, it is an outermost strategy. Every redex is put in normal form as soon as it is recognized. This reduction strategy is not one of the strategies we have discussed in Section 2.1.1.

The transformation of our example specification into its implementation is now straightforward.

```
:- op(500, yfx, ['+', '≀']).

analyze(X, X) :- canonical(X).
analyze(X, Z) :- step(X, Y), analyze(Y, Z).

step(X≀(Y≀Z), (X+Y)≀Z).
step(0≀X, X).
step(0+X, X).
step(1+0, 1).
step(1+1, 1≀0).
step(1+(X≀Y), X≀(1+Y)).
step((X≀Y)+Z, X≀(Y+Z)).
```

```
step(X1≀X2, Y1≀Y2) :-
   analyze(X1, Y1),
   analyze(X2, Y2).
step(X1+X2, Y1+Y2) :-
   analyze(X1, Y1),
   analyze(X2, Y2).

canonical(0).
canonical(1).
canonical(X≀0) :- canonical(X), X \== 0.
canonical(X≀1) :- canonical(X), X \== 0.
```

### The compilational, innermost method of van Emden and Yukawa

In the same paper [13] van Emden and Yukawa describe yet another implementation method, which they refer to as the compilational method. It is slightly more complicated than the previous two methods, and it is applicable to a different class of TRSs. These need no longer be confluent, but must satisfy another requirement, as we shall see shortly.

First, a distinction is made between defined constants and functions on the one hand, and constructors on the other hand. Any function that occurs as outermost function in the left-hand side of any rule in the TRS, is a defined function. Otherwise it is a constructor.

The compilational method can only handle TRSs such that no left-hand side of a rewrite rule contains more than one defined function (it contains at least one of course, its outermost symbol).

Furthermore, no normal form of the TRS should contain a defined function symbol. This means that any term which contains a defined function symbol, should contain a redex.

Now the Prolog implementation is constructed as follows.

- For each defined function $f$ of arity $n$ introduce a corresponding Prolog predicate `pf`, with arity $n+1$.
- For each rule the following three steps are performed.
  1. The rule $f(s_1, ..., s_n) = t$, ($f$ is a defined function) is changed to the Prolog fact

     `pf(`$\tau$`, `$\sigma_1$`, ..., `$\sigma_n$`).`

     Note that we have a predicate `pf` for every defined function $f$.
  2. Suppose that in the Prolog clause

     `pg(`$\tau$`, `$\sigma_1$`, ..., `$\sigma_m$`) :- `$\beta_1$`, ..., `$\beta_k$`.`

     (which is the result of step 1 and initially has an empty body) the argument $\tau$ contains a defined symbol. In this case, let `h(`$\nu_1$`, ..., `$\nu_n$`)` be a subterm of $\tau$ such that

▷ `h` is a defined symbol;

▷ the term $\mathtt{h}(\nu_1, \ldots, \nu_n)$ is not contained in another subterm of $\tau$ which has a defined function as outermost function symbol.

Then that Prolog clause is transformed to the clause

$$\mathtt{pg}(\tau',\ \sigma_1,\ \ldots,\ \sigma_m)\ \mathtt{:-}\ \mathtt{ph}(\mathtt{X},\ \nu_1,\ \ldots,\ \nu_n),\ \beta_1,\ \ldots,\ \beta_k.$$

where `X` is some variable not previously occurring in the clause and where $\tau'$ is obtained by substituting `X` for the subterm $\mathtt{h}(\nu_1, ..., \nu_n)$ in the term $\tau$. This is repeated until the first argument of the head of the clause (and as a consequence of the special property of the TRSs under consideration the entire head of the clause) no longer contains a defined function.

3. If, in the Prolog clause

$$\pi\ \mathtt{:-}\ \beta_1,\ \ldots,\ \beta_k.$$

some $\beta_i$ contains a subterm $\mathtt{h}(\nu_1, \ldots, \nu_n)$ as above, then that clause is transformed into

$$\pi\ \mathtt{:-}\ \mathtt{ph}(\mathtt{X},\ \nu_1,\ \ldots,\ \nu_n),\ \ \beta_1,\ \ldots,\ \ \beta_i',\ \ldots,\ \ \beta_k.$$

where $\beta_i'$ is obtained by substituting `X` for the subterm $\mathtt{h}(\nu_1, \ldots, \nu_n)$ in the term $\beta_i$ (`X` is again a variable not previously occurring in the clause). This is repeated until no such term remains in the body of any clause.

- A term is normalized as follows. The term must have a defined function $f$ as its outermost symbol (this is not a real restriction, as is easily verified). First the query

  $$\mathtt{?-}\ \mathtt{pf}(\mathtt{X},\ \nu_1,\ \ldots,\ \ \nu_m).$$

  is constructed. Then transformation step 3 is performed on the query (viewed as a headless clause).

The theorem proved by van Emden and Yukawa is similar to the ones discussed in the previous sections. The constructed query terminates, `X` having been instantiated to a normal form of $f(\nu_1, ..., \nu_m)$. Note that since the TRS may be non-confluent, no guarantee can be given that the normal form arrived at is unique.

Unfortunately our example specification does not have the property that the left-hand side of each rule contains at most one defined function, so van Emden and Yukawa's compilational method is not applicable to it. In [3], an extension of the compilational method is described which imposes less constraints and is capable of handling this example. The operators $+$ and $\wr$ are both defined functions, as follows from equations `[2]` and `[3]`, but both operators occur in the left-hand sides of equations `[6]` and `[7]`. We will therefore apply the method to another example, namely the natural numbers with 0 and a successor function $s$, addition (+) and multiplication (*).

```
module Successor-numbers
begin
    exports
    begin
        sorts
            SNUM
        functions
            0       :                   -> SNUM
            s       :  SNUM             -> SNUM
            _ + _   :  SNUM # SNUM      -> SNUM
            _ * _   :  SNUM # SNUM      -> SNUM
    end
    variables
            x,y,z  :                    -> SNUM
    equations
      [1]    0 + x        = x
      [2]    s(x) + y     = s(x + y)
      [3]    0 * x        = 0
      [4]    s(x) * y     = (x * y) + y
end Successor-numbers
```

The specified TRS is regular and hence confluent (although we do not need this property now). That the TRS is Noetherian can be proved with induction on the complexity of terms. The defined functions are $+$ and $*$, $0$ and $s$ being constructors. The specification indeed has the required property.

We will now construct the implementation. We will call the Prolog predicates corresponding to the defined function symbols $+$ and $*$, `plus` and `times` respectively.

According to step 1 we start out with the facts:

```
plus(X, 0, X).                              /*1a*/
plus(s(X+Y), s(X), Y).                      /*2a*/
times(0, 0, X).                             /*3a*/
times((X*Y)+Y, s(X), Y).                    /*4a*/
```

Now step 2 must be applied. The first subterm having a defined symbol as its outermost function symbol is `X+Y` in the second fact. This fact is transformed into

```
plus(s(Z), s(X), Y) :- plus(Z, X, Y).      /*2b*/
```

Likewise the fourth fact is transformed to the clause

```
times(Z, s(X), Y) :- plus(Z, (X*Y), Y).    /*4b*/
```

Now, step 3 must be performed. The only clause containing a subterm with a defined symbol in it is clause 4b. This clause is transformed to

```
times(Z, s(X), Y) :-
   times(Z1, X, Y), plus(Z, Z1, Y).        /*4c*/
```

We have now constructed the entire Prolog implementation of the specification `Successor-numbers`:

```
plus(X, 0, X).                              /*1a*/
plus(s(Z), s(X), Y) :- plus(Z, X, Y).      /*2b*/
times(0, 0, X).                             /*3a*/
times(Z, s(X), Y) :-
   times(Z1, X, Y), plus(Z, Z1, Y).         /*4c*/
```

The compilational approach differs from the other implementations not only in the amount of preprocessing needed for the specification, but also in the fact that queries too must be preprocessed. We will show how the term $(s(0) * s(s(0))) + (s(s(s(0))) * s(s(0)))$ is handled.

As described, we start with the query

```
?- plus(X, s(0)*s(s(0)), s(s(s(0)))*s(s(0))).
```

To this query step 3 must be applied, that is, the two occurrences of * must be dealt with. The query is first transformed to

```
?- times(Y, s(0), s(s(0))),
   plus(X, Y, s(s(s(0)))*s(s(0))).
```

and finally to

```
?- times(Z, s(s(s(0))), s(s(0))),
   times(Y, s(0), s(s(0))),
   plus(X, Y, Z).
```

After evaluating this query, with the program we have constructed above, the variable `X` will have been instantiated to the normal form of the term above: `s(s(s(s(s(s(s(s(0)))))))))`.

### The interpretational, parallel outermost method of van Diepen, van der Gaag and Hendriks

In [10] van Diepen, van der Gaag and Hendriks briefly sketch the implementation of a parallel outermost reduction strategy. Remember that that strategy is strongly normalizing even for weakly normalizing TRSs, if they are regular (Section 2.1.1).

The translation is as follows.

1. The Prolog program begins with the clauses (in the order shown)

```
 analyze(T, Res) :-
    step(T, I),
    T \== I, !,
    analyze(I, Res).
analyze(T, T).
```

2. For each rule $s = t$ add the clause

   `rule(`$\sigma$`, `$\tau$`).`

3. Finally, the Prolog program ends with

```
step(T, Res) :- rule(T, Res).
step(T, Res) :-
   T =.. [Func|Args],
   argsstep(Args, Args1),
   Res =.. [Func|Args1].

argsstep([], []).
argsstep([H|T], [H1|T1]) :-
   step(H, H1),
   argsstep(T, T1).
```

By inspecting the Prolog program above we can observe that the predicate `step` succeeds always and that its second argument depends on its first argument (a term) as follows:

- If a rule $s = t$ can be applied to the term at root level (i.e., if a basic reduction step can be performed), then the term is reduced, yielding the one step reduct.
- Otherwise, if the argument is some compound term $f(t_1, ..., t_n)$, the step predicate is applied to the components $t_1$ to $t_n$, yielding the terms $t'_1$ to $t'_n$. Then the term $f(t'_1, ..., t'_n)$ is returned.

We conclude that the predicate step contracts all redexes that are not included in another redex. Hence, a parallel outermost reduction strategy has been implemented.

This implementation uses Prolog's operators `!` and `=..`. The first increases space/time efficiency (see Section 2.2.2.1). The second decomposes compound terms, thereby decreasing the program size by avoiding a clause per function symbol as in step 1 of Section 2.2.1.1 and step 3 of Section 2.2.1.2. Note that the result of steps 1 and 3 does not depend on the specification, not even on its signature, unlike the equivalent Prolog code in the implementations presented in previous sections. The implementations in sections 2.2.1.1 and 2.2.1.2 could be adapted likewise.

After every attempted step a test is done to verify whether a reduction step has indeed taken place. This test plays a similar role to the canonical predicate in Section 2.2.1.2.

The implementation of `Binary-numbers` now becomes:

```
:- op(500, yfx, ['+', '≀']).

analyze(T, Res) :-
   step(T, I),
   T \== I, !,
   analyze(I, Res).
analyze(T, T).

rule(X≀(Y≀Z), (X+Y)≀Z).
```

```
rule(0≀X, X).
rule(0+X, X).
rule(1+0, 1).
rule(1+1, 1≀0).
rule(1+(X≀Y), X≀(1+Y)).
rule((X≀Y)+Z, X≀(Y+Z)).

step(T, Res) :- rule(T, Res).
step(T, Res) :-
   T =.. [Func|Args],
   argsstep(Args, Args1),
   Res =.. [Func|Args1].
step(T, T).

argsstep([], []).
argsstep([H|T], [H1|T1]) :-
   step(H, H1),
   argsstep(T, T1).
```

### A general framework

Recently F. Wiedijk suggested an attractive notation and a general framework in which all previously mentioned Prolog implementations (except the compilational approach by van Emden and Yukawa), and more, are captured. First we will present a translation scheme that implements one particular strategy (leftmost outermost). It consists of the steps GF1, GF2', GF3' and GF4 below. Later we present the steps GF2 and GF3 for arbitrary reduction strategies.

**GF1** The Prolog program begins with the operator declaration

```
:- op(700, xfx, ['--->', '--->>']).
```

declaring two infix operators with appropriate priority and associativity. Their intuitive meaning is a one-step reduction and the corresponding reflexive transitive closure of this one-step reduction.

**GF2'** For each rule $s = t$, add the fact

$\sigma$ `--->` $\tau$.

where $\sigma$ and $\tau$ are again the terms $s$ and $t$ written in Prolog syntax.

**GF3'** For each function symbol $f$ (with arity $n$) add the following $n$ clauses:

```
f(X1, X2, ..., Xn) ---> f(Y1, Y2, ..., Yn) :-
   X1 ---> Y1, X2 = Y2, ..., Xn = Yn.
 ...
f(X1, ..., Xi, ..., Xn) ---> f(Y1, ..., Yi, ..., Yn) :-
   X1 = Y1, ..., Xi ---> Yi, ..., Xn = Yn.
 ...
```

```
f(X1, ..., Xn-1, Xn) ---> f(Y1, ..., Yn-1, Yn) :-
   X1 = Y1, ..., Xn-1 = Yn-1, Xn ---> Yn.
```

Note that the non-diagonal entries are all equalities of variables. The same end could be achieved by giving the corresponding variables the same name as in

```
f(X1, ..., Xi, ..., Xn) --->
                   f(X1, ..., Xi-1, Yi, Xi+1, ..., Xn) :-
   Xi ---> Yi.
```

The former notation was chosen with a certain amount of foresight as we will see shortly.

**GF4** The program ends with the following clauses (in the order shown)

```
X --->> Z :- X ---> Y, Y --->> Z.
X --->> X.
```

The facts in step GF2' should precede the clauses in step GF3'. The intended meaning of the two operators is obvious. `--->` is the operator which will perform one reduction step according to the leftmost outermost reduction strategy (compare with $\rightarrow$ ), `--->>` is its reflexive transitive closure $\twoheadrightarrow$, as can be seen immediately from its definition in step GF4.

The query

`?-` $\sigma$ `--->> V`

where $\sigma$ is a ground term in the signature of the TRS will terminate, having instantiated `V` to the normal form of $\sigma$. This is done by repeatedly attempting to satisfy

```
Xi ---> Xi+1
```

where is `Xi` is the i step reduct of $\sigma$, and where `Xi+1` is not yet instantiated. As soon as `--->` fails, `--->>` succeeds, yielding the last established reduct.

The predicate `--->` succeeds if a basic reduction step can be performed on its left argument, by instantiating its right argument with the contracted redex. If no basic reduction step can be done, and if the argument is a composite term (otherwise it is a constant on which no reduction rule is applicable), a reduction step is attempted in the first subterm, leaving all other subterms. If this fails a reduction step is attempted in the second subterm and so on. If `--->` eventually fails, its argument contained no redex, and was already in normal form. If it succeeds it has indeed performed a reduction step. This process must terminate for Noetherian TRSs.

At this point it is clear that a formal proof can be constructed similar to the proofs sketched in the previous sections.

The generalizations to the Prolog implementation above are twofold. All approaches redefine the meaning of `--->` to be a reduction step in some different strategy.

**Innermost strategies** The facts in step GF2' implement basic reduction steps. For some reduction strategies it is necessary to look for other redexes simultaneously (for instance, the Gross-Knuth reduction strategy reduces all redexes).

These simultaneous reductions can be performed by attempting reduction steps in the subterms of some term which is to be reduced. This is done by renaming the variables in the right-hand sides of the facts created in step GF2', and by showing how the new subterms are derived from the old subterms.

In order to guarantee the validity of the Prolog program, predicates attempting reduction steps in the subterms should not fail if no reduction step is possible (`--->>` is such a predicate, `--->` is not).

Obvious choices would be a predicate which performs zero or one reduction steps and a predicate which performs zero or more reduction steps. The latter is of course `--->>`.

The former is defined as:

```
:- op(700, xfx, '-=->').
X -=-> Y :- X ---> Y.
X -=-> X.
```

Our first generalization can now be presented. Instead of step GF2' above, we will use the transformation step GF2 below.

**GF2** For each rule $s = t$, where $s$ contains exactly the $n$ variables $x_1$ through $x_n$, the program must contain the clause

$\sigma$`[Xi|`$xi$`] ---> `$\tau$`[Yi|`$xi$`] :-`
`X1 `$\rightarrow_1$` Y1, ..., Xn `$\rightarrow_1$` Yn.`

where $\sigma$`[Xi|`$xi$`]` is the Prolog term corresponding to $s$, with each variable $x_i$ substituted by the Prolog variable `Xi`, and where $\rightarrow_1$ is the predicate `=` (effectively resulting in step GF2'), the predicate `-=->` or the predicate `--->>`.

Note that $\sigma$ may contain variables not occurring in $\tau$ (not vice versa though). Obviously, for any such `Xi` no corresponding `Yi` exists. Hence, the corresponding goal `Xi`$\rightarrow_1$`Yi` needs not to be present.

**Parallel strategies** In step GF3' of the general framework a single step is performed in one of the arguments of a composite term, if that term is not a redex. Different reduction strategies may require more steps to be performed, either in the same argument (innermost strategies) or in other arguments (parallel strategies).

As before, special care must be taken in order still to be able to guarantee the validity of the program. Notice that in step GF3' a reduction step is attempted in the diagonal arguments (the i-th clause reduces the i-th argument). Hence, the predicate that concerns the diagonal argument should at least make one reduction step (it should fail if this is impossible). The predicates concerning other arguments should always succeed (possibly after performing some reduction steps). We have already seen the latter to be `=`, `-=->` or `--->>`. The options for the former predicate are to perform exactly one step (the predicate `--->`), or to perform as many steps as possible, but at least one. This is defined as follows.

```
:- op(700, xfx, '->->>').
X ->->> Z :- X ---> Y, Y --->> Z.
```

The generalization of step GF3' is now as follows.

**GF3** For each function symbol $f$ (with arity $n$) add the following $n$ clauses.

```
f(X1, ..., Xn) ---> f(Y1, ..., Yn) :-
   X1 →2 Y1, X2 →3 Y2, ..., Xn →3 Yn.
 ...
f(X1, ..., Xn) ---> f(Y1, ..., Yn) :-
   X1 = Y1, ..., Xi →2 Yi, ..., Xn →3 Yn.
 ...
f(X1, ..., Xn) ---> f(Y1, ..., Yn) :-
   X1 = Y1, ..., Xn-1 = Yn-1, Xn →2 Yn.
```

These clauses will first attempt to perform one (or more) reduction steps in the first argument. The predicate $\rightarrow_2$ fails if no step can be done, it succeeds if at least one step can be done, in which case either one (`--->`) or as many as possible (`->->>` ) steps are actually done. If this fails, the second argument is tried, and so on. When this eventually succeeds for the i-th argument, we know the arguments 1 to $i-1$ to be in normal form. The other arguments may still be reducible. The predicate $\rightarrow_3$ always succeeds and may either perform no reductions (`=`), or one reduction, if possible (`-=->`), or as many reductions as possible (`--->>`).

**Filling in the general framework** There are many combinations of `=`, `-=->` or `--->>` with `--->` or `->->>`. We will not look at all combinations. As a matter of fact all reduction strategies we have seen so far are covered by the following three combinations:

- $\rightarrow_2$ is `--->` and $\rightarrow_3$ is `=`; or
- $\rightarrow_2$ is `--->` and $\rightarrow_3$ is `-=->`; or
- $\rightarrow_2$ is `->->>` and $\rightarrow_3$ is `--->>`.

The two presented generalizations for innermost and parallel strategies, giving three options each, result in nine possibilities altogether. Figure 2.7 shows these nine possibilities, relating them to the other implementation methods we have discussed in this chapter.

| $\to_3$<br>$\to_2$<br>$\to_1$ | `=`<br>`--->` | `-=->`<br>`--->` | `--->>`<br>`->->>` |
|---|---|---|---|
| `=` | leftmost outermost | parallel outermost (van Diepen *et al.*) | van Emden and Yukawa |
| `-=->` | | Gross-Knuth | |
| `--->>` | leftmost innermost | | parallel innermost (Drosten and Ehrich) |

*Figure 2.7. Comparison of several implementation techniques*

Actually, the transformation steps GF2 and GF3 may be exchanged, resulting in nine more reduction implementations. These new implementations, however, do not lead to new reduction strategies.

For ambiguous TRSs care must be taken with the implementations in the two lower lines of Figure 2.7 (where $\to_1$ is unequal to `=`). Redexes contained in and overlapping with other redexes are not always reduced as can be seen in the following example:

```
TRS:
[1] p(s(x)) -> x
[2] s(p(x)) -> x
```

Now consider the term `p(s(p(0)))`. The redex in rule `[1]` is recognized, and a reduction step is attempted in the subterm `p(0)`, which fails. Then rule `[1]` is applied. The redex `s(p(0))` is never recognized. It is clear that the parallel contraction of redexes with overlapping redex patterns is meaningless.

Due to the phenomenon we have just observed, the parallel innermost strategy in table one is not the same as that of Drosten and Ehrich (Section 2.2.1.1) for ambiguous TRSs.

This difference stems from the fact that Drosten and Ehrich do not even try basic reduction steps before normalizing the arguments of a compound term.

Now we will use this implementation method for our example. We will only show the parallel outermost reduction strategy (all nine implementations are very similar).

```
:- op(500, yfx, ['+', '≀']).
:- op(700, xfx, ['--->', '--->>', '-=->']).

X≀(Y≀Z) ---> (X + Y)≀Z.
0≀X ---> X.
```

```
0 + X ---> X.
1 + 0 ---> 1.
1 + 1 ---> 1≀0.
1 + (X≀Y) ---> X≀(1 + Y).
(X≀Y) + Z ---> X≀(Y + Z).

X1 + X2 ---> Y1 + Y2 :- X1 ---> Y1, X2 -=-> Y2.
X1 + X2 ---> Y1 + Y2 :- X1 = Y1, X2 ---> Y2.
X1≀X2 ---> Y1≀Y2 :- X1 ---> Y1, X2 -=-> Y2.
X1≀X2 ---> Y1≀Y2 :- X1 = Y1, X2 ---> Y2.

X --->> Z :- X ---> Y, Y --->> Z.
X --->> X.
X -=-> Y :- X ---> Y.
X -=-> X.
```

### The ASF compiler

An interesting research project in the context of Prolog implementations is the ASF compiler of P.R.H. Hendriks [16, 17].

As can for instance be seen in [3, 17], the semantics of a modular ASF specification is defined in terms of a normalization procedure and the initial algebra semantics of normalized specifications. Given a specification $S$ and one particular module $M$ in $S$, this procedure constructs a module $N$, with the same exported signature as $M$, by repeatedly resolving imports and binding parameters in $M$. The semantics of module $M$ (which contains imports) is defined to be equal to the initial algebra of $N$ (which does not contain imports). Exceptions are modules which after normalization still contain unbound parameters, and modules which after normalization contain a sort for which no ground term exists. No semantics has been defined for such modules.

The ASF compiler performs this normalization procedure for each of the modules in a specification, checking the specification along the way for correctness. The output of this process is an abstract representation of the normalized modules. Subsequently, these normalized modules are translated into Prolog clauses.

Two translation methods have been implemented so far. The first method is the one described in Section 2.2.1.4. and results in a parallel outermost reduction strategy which is strongly normalizing for all regular TRSs (and trivially also for Noetherian TRSs). The regularity condition is not checked in that implementation.

The second method is an extension of the compilational method described in Section 2.2.1.3. and results in a leftmost-innermost reduction strategy (which of course is only strongly normalizing for Noetherian TRS's). This second method turns out to be far more efficient than the first one.

### 2.2.2 Efficiency

In this section we consider the efficiency of implementation methods for algebraic specifications based on rewriting. These implementations will rewrite a term, or its subterms, until it is in normal form. The efficiency of an implementation is expressed in the number of reduction steps it needs to perform.

Although most issues discussed in this section concern implementations of TRSs in general, we will use Prolog to specify algorithms.

A reduction strategy is an algorithm (formally a function) which, when given a TRS and a term, determines a redex in that term, and the rule in the TRS which must be applied to that redex. In case of a parallel strategy, a set of redexes and rules is given.

Counting the number of reduction steps in order to determine the efficiency of a strategy can be misleading if one is interested in execution time. A strategy may be efficient in the number of reduction steps it performs, and yet it may do a lot of work per reduction step, determining which redex to reduce next. The "optimal" reduction strategy, which for each reduction step determines all possible reduction sequences ending in a normal form, and then suggests the first step of the shortest sequence is hardly optimal in execution time.

In Section 2.1.1 we have seen that an optimal reduction strategy exists for strongly sequential TRSs. Unfortunately there currently exists no simple algorithm which determines whether a TRS is indeed strongly sequential. Furthermore, the class of strongly sequential TRSs is very restrictive and optimality for such strategies is a relative notion, as the following example of the non-negative integers with `0`, successor and addition shows.

TRS 1:
```
[1] 0 + x       = x
[2] s(x) + y    = s(x + y)
```

TRS 2:
```
[1] 0 + 0       = 0
[2] 0 + s(y)    = s(y)
[3] s(x) + 0    = s(x)
[4] s(x) + s(y) = s(s(x + y))
```

These specifications define the same data type and they are both regular. TRS1 is strongly sequential; the first argument of `+` and the argument of `s` are always needed. Nevertheless, TRS2 is more efficient when, for instance, the parallel outermost reduction strategy is used. The term `s(s(0))+0` is in that case normalized in one reduction step in TRS2, but in TRS1 it takes three steps.

We will now present some efficiency increasing methods which are helpful even when optimality cannot be achieved.

### Cuts

This section is only applicable to Prolog implementations. The evaluation of Prolog programs is based on backtracking. This may lead to reduction steps being retried in a different manner. The reduction strategy itself, however, already tries all possibilities. Hence, backtracking over a reduction step is unnecessary, and a cut (`!`) may be inserted after every reduction step to save time.

For instance, the definition of `--->>` now becomes

```
X --->> Z :- X ---> Y, !, Y --->> Z.
X --->> X.
```

A proof of the correctness of this modification would rely on the confluence of the TRS, which guarantees that by making a reduction step, no potential reduct becomes unreachable.

### Graph reduction

In Section 2.2.1.3 we presented a TRS defining the non-negative integers with the constant `0` and functions `s`, `+` and `*`. We will repeat the equations from that module:

```
[1]    0 + x      = x
[2]    s(x) + y   = s(x + y)
[3]    0 * x      = 0
[4]    s(x) * y   = (x * y) + y
```

In any straightforward top-down implementation (such as leftmost outermost, or parallel outermost), rule `[4]` of `Successor-numbers` poses an efficiency problem. Consider, for instance, the term `s(t1)*t2`. When rule `[4]` is applied to this term it would mean making a copy of `t2`, which then might have to be normalized twice. Normalizing `t2` beforehand does not always solve the problem because its normal form may not even be needed (like in `(0 + 0) * (s(t1) * t2)`).

A solution to this problem is the following. Instead of making two copies of the subterm `t2` we simply create two references to an object which initially contains the term `t2` but which can be changed to contain reducts of `t2` and eventually its normal form.

One can easily see how this works when one considers the tree representation of terms in Figure 2.8. A term is no longer represented as a tree, but rather as an acyclic directed graph.

*Figure 2.8. Graph reduction*

In traditional programming languages graph reduction can easily be implemented using pointers. Since we focus on Prolog, we will use this language to present an implementation of graph reduction. Unfortunately Prolog does not support pointers or global variables. The only "assignable" item is an uninstantiated variable, which can be made to "refer" to some term by unifying it with that term.

We present a modification of the generalized Prolog implementation given in Section 2.2.1.5. It is based on an algorithm by J.C. Mulder. All terms, including subterms of terms, are associated with an uninstantiated variable. Whenever a term is reduced, its variable is instantiated with the reduct (which itself is again associated with an uninstantiated variable). Terms which are "copied" by the application of a rewrite rule will now contain the same uninstantiated variable. As soon as one copy of that variable is instantiated with a value, all other copies will refer to the same value.

The association of a term with a variable can be achieved by the introduction of a new function, not yet occurring in our TRS. We will call this function `red` because its second argument will eventually contain the list of reducts of the first argument.

1. First we define two translations between conventional terms and terms in the new representation containing uninstantiated variables:

   ```
   trans(U, red(V, _)) :-
      U =.. [Func|Args],
      trans_list(Args, Newargs),
      V =.. [Func|Newargs].

   trans_list([], []).
   trans_list([Arg|Args], [NewArg|NewArgs]) :-
      trans(Arg, NewArg),
      trans_list(Args, NewArgs).

   back(red(V, _), U) :-
      V =.. [Func|Args],
      back_list(Args, Newargs),
      U =.. [Func|Newargs].

   back_list([], []).
   back_list([Arg|Args], [NewArg|NewArgs]) :-
      back(Arg, NewArg),
      back_list(Args, NewArgs).
   ```

   This has the effect that an uninstantiated variable is associated with each term (and, recursively, its subterms).

2. Having furnished all terms with an uninstantiated variable, we must now see to it that these variables are indeed instantiated as soon as a reduction step is performed. One way to do this is to adapt all reduction rules created in steps GF2 and GF3, to reflect the fact that if a rule is applied to a term (yielding its reduct), then the variable associated with that term must be instantiated to the reduct.

Since this procedure would render many clauses very hard to understand, we have chosen another approach and define a new predicate which does the instantiation mentioned above:

```
rset(red(S, V), T) :- var(V), !, V=T.
rset(_, _).
```

If the first argument of the predicate `rset` is a `red`-term with an uninstantiated variable `V`, that variable is instantiated to the term `T`. If `V` was already instantiated, then `T` is a reduct of `S` and already occurs somewhere in `V`. In that case nothing needs to be done, and `rset` will simply succeed.

Now, whenever a reduction step is performed, `rset` must be called with the proper arguments. Hence, after each occurrence of the operator `--->` in the body of a clause `rset` is called. For instance, the definition of the operator `--->>` contained the clause

```
X --->> Z :- X ---> Y, Y --->> Z.
```

and will now appear as

```
X --->> Z :- X ---> Y, rset(X, Y), Y --->> Z.
```

3. Next we are going to adapt step GF2 of the general framework. Obviously, if $\rightarrow_1$ is chosen to be `--->>`, graph reduction will not be very useful, since the common subterm is normalized beforehand anyway.

   Now, reduction rules must take into account that terms and their subterms are modified. This is done by enveloping every subterm in a rule with a `red` (except variables, which will simply match the new term, including uninstantiated variables). Note that this process boils down to performing the `trans` predicate on both sides of a rule.

   For instance, the rules

```
0 * X ---> 0.
s(X1) * X2 ---> (Y1 * Y2) + Y2 :- X1 →1 Y1, X2 →1 Y2.
```

   will now appear as

```
red(red(0, _)*X, _) ---> red(0, _).
red(red(s(X1), _)*X2, _) ---> red(red(Y1*Y2, _)+Y2, _) :-
   X1 →1 Y1, X2 →1 Y2.
```

4. The clauses in step GF3 must be adapted in the same manner. They will now appear as

```
red(f(X1, ..., Xn), _) ---> red(f(Y1, ...Yn), _) :-
   X1 = Y1, . . ., Xi →2 Yi, . . . , Xn →3 Yn.
```

5. Finally, a reduction rule must be added for the function `red` in order to reap the harvest of already performed reductions.

   The following clause is added in step GF2 and must appear as the first clause therein.

   ```
   red(X, S) ---> S :- nonvar(S), !. /* re-use */
   ```

   The cut is there because special care must be taken that terms in which the variable is already instantiated are not matched with the other rules in step GF2. The uninstantiated variable in those rules would obligingly match anything. This would not lead to errors, but it would result in the Prolog system doing all the work we tried to save by using graph reduction, as can be seen as follows. Suppose the term $\tau$ is a term in which reduction steps have already been performed (in an other context). Then $\tau$ looks like this:

   ```
   red(t0, red(t1, red(t2, ..., red(tn, _)...))
   ```

   where `t0` $\rightarrow$ `t1` $\rightarrow$ `t2` $\rightarrow \ldots \rightarrow$ `tn` is some reduction sequence. Reattempting to reduce $\tau$ would eventually make the Prolog system find the very same reduction sequence which is already present in $\tau$, because this reduction sequence is the only one that will match.

An interesting property of the method we have just presented is that the classification in Figure 2.7, Section 2.2.1.5, is still applicable, as can be seen as follows. In every clause in parts GF2 and GF3 (the clauses concerning basic reduction steps), apart from the clause marked `/* re-use */`, the second argument of `red` is always `_`. Consequently, those clauses induce the same reduction strategy whether such a second argument is instantiated or not. The clause

```
red(X, S) ---> S :- nonvar(S), !. /* re-use */
```

succeeds exactly when `S` is instantiated. This was done by `rset`, immediately after some goal `red(X, V)--->T` succeeded. Hence the goal `red(X, S) ---> T` will find the same result, whether `S` is instantiated or not. Since `red` did not occur in the TRS, no redex patterns are added or eliminated. As a consequence, the reduction strategies implemented by the graph reduction method discussed in this section and the general method discussed in Section 2.2.1.5. are identical.

If it is unimportant whether a standard strategy is followed, an alternative re-use clause may increase the efficiency:

```
red(X, S) ---> T :-
    nonvar(S), !, S -=-> T. /* re-use2 */
```

Now, whenever a reduction is required in a term for which reducts are already known, a reduction step is attempted in the final reduct available. This happens because `S -=-> T` first tries to satisfy `S ---> T`, which will again trivially succeed if previously found reducts are available. Eventually the chain of reducts is exhausted, in which case `-=->` succeeds, either trivially (normal form has been reached), or by performing a "real" reduction step.

A final remark about the implementation we have just presented is about its efficiency (in execution time!). Obviously, the above program introduces a large amount of extra work, because the depths of all terms and all redex patterns are doubled. Whether this work is repaid by graph reduction depends on the size of the TRS (the bigger the better) and of course on the amount of duplication of subterms. The implementation of the TRS shown in the beginning of this section (using clause `re-use2`) is already more efficient in execution time than the non-graph implementation (using the same strategy) for the term

```
s(s(0))*(s(s(0))*(s(s(0))*s(s(0))))
```

To conclude this section, we will now present the graph-implementation of the module `Successor-numbers`. Note that the operator `--->>` finds the normal form of a term in its external representation, `--->>>` does the same for terms in their internal (unbound variables inserted) notation. The program terminates for queries $\sigma$`--->>X`, where $\sigma$ is a ground term in the signature of the original TRS (without `red` terms).

```
:- op(500, yfx, ['*','+']).
:- op(700, xfx, ['--->','--->>','-=->','->->>','--->>>']).

X --->> Y :- trans(X, X1), X1 --->>> Y1, back(Y1, Y).

trans(U, red(V, _)) :-
   U =.. [Func|Args],
   trans_list(Args, Newargs),
   V =.. [Func|Newargs].

trans_list([], []).
trans_list([Arg|Args], [NewArg|NewArgs]) :-
   trans(Arg, NewArg),
   trans_list(Args, NewArgs).

back(red(V, _), U) :-
   V =.. [Func|Args],
   back_list(Args, Newargs),
   U =.. [Func|Newargs].

back_list([], []).
back_list([Arg|Args], [NewArg|NewArgs]) :-
   back(Arg, NewArg),
   back_list(Args, NewArgs).

rset(red(S, V), T) :- var(V), !, V=T.
rset(_, _).

red(X, S) ---> T :- nonvar(S), !, S -=-> T.

red(red(0, _)+X, _) ---> X.
```

```
red(red(s(X1), _)+X2, _) ---> red(s(red(Y1+Y2, _)), _) :-
   X1 $\rightarrow_1$ Y1, X2 $\rightarrow_1$ Y2.

red(red(0, _)*X, _) ---> red(0, _).
red(red(s(X1), _)*X2, _) ---> red(red(Y1*Y2, _)+Y2, _) :-
   X1 $\rightarrow_1$ Y1, X2 $\rightarrow_1$ Y2.

red(X1 + X2, _) ---> red(Y1 + Y2, _) :- X1 $\rightarrow_2$ Y1, X2 $\rightarrow_3$ Y2.
red(X1 + X2, _) ---> red(Y1 + Y2, _) :- X1 = Y1, X2 $\rightarrow_2$ Y2.
red(X1 * X2, _) ---> red(Y1 * Y2, _) :- X1 $\rightarrow_2$ Y1, X2 $\rightarrow_3$ Y2.
red(X1 * X2, _) ---> red(Y1 * Y2, _) :- X1 = Y1, X2 $\rightarrow_2$ Y2.
red(s(X), _) ---> red(s(Y), _) :- X $\rightarrow_2$ Y.

X --->>> Z :- X ---> Y, rset(X, Y), Y --->>> Z.
X --->>> X.

X -=-> Y :- X ---> Y, rset(X, Y).
X -=-> X.

X ->->> Z :- X ---> Y, rset(X, Y), Y --->>> Z.
```

Remember that $\rightarrow_2$, $\rightarrow_3$ and $\rightarrow_1$ have to be filled in with `--->`, `-=->`, `--->>>` and `->->>`.

The above program is the result of a program generation process which can easily be automated.

### Fine-tuning a strategy

In Section 2.1 we have seen a number of standard reduction strategies, which were independent of any TRS.

We have also seen that for some TRSs (the strongly sequential ones) an optimal reduction strategy exists, which may or may not be one of the standard strategies.

For a confluent and Noetherian TRS adaptations can often be made to some standard strategy (resulting in a non-standard strategy) which will increase the efficiency of its execution.

Consider, for instance, a TRS for Booleans (with functions `true`, `false`, `and`, `or` and `not`):

```
not(true) ---> false.
not(false) ---> true.
and(true, X) ---> Y :-
   X $\rightarrow_1$ Y.                          /*1*/
and(false, X) ---> false.
or(X1, X2) ---> not(and(not(Y1), not(Y2))) :-
   X1 $\rightarrow_1$ Y1,                        /*2*/
   X2 $\rightarrow_1$ Y2.                        /*3*/
```

In the last clause, the second argument place is not an index, because the `not(Y1)` may eventually yield `false`, in which case the `not(Y2)` is never used.

Hence the predicate $\rightarrow_1$ in the line marked `/*3*/` should be `=` independent of what it is in other rules.

If a term with `not` or `and` as outermost function symbol cannot be reduced, then the first argument is needed, and it must be normalized before a rule can be applied to the entire term (note that this is true even if there are Boolean normal forms other than `true` and `false`). This can be achieved by adapting the rules concerning `not` and `and` created in step GF3 as follows:

```
not(X) ---> not(Y) :- X1 ->->> Y1.
and(X1, X2) ---> and(Y1, Y2) :- X1 ->->> Y1, X2 = Y2.
and(X1, X2) ---> and(Y1, Y2) :- X1 = Y1, X2 ->->> Y2.
```

Now, the first argument will be normalized entirely as soon as an irreducible `not`- or `and`-term is found. If no step can be performed in the first argument of an `and` term, the second argument is normalized.

Observe that the `or`-clause above can always be applied. That is, any term with `or` as outermost function symbol is a redex. Consequently no context rules (step GF3) need to be supplied for `or`.

From the example above, we can see that fine-tuning a strategy is very much dependent on the TRS under consideration. To date, little literature exists about optimizing a strategy automatically. An alternative approach is to allow users to annotate a specification with suggestions about the strategy. This possibility is investigated in [12].

### The canonical predicate

In Section 2.2.1.2 we have seen that the reduction strategy described by van Emden and Yukawa uses the predicate `canonical`. Often, such a predicate can be evaluated very efficiently. If so, it may be advantageous to check any term before attempting to reduce it (which would be done by trying to apply every rule on every node of the term). The canonical predicate can be introduced in Wiedijk's implementation by redefining the predicate `--->>`.

```
X --->> X :- canonical(X), !.
X --->> Z :- X ---> Y, Y --->> Z.
```

Note that the correctness of this definition (and of the ensuing program) relies on the assumption that `canonical(`$\tau$`)` succeeds if and only if $\tau$ `---> _` fails. As a consequence, the cut may indeed be added as shown. When `canonical(X)` succeeds, the goal `X --->> _` cannot be satisfied in any other way.

An alternative approach, useful when the canonical predicate is relatively hard to specify, is the following. Whenever a normal form of some term is found, that normal form is added to a database. We can use the Prolog database for this purpose, and we use the same predicate `canonical` for storing normal forms, this time assuming that no clauses are present initially for this predicate.

```
X --->> X :- canonical(X), !.
X --->> Z :- X ---> Y, Y --->> Z.
X --->> X :- assert(canonical(X)).
```

Initially, when no clauses are present for `canonical`, the first clause above simply fails, and `--->>` is attempted in the normal fashion. Then, when `--->` eventually fails because its first argument is already in normal form, the third clause is attempted. This clause succeeds, without changing the normal form. As a side-effect the fact that the term was a normal form is stored. Next time an attempt is made to normalize that term, the first clause above will immediately succeed.

### 2.2.3 Conditional, many-sorted specifications

So far we have only considered unconditional specifications over a single sort. Yet most specification formalisms (including ASF) support specifications with conditional equations, involving more than one sort. In this section we will investigate these two extensions, showing implementation methods for each of them.

A third extension of our algebraic formalism would be to allow for modular specifications. Unfortunately every specification formalism introduces modularity in a different manner. For instance, the meaning of a modular ASF specification is defined in terms of a normalization process, which transforms the modular specification into an equivalent non-modular one. Here, in order to implement modular specifications, one only needs to implement the normalized non-modular specification using one of the implementation methods described in the previous sections, as we have seen in Section 2.2.1. We will not consider the implementation of modular specifications in general.

#### Conditional equations

A conditional equation is an equation with a number of conditions, and looks like

$$s = t \textbf{ when } sc_1 = tc_1, ..., sc_n = tc_n$$

A conditional equation is only applicable when the conditions are all met. That is, $s^\phi \rightarrow t^\phi$ is a basic reduction step if $s^\phi$ is an instantiation of $s$ using some substitution $\phi$, $t^\phi$ is the $\phi$-substitution of $t$ and for each $i$, $sc_i^\phi$ is equal to $tc_i^\phi$.

This description does not tell us what to do about variables occurring in some $sc_i$ or $tc_i$, but not in $s$. This occurs, for example, in the equation

$$x > z = true \textbf{ when } x > y = true,\ y > z = true.$$

The problem is that matching some ground term $s_0$ with the left-hand side of the equation $s = t$ only gives rise to a substitution of the variables occurring in the open term $s$. Variables occurring in the conditions but not in $s$

could be substituted by anything. Indeed, whereas variables occurring in $s$ are universally quantified, all others are existentially quantified, as can easily be verified:

$$\begin{array}{l}
\forall\ x, y[s(x) = t(x) \quad \textbf{when} \quad sc(x,y) \ = \ tc(x,y)\ ] \ \Leftrightarrow \\
\forall\ x, y[\ sc(x,y) \ = \ tc(x,y) \ \Rightarrow \ s(x) \ = \ t(x)\ ] \ \Leftrightarrow \\
\forall\ x, y\ [\ \neg(sc(x,y) \ = \ tc(x,y)\ ) \ \vee \ s(x) \ = \ t(x)] \ \Leftrightarrow \\
\forall\ x[\forall\ y[\neg(sc(x,y) = tc(x,y)\ )] \vee \ s(x) = t(x)] \ \Leftrightarrow \\
\forall\ x[\neg\exists\ y[sc(x,y) = tc(x,y)] \vee \ s(x) = t(x)] \ \Leftrightarrow \\
\forall\ x[\exists\ y[sc(x,y) = tc(x,y)] \Rightarrow \ s(x) = t(x)] \ \Leftrightarrow \\
\forall\ x[s(x) = t(x) \quad \textbf{when}\ \exists\ y[sc(x,y) = tc(x,y)]]
\end{array}$$

The problem of finding a substitution (for existentially quantified variables) which validates the conditions is, of course, undecidable in general.

At this point we should also look at the reason why new variables are not allowed in the right-hand side of a rewrite rule: we do not want arbitrary values to pop up out of nowhere. In conditional TRSs however, a "new" variable may well have been introduced in the conditions. As an example consider the following equation, taken from a static type checker in [3].

```
[143] check(abs-pico-program(dec, ser)) = and(b1, b2)
  when <b1, env1> = check(dec, null-type-env),
       b2         = check(ser,env1)
```

This equation states that a program is syntactically correct if the declaration section is correct (which, as a "side-effect", yields a type-environment) and if the statement section of the program is correct, using the established type-environment.

Note that the variables `b1` and `b2` occur in the right-hand side of the unconditional part of the equation, but not in the left-hand side. Nevertheless, the specification is a TRS if the `check` functions always return a proper Boolean/environment pair and a proper Boolean respectively.

In order to implement the harmless uses of "new" variables, we are going to impose some restrictions.

Recall that an open term is called stable if there exists no substitution (instantiating the variables in the term with ground terms of the proper sort) such that the resulting term contains a redex which overlaps with the original open term. In other words, an open term is stable if the normal form of any instantiation of that term is again an instantiation of that term. Note that this can be checked syntactically.

Now we will classify variables occurring in an equation. Consider the conditional equation

$$s = t\ \textbf{when}\ sc_1 = tc_1, ..., sc_n = tc_n$$

We will call a variable $v$ **explicit** if either

- $v$ occurs in $s$ or
- $v$ does not occur in $s$, but it does occur in the conditions, and in the first (i.e., leftmost) condition in which $v$ occurs (say $sc_k = tc_k$) $sc_k$ is stable and $v$ does not occur in $tc_k$.

We will call a specification explicit if all its variables are explicit. Henceforth we will only consider explicit specifications.

Now we will describe a procedure which, if we attempt to perform a basic reduction step $s^\phi \rightarrow t^\phi$, determines whether there exists a substitution $\phi^+$ under which all conditions are met, and which instantiates all variables in $s$ to the same value as did $\phi$.

Let $\phi_0$ be the given substitution which assigns values to each of the variables in $s$. Now, for each $i$ there are two mutually exclusive possibilities.

1. If $sc_i$ and $tc_i$ contain only variables already instantiated by $\phi_{i-1}$, then let $\phi_i = \phi_{i-1}$. Now we must check that the condition is met. This is done by normalizing $sc_i^{\phi_i}$ and $tc_i^{\phi_i}$ and checking for syntactical equality (we are still only considering confluent Noetherian conditional TRSs). If the two normal forms are not equal, no suitable substitution $\phi^+$ exists.
2. Otherwise the condition $sc_i = tc_i$ contains variables not occurring in $s$ or in any of the conditions $sc_k = tc_k$, $k < i$. Since every such variable is explicit, it does not occur in $tc_i$. Hence every variable in $tc_i$ is instantiated by $\phi_{i-1}$ and we can normalize $tc_i^{\phi_{i-1}}$, resulting in some term $t_i$. Then we have to normalize all values of variables occurring in $sc_i$ which are instantiated by $\phi_{i-1}$. Remember, $sc_i$ is a stable open term, but the values given to some of its variables by $\phi_{i-1}$ are not necessarily in normal form. Let $s_i$ be the instantiation of $sc_i$ with the normal forms of their $\phi_{i-1}$-value substituted for variables instantiated by $\phi_{i-1}$. The condition $sc_i = tc_i$ is now satisfied when $t_i$ is an instantiation of $s_i$. If so, let $\phi_i$ be the extension of $\phi_{i-1}$ which instantiates $s_i$ to $t_i$. Otherwise no suitable $\phi^+$ can be found.

$\phi_n$ is the substitution $\phi^+$ we are looking for.

Observe that $\phi^+$ is identical to $\phi_0$ for the variables in $s$ and that all conditions are met using this substitution.

We conclude this section with the Prolog implementation of conditional specifications.

Again we consider the rule

$$s = t \textbf{ when } sc_1 = tc_1, ..., sc_n = tc_n$$

and define for each condition the Prolog fragment $\gamma_k$.

- If $sc_k = tc_k$ is a "normal" condition, containing only variables which occurred in $s$ or in one of the previous conditions (with index less than $k$), then let $\gamma_k$ be the Prolog text

```
σk --->> Sk, τk --->> Tk, !, Sk = Tk
```

where $\sigma_k$ and $\tau_k$ are the terms $sc_k$ and $tc_k$ written in Prolog syntax.

- Otherwise, $sc_k$ is a stable open term containing some "new" variables (say $X_1^k$ to $X_n^k$) and some variables which occur in $s$ or in one of the previous conditions (say $Y_1^k$ to $Y_m^k$). In this case, let $\gamma_k$ be the partial Prolog body

```
τk --->> Tk,
Yk1 --->> NYk1,
 ...
Ykm --->> NYkm,
!,
σk[NYki|Yik] = Tk.
```

where $\sigma$`k[NYki|`$Y_i^k$`]` is the textual substitution of each $Y_i^k$ with the corresponding `NYki`.

As usual, cuts are provided to prevent Prolog from retrying to find normal forms.

Now, each rule

$$s = t \ \textbf{when} \ sc_1 = tc_1, ..., sc_n = tc_n$$

is translated to the Prolog clause

```
σ ---> τ :-
    γ1,
    .
    .
    γn.
```

where $\sigma$ and $\tau$ are the Prolog translations of $s$ and $t$ respectively.

### Many-sorted specifications

The major problem with the implementation of many-sorted specifications in a single-sorted implementation language (such as Prolog) is that a rule cannot always be applied to a term, even if the term matches the head of the rule. Associated with the head of the rule, and all its subterms including the variables, are their sorts. When a term is matched with the head of a rule all the corresponding sorts must be equal.

This problem is caused by the overloading of function symbols, as can be seen as follows. If every function symbol is unique, then matching two terms, which involves checking whether all corresponding function symbols are equal, implicitly checks whether the corresponding sorts are equal. Typechecking of variables should still be done. With each variable, a set of allowed function symbols is associated.

Overloading the function symbols may change this situation, as the following example shows. Consider the non-negative integers with `0`, `s` (successor) and $+$, with among others the rule

```
[1]    s(x) + y = s(x + y)
```

and assume the existence of a second sort, say strings, with a dyadic $+$ operator indicating concatenation and with a function `s` from strings to integers which counts the number of non-blank characters in a string.

Now consider the term

```
s("a perfect day for bananafish") + s(s(s(s(s(0)))))
```

Without proper typechecking this term can unexpectedly be rewritten by rule `[1]` to the term

```
s("a perfect day for bananafish" + s(s(s(s(0)))))
```

which is meaningless.

In order to solve this problem, we have to implement some kind of typechecking. Unfortunately for our purposes, standard Prolog does not support user-defined data types. To implement it ourselves would mean to add its sort to every node in a term and to every node in the Prolog program. This would decrease the efficiency of the Prolog implementation, but that is to be expected for added functionality. The ASF compiler described in Section 2.2.1.6 automatically adds type information to the generated code.

From the observations above we may conclude that the actual amount of typechecking necessary depends on the amount of overloading allowed by the specification formalism. For instance, OBJ2 [15] allows overloading of functions to a fairly large degree. This is combined with a hierarchical type system, which includes automatic type-conversions (such as widening) in some cases. ASF is slightly less complicated. There is no hierarchical sort-structure. Overloading of functions is allowed as long as the sort of a function can be determined from its name, and the sorts of its arguments. In other words, the sort of a term can be determined by finding the sorts of its innermost subterms, and progressing outwards.

We conclude this section with the description of an implementation technique for a limited form of overloading. It is applicable to ASF specifications with the property given below. If an ASF specification does not have this property, the implementation technique can still be used, provided that some functions are renamed. **Property:** If the specification contains more than one definition for a given function symbol, then the definitions with the same output sort have different numbers of arguments (only the number of arguments is considered here, not the sorts of the arguments).

As a consequence, given the sort $S$ of some term $t$, the sorts of all its subterms can be determined as follows: consider the outermost function symbol, say $f$. The output sort of this function is $S$, and the number of arguments for this function $f$ can be seen in $t$ (say $t$ has $n$ immediate subterms). Due to the property above, the specification contains exactly one function with function symbol $f$, output sort $S$ and $n$ arguments. We can simply look at the declaration of this function to determine the actual sorts of its arguments. This process can now be repeated for the immediate subtrees.

The example with strings and integers given earlier does not have the above property. A binary + yielding integer and a binary + yielding strings are allowed. A unary $s$ yielding an integer, but taking either a string or an integer as argument, is not allowed.

To implement this limited form of overloading, only one sort needs to be kept with every rewrite rule: the sort of the terms it is applicable to. When a term of that sort matches the head of the rule (which implies that the outermost functions symbols are equal, and have the same arity), then all types of the corresponding subterms must be equal.

It would be very nice to be able to index the `--->` operator with a type (as in $-int \rightarrow$), but this is not a feature supported by Prolog. An alternative would be to give the `--->` operator an extra argument, the sort it may be applied to. Unfortunately this would mess up our attractive infix notation for `--->`, because there are no ternary operators.

We will use yet another approach: introduce a prefix operator for every sort, and prefix every rewrite rule with its sort. For example, the fact

```
s(X) + Y ---> s(X + Y).
```

will now appear as

```
nat s(X) + Y ---> s(X + Y).
```

The priority of the `--->` operator is chosen to be higher than that of the sort operators, so that the implied parentheses are:

```
nat(--->(+(s(X), Y), s(+(X, Y)))).
```

In summary:

- For each sort `sort` the Prolog program contains the operator declaration

  ```
  :- op(650, fx, sort).
  ```

- For every rewrite rule $s = t$ of sort `sort` the program must contain the clause

  `sort` $\sigma$ `--->` $\tau$.

- For every function symbol $f$ with domain `sort1 # sort2 # ... # sortn` and range `sort` the program should contain the $n$ clauses

  ```
  sort f(X1, X2, ..., Xn) ---> f(Y1, Y2, ..., Yn) :-
     sort1 X1 ---> Y1, X2 = Y2, ..., Xn = Yn.
   ...
  sort f(X1, ..., Xi, ..., Xn) --->
                    f(Y1, ..., Yi, ..., Yn) :-
  ```

```
   X1 = Y1, ..., sorti Xi ---> Yi, ..., Xn = Yn.
 ...
sort f(X1, ..., Xn-1, Xn) --->
                   f(Y1, ..., Yn-1, Yn) :-
   X1 = Y1, ..., Xn-1 = Yn-1, sortn Xn ---> Yn.
```

- The definition of `--->>` should be replaced by a new definition for each sort `sort`

  ```
  sort X --->> Z :- sort X ---> Y, sort Y --->> Z.
  sort X --->> X.
  ```

- The program terminates for the query

  ```
  ?- sort τ --->> V.
  ```

  if $\tau$ is a ground term of sort `sort`. `V` is instantiated to the normal form of $\tau$.

# Chapter 3

# Tree matching

Efficient tree matching is essential for the implementation of term rewriting systems. We present a method for tree pattern recognition based on string scanning techniques. This method poses no constraints on the set of patterns and its efficiency does not depend directly on the size of the set of patterns.

## 3.1 Introduction

In most symbolic computations involving tree structured data the key operation is tree matching. Given a set of tree patterns and a subject tree, an instance of any of the patterns is searched for, in the subject tree.

Some areas where tree matching occurs are: interpreters for symbolic languages (equational languages, functional languages and logic programming languages), automatic deductive systems (expert systems, theorem provers), structure editors and code optimizing compilers.

Our area of interest is that of equational languages. In the ESPRIT project GIPE (Generation of Interactive Programming Environments) the Algebraic Specification Formalism ASF is used to define semantics. ASF is a modular, conditional, equational language. In the implementation, ASF specifications are interpreted as term rewriting systems. In a term rewriting implementation the major part of execution time – typically more than 80 % – is spent in determining, given a subject term and a set of rewrite rules, the left-hand side of which rule, if any, the term is an instance of. This operation is precisely the tree matching problem described above. In Section 3.5, an implementation of term rewriting systems is sketched.

For a long time already, string searching methods have been applied to tree matching and many efficiency increasing techniques (such as [30], [7], [1], [31]) have been applied successfully to tree matching ([37], [22], [31], [36]). Our current problem domain is similar to that in [22] and [36]; our solution is similar to the one used in the Equation Interpreter (EQI) [36]. Because of this similarity, a detailed comparison of the two methods is presented in Section 3.5.

An assumption on which most of these efficiency increasing techniques are based, is that a single set of tree patterns is used on many different subject trees. Under this assumption efficiency can be gained by preprocessing the set of patterns.

In the context of our research this assumption is not valid. The ASF+SDF system [28] is an integrated programming environment, based on a syntax driven text/structure editor and user definable syntax and semantics (based on term rewriting). The emphasis in this system is on the incremental maintenance of syntax and rewrite rules. Techniques which preprocess the set of patterns usually behave badly under incremental changes to the pattern set. Furthermore, in a repeated edit/test cycle often only a part of the entire pattern set is actually used (also consider libraries). In this case the time spent in preprocessing unused parts of the pattern set, each cycle, often outweighs the gain in run-time efficiency. As a consequence, we are interested in efficient solutions which do not assume *a priori* knowledge of the set of rewrite rules.

For the ASF+SDF system a scanner generator was developed which generates lazy, incremental text scanners for arbitrary modular regular languages [19]. This generator features several efficiency increasing techniques we are able to exploit. We will present a conservative extension (i.e., which still serves the original purpose) of that generator which allows us to create tree scanning automata.

As mentioned, the method of adapting string scanning to tree matching is basically identical to that in [36]. In that chapter however, constraints are imposed on the pattern set such that partially overlapping tree patterns can be disambiguated in a pre-order traversal of function symbols. With this constraint the existence of variables can largely be ignored. In Section 3.2 we will show how this is done, and we will show why the presence of variables is problematic. Note that there are more reasons behind these constraints than the relative simplicity of the scanners. This is discussed in Section 3.5.2.

In the context of ASF these constraints are too rigid. ASF allows arbitrary term rewriting systems, and the occurrence of variables cannot be avoided. Our method imposes only a single constraint on the pattern set: it may not contain a pattern consisting of a sole variable. In most situations such patterns are not useful, because they match any term. In the case of term rewriting systems, single variables are even forbidden [32].

We will not consider the problem of overloading. It is assumed that identical function symbols representing different functions have been disambiguated in a type-checking phase. All variable names are assumed to be distinct. Non-linear patterns (i.e., patterns containing the same variable more than once, where every occurrence should match an identical subterm) are handled by checking for equality afterwards. That is, application of an automaton to a term yields the set of matching patterns, together with the corresponding substitutions. In the case of non-linear patterns that substitution must be checked for validity, by comparing the values found for all occurrences of the same variable (all must be equal). The reason why this validation cannot be done on the fly, as it is done in straightforward matching, is addressed in Section 3.4.

Throughout this chapter we will use the following signature in our examples concerning terms: $z$ is a constant, $s$ is a monadic function symbol, $p$ is a dyadic function symbol and $\alpha$, $\beta$, etc. are variables. This signature can be seen as a signature of the natural numbers, where $z$ is 0, $s$ maps every number to its successor and $p$ indicates addition.

In this chapter we will use 'tree' and 'term' interchangeably.

## 3.2 Tree matching automata ⋈

The key idea when applying string scanning techniques to tree matching is to flatten all trees or, in other words, to forget the tree structure and only consider the linear string of function symbols in the tree. The most common way to do this is to look at the function symbols as they are encountered in a pre-order traversal of the tree. This is the same order as they appear in a normal textual representation. For instance the term $p(z, s(z))$ would appear as $pzsz$.

The resulting language of regular expressions is straightforward in this case. Apart from symbols only a single operator is used: concatenation.

For pattern sets without variables this method suffices to construct a tree matcher, as the following example shows.

Consider the pattern set $\{z, s(z), p(z, z), p(s(z), z), p(z, s(z))\}$. This set is recognized by the automaton in Figure 3.1.

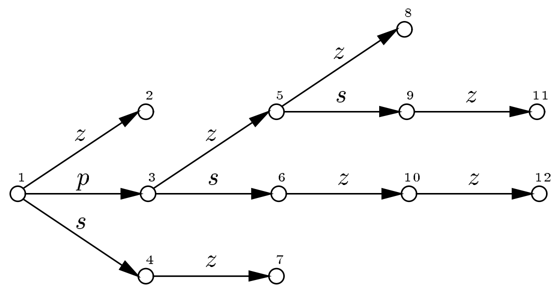


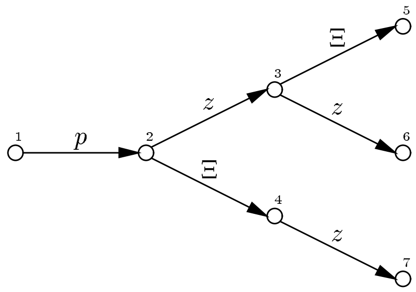

*Figure 3.1.*

For example, from the subject tree $p(s(z), z)$ the string $pszz$ is derived, which is accepted by the automaton after visiting the states 1, 3, 6, 10 and 12, the last state of which is an accepting state where all input has been read. From the subject tree $p(z, s(s(z)))$ the string $pzssz$ is derived, which is not accepted by the automaton, because after visiting states 1, 3, 5 and 9 no transition labeled $s$ is possible.

Observe that if no identical patterns occur in the set, every final state is associated with a unique pattern.

When the pattern set contains open terms (terms containing variables), a problem arizes, as can be seen in the following example.

Consider the pattern set $\{p(z,\alpha), p(\beta,z)\}$. An automaton accepting instances of this set is sketched in Figure 3.2.




The intended meaning of edges marked $\Xi$, is that transitions along such edges can only be taken if no other transition is possible. Consequently the automaton is deterministic.

*Figure 3.2.*

As we can see, edges corresponding to variables are marked with the symbol $\Xi$. Informally this symbol could be viewed as a new regular operator. Formally some care should be taken, because the language corresponding to $\Xi$ is not even regular (the language generating arbitrary terms is context free) [23]. Consequently every edge marked $\Xi$ should be substituted by a push down automaton or a parser, and information about the signature would be needed (this can be seen by looking at the flattened term $pszsz$; to get at the second subterm, we need to know that $s$ is a monadic function symbol, and that $z$ is a constant).

We will refrain from giving a formal definition of $\Xi$. An operational characterization will be developed in this chapter. The reason is that we should wonder if all this additional information and 'machinery' is really needed.

Consider the tree in Figure 3.3. In order to recognize it as an instance of $p(\beta,z)$ we do not have to look at a single symbol of the subterm $s(s(s(z)))$. After visiting the $p$-node we immediately move to its second child; the first child is entirely skipped. Looking at the string $psssszz$ however, it is not clear how this could be done. The reason is, that information (about the structure) has been lost in flattening the trees.

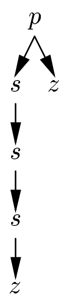


*Figure 3.3.*

We would like an automaton to be able to skip parts of the tree being scanned. The term is not flattened beforehand, but the traversal is done while the automaton is scanning, and the traversal process can be directed by the automaton.

In our automata all edges will be labeled with the tree move to be performed when the transition corresponding to that edge is taken. Normally this tree move coincides with pre-order traversal. When a subterm is not needed, because it is matched entirely by a variable, it is skipped.

These tree moves are 'coded' into the automaton by the generator, at a time when the tree structure is still available.

When a scan has been successful, we are not only interested in which patterns have been recognized. We are also interested in the subterms corresponding to the variables in those patterns. For instance, if the automaton reports that some term is an instance of $p(\beta, z)$, we would like to know for which value of $\beta$ this is the case. This information is available when the automaton is scanning the subject tree.

Consider for example state 2 in the automaton shown in Figure 3.2. In this state the outermost function symbol corresponding to the variable $\beta$ will be read. When the automaton enters this state, while scanning some term, the 'current' subterm (pointed to by the traversal process) is precisely the value which should be associated with $\beta$.

In our automata we will annotate all states with the list of variables which match the current subterm at that point. This will be made precise in Section 3.4.

We will now present the automaton for a bigger pattern set.

The pattern set is the set of left-hand sides of the following rewrite rules defining addition on natural numbers.

1. $p(z, \alpha) \rightarrow \alpha$
2. $p(\beta, z) \rightarrow \beta$

3. $p(s(\gamma), \delta) \rightarrow s(p(\gamma, \delta))$
4. $p(\varepsilon, s(\zeta)) \rightarrow s(p(\varepsilon, \zeta))$

In Figure 3.4 tree moves are represented by expressions such as $u$ k $d$ n. This means: move $k$ levels up in the tree, and then move to the $n$-th child.

Furthermore, to every final (i.e., accepting) state the patterns which have been recognized are attached in the form of the corresponding rule number. For instance, in the state labeled $1, 4$, an instance of the term $p(z, s(x))$ has been recognized, which matches both patterns one and four.

Observe that given some term, all patterns which that term is an instance of are found in a single scan. Also observe that the time needed for a single scan does not depend directly on the size of the pattern set (it depends on the degree of overlapping in the pattern set).

In Figure 3.4 states have been annotated with the variables which occur in that state for the first time. This means that the state has the same tree location as that variable.

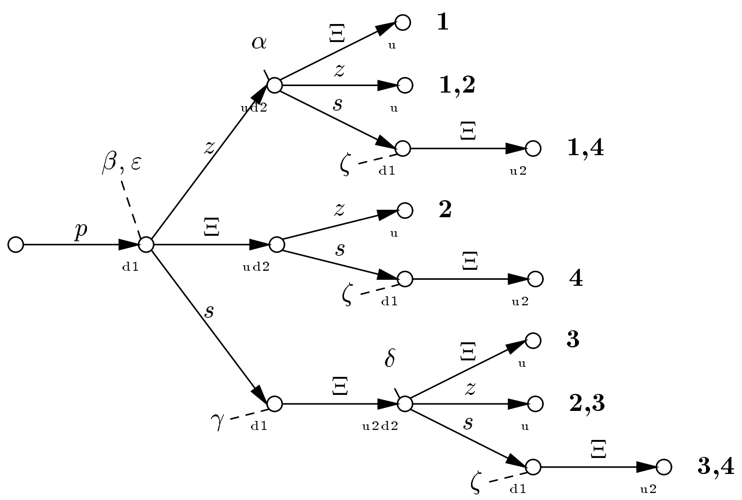



*Figure 3.4.*

## 3.3 String scanning

In this section we will look at the generation of text scanning automata. We will explain the entire process by way of a step by step example. The method we will use is originally described in [34]. Our notation and example are taken from [19].

Consider the set of regular expressions $\{a(b|c)*, b*d, abc, bcd\}$, where $*$ means zero or more repetitions, $|$ indicates alternatives, juxtaposition means concatenation, and parentheses are used for grouping. For example, some sentences defined by this set are *abbcb*, *a*, *bbbd* and *abc*.

The first step in creating an automaton is to append a termination symbol (we will use \$) to each expression. Next, all characters (including \$) are tagged with a unique position number. These tagged symbols will be called **positions**. This process yields the labeled expression set $\{a_0(b_1|c_2) * \$_3, b_4 * d_5\$_6, a_7b_8c_9\$_{10}, b_{11}c_{12}d_{13}\$_{14}\}$.

On labeled expressions the functions *nullable*, *firstpos*, *lastpos* and *pos* are defined. *Nullable* yields *true* when an expression can generate the empty string; *firstpos* yields the set of positions which can appear at the beginning of any string generated by an expression; *lastpos* similarly; and *pos* yields the set of positions in an expression. These functions are trivially extended to sets of patterns.

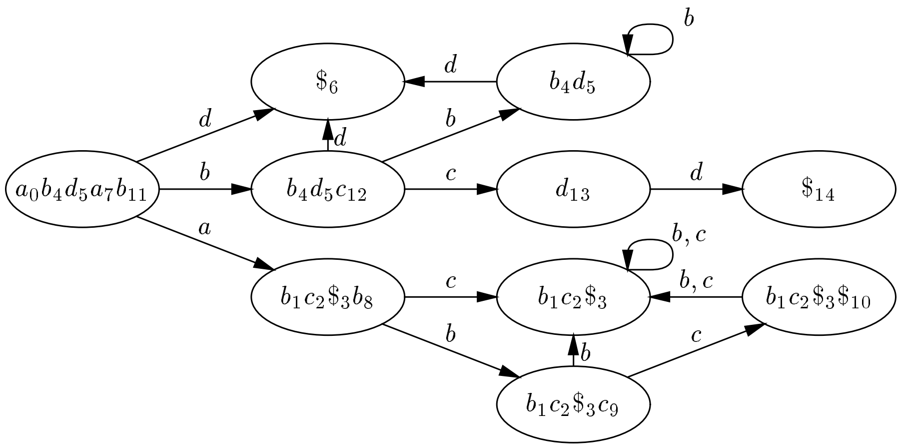
Examples:

- $firstpos(a_0(b_1|c_2) * \$_3) = \{a_0\}$;
- *firstpos* on the entire pattern set is $\{a_0, b_4, d_5, a_7, b_{11}\}$;
- $nullable(a_0(b_1|c_2)*) = false$;
- $nullable((b_1|c_2)*) = true$.

Using these auxiliary functions the function *followpos* can be defined, which, given a position and a labeled expression (set), yields the set of positions which may appear immediately after the given position. In this chapter we will assume the pattern set to be an implicit parameter and accordingly it will not be shown. For example, given the pattern set above, $followpos(b_1) = \{b_1, c_2, \$_3\}$.

The automaton is created by computing all states and transitions. A state is uniquely defined by the set of positions which can be accepted when the automaton is in that state. We will identify a state with its set of positions. Thus, the start state is the *firstpos* of the pattern set. In our example the start state is $\{a_0, b_4, d_5, a_7, b_{11}\}$. From a state, an outgoing edge exists for every symbol which occurs at some position in that set, except \$ symbols of course. For instance, the start state has outgoing edges labeled $a$, $b$ and $d$. The state at the end of an edge is defined by the set of positions which may occur there. This set is determined by taking the *followpos* of all positions in the originating state which have the same symbol as the edge under consideration. For instance, at the end of the $b$ edge from the start state we have the *followpos* of $\{b_4, b_{11}\}$, which is the set $\{b_4, d_5, c_{12}\}$. When these new states have been determined their outgoing edges can be computed and so on. The process ends when all states and edges are known.

In Figure 3.5 the entire automaton is shown.




States with a \$ symbol are accepting states.

*Figure 3.5.*

## 3.4 Generation of tree matching automata

We will now adapt the process described above for tree matching. We will take the example used earlier, the pattern set $\{p(z,\alpha), p(\beta,z), p(s(\gamma),\delta), p(\varepsilon, s(\zeta))\}$.

We will start by flattening the tree patterns into strings, and adding terminal symbols. As mentioned earlier, the only regular operator used is concatenation. All other operators (such as $|$ and $*$) do not occur, and will not be used in the term case. Since the tree structure should not be lost, all symbols (including variables and terminal symbols) are tagged with their location within the tree. Terminal symbols are tagged with the root location. This will be explained later. A **location** is a list of child-numbers. For instance, [2,3,1] is reached by starting in the root, and going down in its second child. In that node the third child is entered, and finally the first child is entered. In other words, [2,3,1] indicates the first child of the third child of the second child of the root. The notation $[n \parallel l]$ means: a list of which the first element is $n$, and the list of remaining elements is $l$. We will only show tree locations where they are needed, although every position is assumed to have a tree location.

$p_0^{[\,]} z_1^{[1]} \alpha_2^{[2]} \$_3^{[\,]}$
$p_4^{[\,]} \beta_5^{[1]} z_6^{[2]} \$_7^{[\,]}$
$p_8^{[\,]} s_9^{[1]} \gamma_{10}^{[1,1]} \delta_{11}^{[2]} \$_{12}^{[\,]}$
$p_{13}^{[\,]} \varepsilon_{14}^{[1]} s_{15}^{[2]} \zeta_{16}^{[2,1]} \$_{17}^{[\,]}$

The functions *nullable*, *firstpos*, *lastpos* and *pos* are defined the same as in the text scanning situation.

The function *followpos*, for variables, cannot be computed directly from these functions. In the following example we will see why.

Consider an automaton while it is scanning some term, from the moment it leaves the first state in which a variable occurs. As long as the automaton continues scanning within the same subterm, the variable remains 'active' and hence its *followpos* is itself. When the first symbol after that subterm is expected however, the variable's *followpos* should be the position immediately after that variable.

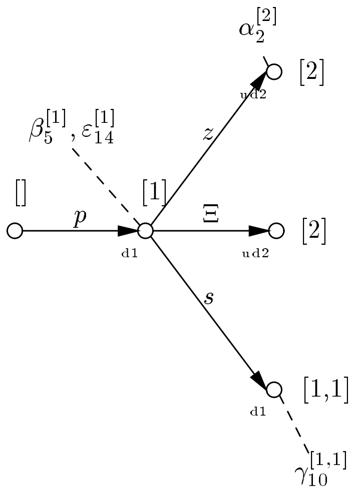



*Figure 3.6.*

For instance, the variable position $\beta_5$ in our example appears in a state together with positions $z_1$, $s_9$ and $\varepsilon_{14}$. Outgoing edges from that state will be labeled $z$, $s$, and $\Xi$. When, during scanning, the $z$ transition is taken, the entire subterm corresponding with the variable $\beta_5$ has been scanned. Consequently $followpos(\beta_5) = z_6$. If the $s$ transition is taken however, the next symbol will still be within the subterm corresponding to $\beta_5$. Hence $followpos(\beta_5) = \beta_5$. Clearly *followpos* depends on the state we are in and the edge under consideration. On variable positions, *followpos* has three arguments (apart from the implicit pattern set): a position, a symbol and a state. The result is one of two alternatives: either it is the variable itself, or its immediate successor. In order to determine which of the two alternatives should be taken, we should compare the location of the variable with the next tree location. But what is the next location?

With every state we can associate the location of the function symbol which will be read when the automaton is in that state. This does not help us very much though, because in the situation described above, the next state is not yet available. The function *followpos* is needed precisely to compute the next state!

Fortunately, the location of the next state can already be determined. Consider the example. In order to determine $followpos(\beta_5, z, \{\beta_5, z_1, s_9, \varepsilon_{14}\})$, we should note that the $z$ edge is being considered, and that this edges owes its existence to position $z_1$. The *followpos* of this position is well defined. It is $\{\alpha_2^{[2]}\}$. From this we may conclude that the location of the state at the end of the $z$ edge will be [2]. Since going from location [1] (the location of $\beta_5$) to [2] skips the entire subterm, *followpos* yields the second alternative, i.e., the immediate successor. Hence $followpos(\beta_5, z, \{\beta_5, z_1, s_9, \varepsilon_{14}\}) = z_6$. Likewise, $followpos(\beta_5, s, \{\beta_5, z_1, s_9, \varepsilon_{14}\}) = \beta_5$ because $followpos(s_9) = \{\gamma_{10}^{[1,1]}\}$ and $[1,1]$ occurs below [1].

The ordering on locations used here, is formally defined by the relation 'occurs at or below' ($\triangle$):

- $l \triangle [\,]$ for any location $l$;
- $[i \parallel l] \triangle [i' \parallel l'] \Leftrightarrow i = i' \wedge l \triangle l'$ for all numbers $i$ and $i'$, and locations $l$ and $l'$.

That is, $l \triangle l'$ precisely if $l'$, regarded as a string of integers, is a prefix of $l$.

With this definition we can finally formulate the definition of *followpos* for variables:

$$followpos(v_n^l, c, S) = \begin{cases} v_n^l & \text{when } c = \Xi; \\ v_n^l & \text{otherwise,} \\ & \text{if } c_m \in S \text{ is a position with symbol } c, \\ & \text{and } followpos(c_m) = \{p_{m'}^{l'}\}, \\ & \text{when } l' \triangle l; \\ p'_{n+1} & \text{otherwise,} \\ & \text{where } p'_{n+1} \text{ is the position immediately} \\ & \text{following } v_n, \text{ in the regular expression.} \end{cases}$$

This definition is well founded because:

- a $c_m$ in $S$ can always be found. Otherwise the edge labeled $c$ would not exist;
- $followpos(c_m)$ has exactly one member because in tree patterns no regular expression operators such as $*$ or $|$ exist. Hence, $c_m$ is followed by precisely one position. If this position is a terminating symbol ($), then the root location is returned. This location only 'occurs at or below' the root itself, but the start state is not allowed to contain variables.

In the introduction we mentioned that there is a technical reason why nonlinearity of patterns poses a problem. We can now investigate this problem in detail.

Consider a state in which one or more variables are 'active', which already occurred earlier in the pattern to which they belong. For the moment we will call these variables **secondary** variables. When the automaton makes a transition from this state, each of the secondary variables has to be checked to see if the same function symbol was present at the current location, the first time that variable occurred. This is the case both for transitions labeled $\Xi$ and for normal transitions. All secondary variables which do not satisfy this condition should not be present in the next state. This means that the transition ends in different states, depending on whether the condition is met or not. But this is the case for each secondary variable independently, as the following example shows.

Consider an automaton generated for the pattern set

$$\begin{array}{lll} \{ & \langle 1\rangle & f(f(\alpha,\beta),\alpha), \\ & \langle 2\rangle & f(f(\gamma,\delta),\delta), \\ & \langle 3\rangle & f(f(\varepsilon,\zeta),\xi) \ \ \} \end{array}$$

The first part of this automaton looks like

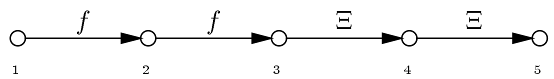



*Figure 3.7.*

We will now look at the outgoing edges of state 5. For instance, suppose the automaton scans the term $f(f(a,a),b)$. In state 5 the automaton reads the symbol $b$, which means that the variables $\alpha$ and $\delta$ do not match. Hence a transition exists to, say, state 6, in which only pattern 3 can be accepted. Now suppose the automaton scans the term $f(f(a,b),a)$. In state 5 the symbol $a$ is read. This time variable $\alpha$ does match its previous use, whereas $\delta$ still does not. Consequently a transition exists to a state distinct from state 6, say state 7, in which patterns 1 and 3 are applicable. Similarly, the terms $f(f(a,b),b)$ and $f(f(a,a),a)$ lead to states 8 and 9, in which, respectively, patterns 1 and 2, and 1, 2 and 3 are applicable. This reasoning is applicable to both kinds of transitions (labeled $\Xi$ and labeled with a function symbol).

In general, a state which would ordinarily have $n$ outgoing edges, but which has $k$ secondary variables, may eventually need $n2^k$ outgoing edges, leading to so many distinct states. This is clearly undesirable.

For this reason the automaton does not check for non-linear variables. Instead, the automaton keeps the value belonging to every occurrence of a variable separately. After a pattern has been recognized by the automaton, the corresponding substitution should be validated by comparing the values for all occurrences of the same variable.

The penalty in execution time for this approach is small. At first glance it seems that work is being done to recognize a pattern which will be disqualified afterwards. However, when other, applicable patterns are still being considered the work has to be done anyway (remember that all patterns the term is an instance of, are found at once). There is of course a certain amount of extra bookkeeping for the useless non-linear pattern, but this is negligible. So the largest overhead only occurs when all patterns being considered are non-linear, and will all be disqualified afterwards. This is of course a rare situation.

We have now established a correct definition of *followpos* in the case of term patterns with variables. Using this definition, the construction of the actual automaton is entirely the same as in the text scanning case.

We will show how the various annotations of the automata are computed.

- The tree locations of all constants in the same state are equal. This may not be immediately clear, but can be seen as follows. A state is reached by following some path through the automaton. The labels of the edges along that path define the pre-order traversal of all patterns up to the positions in that state. Since overloaded function symbols are not allowed, this pre-order traversal uniquely defines the location of the next symbol. Consequently all locations are equal.

  Using this property, we can define the tree location of a state. It is the location of one of the constant positions in that state, or, if there are only variables, it is the location of the 'deepest' variable. That is, the location of a variable which occurs at or below all others. An example of this situation occurs in the pattern set $\{f(\alpha), f(g(\beta, a)), f(g(\gamma, b))\}$. After reading the $f$ and $g$ symbols, the automaton is in a state which is associated with the variable positions $\alpha^{[1]}$, $\beta^{[1,1]}$ and $\gamma^{[1,1]}$. This state has location $[1, 1]$.

- The tree moves which are kept with every edge are computed by comparing the tree locations of states on either side of that edge. Tree moves always look like $u$ k  $d$ n, which means $k$ (possibly $k = 0$) levels up and then down into the $n$-th child.

  There is never need for more than one downward move. Normally the tree move coincides with pre-order traversal, and each time a single downward move is made. The only exception is when an entire sub-term is being skipped, possibly from deep within it. This means zero or more upward moves followed by again a single downward move.

  For instance, in our example the state arrived in after scanning $ps$ has location $[1, 1]$; the next state has location $[2]$. Consequently the computed tree move is $u$ 2 $d$ 2, as was shown in Figure 3.4.

- The last annotation to be discussed is the set of patterns appearing at each accepting state. As a matter of fact, such a set is maintained with every state. As mentioned earlier, the pattern set is an implicit argument to functions such as *followpos*, and consequently it must be available throughout the construction of the automaton.

The determination of this set for a given state is straightforward. The pattern set of the start state is the entire set and if some state $S_2$ is reached from $S_1$ traversing an edge labeled $c$, where the pattern set of $S_1$ is $P_1$, then the pattern set of $S_2$ is the subset of $P_1$ containing all patterns which have at the 'current place' either the symbol $c$ (if $c$ is a constant), or a variable (whatever $c$ is). The current place is the position with the same location as $S_2$, if such a constant or variable exists, or it is the deepest variable such that $S_2$ is located below it.

## 3.5 Term rewriting and the Equation Interpreter

In this section we will present a brief introduction to term rewriting implementations, and we will show how the method presented in this chapter can be used. Then we will present the Equation Interpreter (EQI) and the tree pattern matching method used therein. Finally, we will discuss similarities in, and differences between the two methods.

### 3.5.1 Term rewriting $\bowtie$

Formally, a term rewriting system is a set of rewrite rules such as $s_0 \to t_0$, where $s_0$ and $t_0$ are open terms (tree patterns) over some fixed signature. Two constraints are, that $s_0$ cannot be a single variable, and that every variable in $t_0$ also occurs in $s_0$. These rewrite rules define the rewrite relation, also denoted by $\to$. We say that $s$ can be rewritten to $t$ ($s \to t$) if $t$ can be derived from $s$ by replacing a subterm $r_s$ (called the **redex**), in $s$, with a term $r_t$, such that there exists a rule $s_0 \to t_0$ for which the following holds.

- $s_0$ matches $r_s$. That is, $r_s$ is an instance of $s_0$;
- $r_t$ is obtained from $t_0$ by replacing all variables with their value, as defined in matching $r_s$ against $s_0$. This makes sense because all variables in $t_0$ also occur in $s_0$.

For instance, suppose the rule

$$if(true, \alpha, \beta) \to \alpha$$

exists. Then the term

$$add(a, if(true, list(b), list(c)))$$

can be rewritten to

$$add(a, list(b)).$$

A term rewriting implementation gets a term (represented by a tree datastructure) and continuously attempts to rewrite that term until it can no longer be rewritten. The final result – if any, – is called the **normal form** of the original term.

For this purpose the given term must be traversed, and eventually all its subterms must be matched against the left-hand sides of all rewrite rules. The order of traversal determines which redex – if more than one exists – will be considered first. Although it is not essential to the tree scanning method, we will consider bottom-up traversal. That is, given some term first its immediate subterms are processed entirely; only then is the root level investigated.

For convenience we will assume all patterns to be linear. That is, a variable occurs at most once, in the left-hand side of any rule.

When a match is established, that subterm may be replaced by the instantiation of the right-hand side of the rule under consideration, using the values of variables established during matching.

We will now sketch a straightforward implementation in an algorithmic notation. Then we will show how the term scanning technique can be applied. We will not show how the automata are actually generated (this was already described in sections 3.2 and 3.4).

Our term rewriting implementation is based on the functions *normalize*, *reduce*, *match* and *instantiate*. The pattern set exists as a global variable *rules*. The central function is *reduce*. It gets a term, and returns the normal form of that term. Since a bottom up traversal is used, *reduce* can assume that the immediate subterms of its first argument are already in normal form (they do not contain a redex). Consequently, if the term can be reduced, it must be reduced at root level.

The assumption mentioned above is not valid for the input term. for this purpose *normalize* is used. It gets a term, and (implicitly) traverses it in a bottom up fashion. At every level it computes the normal forms of the immediate sub-terms of its argument (by calling itself recursively). Then it computes the normal form of the entire term by passing it to *reduce*. Observe that *normalize* and *reduce* both have the same argument (a term), and both return the same result (the normal form).

The function *match* gets an open term (the left-hand side of a rewrite rule) and a subject term, and checks if the subject term is an instance of the pattern. If it is, the (possibly empty) list of variables in the pattern, together with their associated value is returned. Otherwise failure is indicated. It proves to be convenient to use a global variable in this function, which contains the list of variable-value pairs established so far. Initially this list is empty. It is extended on the fly.

The function *instantiate* gets an open term and the established list of variables and values. Note that, if efficiency is considered, it is not desirable to merely substitute all variables by their associated value, and to apply *normalize* to the result. The reason is, that in that way, information would be lost. Due to bottom up traversal the values associated with variables are already in normal form. This fact can be used.

*Instantiate* should use *reduce*. For this purpose it traverses the pattern bottom up, substituting variables for their associated values (without applying any function), and applying *reduce* on non-variable patterns.

Some auxiliary functions are:

- *term* creates a term with a given outermost function symbol, and list of immediate subterms;
- *ofs* returns the outermost function symbol of a given term;
- *args* returns the list of immediate subterms of a given term;
- *lhs* and *rhs* return the left-hand and right-hand side of a given rule, respectively;
- *lookup* retrieves the value associated with a given variable in a given list of variable-value pairs;
- *map* applies a given function to each value in a given list, returning the list of returned values;
- *each* applies a given function to each value in a given list (or pair of values taken from each list), and checks if that functions succeeds every time. In our case the function may have side effects;
- *isvar* checks if a given pattern represents a variable.

The implementation.

```
normalize(term)
  if args(term) = []                            /* if the term is a constant */
  then return reduce(term)                    /* all subterms in normal form */
  return reduce(term(ofs(term),                   /* normalize subterms, then */
                   map(normalize,args(term))))               /* reduce root */
reduce(term)
  for rule∈rules
    varval := []                 /* global variable value list, initially empty */
    match(term,lhs(rule))                    /* attempt to match this pattern */
    if varval != failure                                 /* rule applicable? */
    then return instantiate(rhs(rule))                   /* also uses varval */
  return term                                    /* term is in normal form */
match(term,pat)                                          /* also uses varval */
  if isvar(pat)                              /* if variable, store its value */
  then varval := varval + <pat,term>               /* vars unique in pattern */
  else
    if ofs(pat) != ofs(term) or                                 /* wrong ofs */
       not each(match,args(term),args(pat))                 /* any argument */
    then varval := failure
  return
```

```
instantiate(pat)                                    /* also uses varval */
  if isvar(pat)
  then return lookup(pat,varval)
  return reduce(term(ofs(pat),
                    map(instantiate,args(pat))))
```

We will now present the adaptations to this system, if term scanning is used. We will assume the existence of a function *scan*, which has been generated from the set of rules. This function gets a term and scans it using the method described in this chapter. The result is either failure, or one of the applicable rules, together with the corresponding variable-value list. Note that our scanner method actually yields all matching patterns, together with the corresponding variable-value lists. In our example implementation we cannot make use of this however, so the function *scan* simply returns an arbitrary pattern in that list.

The only function which needs to be altered is *reduce*. The function *match* is of course no longer needed.

```
reduce(term)
  scan := scan(term)          /* scan is a local variable */
  if scan = failure
  then return term                   /* no rule applicable */
  <rule,varval> := scan                      /* decompose */
  return instantiate(rhs(rule))
```

### 3.5.2 The Equation Interpreter

The Equation Interpreter by Hoffmann and O'Donnell ([36]) is an implementation of term rewriting systems. The system aims to provide equational logic as a programming language, with an efficiency comparable to symbolic functional languages such as LISP.

The two major factors determining the overall efficiency of a term rewriting implementation are:

- the total number of reductions (rewrite steps) used to reach normal form;
- the efficiency of pattern matching, especially if larger pattern sets are considered.

As it turns out these two problems are addressed simultaneously, in the EQI. We will first look at the number of reductions.

Under certain conditions – the term rewriting system must be regular – the concept of a **needed** redex can be defined. A needed redex in some term, is a subterm which can be rewritten (it matches one of the patterns) and which, in order for the entire term to reach normal form, must be rewritten at some point (that is, it will not be thrown away). If an implementation only rewrites needed redexes, as few reductions as possible are being done. Unfortunately it is in general undecidable whether a redex is needed. By restricting ourselves to the so-called strongly sequential term rewriting systems, a decision procedure at least exists. Again unfortunately, this decision procedure is very complex, and cannot be used efficiently in an implementation. For this reason Hoffmann and O'Donnell accept even stronger restrictions. In **strongly left-sequential** term rewriting systems a decision procedure as indicated above can be implemented efficiently. We will present this procedure shortly.

For completeness we will now list all restrictions imposed on the pattern set in the EQI.

1. no variable may occur more than once in a single pattern. For instance, $or(\alpha, \alpha)$ is not allowed.

2. no two patterns may match the same term. For instance, the patterns $or(\alpha, true)$ and $or(true, \beta)$ cannot occur together, because both match the term $or(true, true)$.

3. when two (not necessarily distinct) patterns match two different parts of the same term, they should not overlap. That is, $pred(succ(\alpha))$ and $succ(pred(\beta))$ are not allowed, because they overlap in the term $pred(succ(pred(zero)))$.

4. it must be possible to recognize an instance of any pattern in pre-order traversal without traversing any part of the term below that instance. For example, the patterns $f(g(\alpha, a), \beta)$ and $g(b, c)$ cannot occur together, because after scanning the partial term $f(g($ we cannot decide whether to read the next subterm, or skip it. Reading the next subterm would get us below the instance, if eventually the subject term matches the first pattern; skipping it would make it impossible to recognize the second pattern in pre-order traversal, should the subject term prove to be an instance of the second pattern.

Restrictions 1—3 together constitute regularity; restriction 4 constitutes strong left-sequentiality.

We will now look at the efficiency of pattern matching. As mentioned earlier, the term scanning technique using automata does not significantly deteriorate when the set of patterns grows, which is the first requirement.

A major difference between the EQI and our scanning method stems from rule 4 above. In the EQI, it can never happen that, of two patterns which share the same pre-order prefix, one proceeds with a variable and the other with a function symbol. That is, both may have a variable, in which case the entire subterm does not exist as far as the automaton is concerned (it is skipped with a tree move), or both have a function symbol, in which case scanning continues with that symbol. As we can see, the existence of variables can largely be ignored. In sections 3.2 and 3.3 it has already been established that the construction of automata in that case is much more straightforward.

In the EQI the scanning method using automata is extended using a technique described in [1], which addresses keyword searching in texts. There, a set of keywords is analyzed in order to extend a normal text scanning automaton – created from the keywords – with so-called **failure transitions**. Failure transitions are used to avoid re-visiting parts of the subject text after a failed match, because in this situation it may happen that a part of another keyword has already been recognized. Then, the automaton makes a failure transition to the state corresponding to that partially recognized keyword.

For instance, suppose the pattern set consists of the keywords ‘hamburg’ and ‘amsterdam’, and suppose the automaton is presented with the string ‘hamster...   ’. After reading the first three letters, the automaton is in a state corresponding to the partial keyword ‘ham’. Now an ‘s’ is read, and there is no suitable outgoing edge. There is however a failure transition from the state corresponding to ‘ham’ to that corresponding to ‘am’. This transition is taken (the symbol ‘s’ is not discarded), and scanning continues. Since the symbol ‘s’ was not discarded, it is now used in the transition to the state corresponding to ‘ams’, and the automaton continues precisely as if it had started scanning in its start state, with the symbol ‘a’ as first symbol.

Observe that every character in the subject text is looked at precisely once.

The failure transitions are determined statically by scanning the suffixes of all keywords. For every suffix of which at least one character is accepted, failure transitions are added (as many as there are matching characters). For instance, in the example above all suffixes of ‘hamburg’ are tried. The string ‘amburg’ leads to addition of two failure transitions, one from state ‘ha’ to state ‘a’, and one from state ‘ham’ to state ‘am’. Note that there is a failure transition precisely if a substring or suffix (that is, it may include the last character, but not the first) of one keyword, appears as a prefix of another keyword.

In the EQI this method has successfully been adapted to term scanning. Since variables can be ignored, the method of Aho and Corasick can be applied directly to the remaining strings of function symbols. Tree locations and tree moves are defined in a manner similar to that of our method, except for a small difference which we will now explain.

In our automata, tree moves are tagged to edges, because different paths may lead to the same state, as can be seen, for example, with the pattern set $\{f(\alpha, a),\ f(g(b), c)\}$. The state where the symbol $a$ will be accepted can be reached if at some point no $g$ is present, but also, from another state, if no $b$ is present. In the EQI this cannot happen. In accordance, tree moves are tagged to the state an edge leads to.

After the automaton has been created, failure transitions are added as described earlier. There is no tree move along failure transitions, because the function symbol which caused the failure still has to be processed (there is something like a virtual move of the root of the ‘current’ tree, due to the fact that the function symbols above the new pattern are ignored, which only means that the new tree location regarded as a string of integers, is a suffix of the old tree location, but no action is actually taken).

This brings us to the next question: what to do if the tree location of the end-state of a failure transition is not a suffix of that of the corresponding start-state, or, alternatively, if the tree move of that state (in the EQI states, not edges, are tagged with the tree move) differs from that of the start state (which was already performed in order to get the current symbol).

The first situation can only occur if the second situation occurs for some other failure transition, so we will only look at that. For an example, consider that pattern set $\{f(g(\alpha, a)), g(b, c)\}$. There is a failure transition from the state corresponding to $fg$ (location [1]) to the state corresponding to $g$ (location []). The tree move in the first state is $d$ 2, whereas that in the last state is $d$ 1.

The somewhat surprising result of Hoffmann and O'Donnell is that the term rewriting system under consideration is strongly left sequential **if and only if** this situation never happens. That is, the EQI simply attempts to create the automaton, with failure transitions. If at any point a failure transition leads to a conflict in tree moves, the term rewriting system is not strongly left-sequential. In fact, in the beginning this method was developed precisely to check strong left-sequentiality; only later was it also used in the rewrite process.

The automaton only accepts needed redexes. This fact is not as astounding as it may seem; it is a consequence of the fairly restrictive strong sequentiality. Without going into detail, the intuition can be sketched as follows. Starting at the root of a term, the automaton either recognizes a pattern or it fails somewhere. If a pattern is recognized, that redex is needed; since patterns cannot overlap the pattern would otherwise remain intact indefinitely. Now suppose the automaton does not succeed. Then it fails at the first position, in pre-order traversal, where the subject term differs from any pattern. Consequently, the subject term can only be changed above the current location if the current location itself changes. In other words, everything above it is **stable**. From this location on, possibly using a failure transition, scanning is continued. Each time, either a redex is found or another stable part is established. If a redex is found it is needed, because above it only stable parts occur. The stable parts cannot be reduced, so the current redex will not be thrown away. Hence it is needed. If no redex is found the stable part at the root location can never be reduced. In that case, the process is started separately on each of its immediate subterms.

Consider, for example, Figure 3.8, where a term is shown in which several redexes have partially been scanned, but led to failure at some point. Scanning continues each time (possibly avoiding double work by using a failure transition). Finally an entire redex may be recognized. This redex is needed because the entire path to the root is stable; rewritings elsewhere in the term could not change anything on that path. Consequently, the established redex will never be thrown away.

Note that if the automaton fails throughout, we cannot conclude that the term is in normal form. All we know is that a redex can never come into existence at root level. Consequently the term's outermost function symbol will remain, but all its immediate subterms still have to be checked. In this situation the EQI restarts its normalization process on each of the immediate subterms.

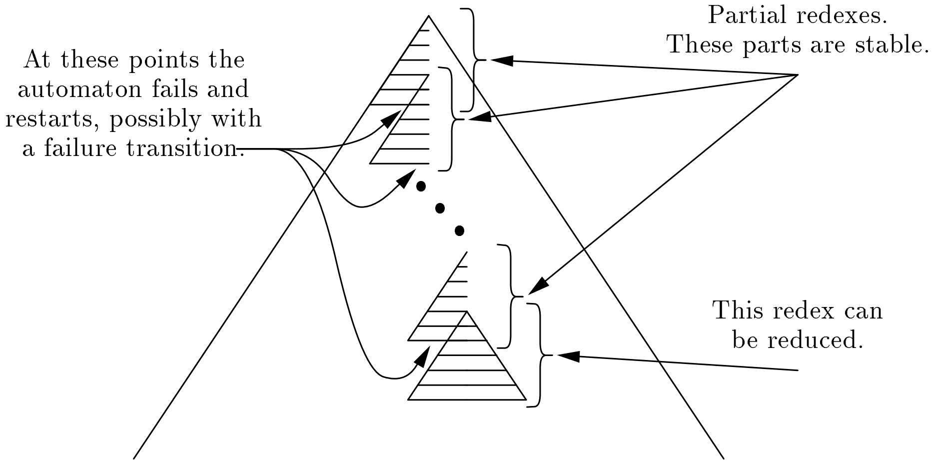



*Figure 3.8.*

With the results sketched above, both criteria mentioned earlier are met, to a certain degree:

- Since the term rewriting systems under consideration are strongly left-sequential, only needed redexes are reduced. Therefore the number of reduction steps is minimal;
- An automaton is used for term scanning. As mentioned the performance of such an automaton does not deteriorate significantly as the set of patterns grows. There are, however, some circumstances which should be noted.
  - ▷ In the text case every character is looked at once. This is not so in the term case. If an entire term is found to be stable, then the rewrite process is retried on its immediate subterms, which means that the same locations may be visited again. This matter is quite separate from failure transitions. This means, for instance, that if a term is already in normal form, eventually every node will be visited at least once.
  - ▷ A second issue which should be discussed is the gain we can expect from using failure transitions. Remember that a failure transition occurs precisely if a substring or suffix of one pattern occurs as prefix of another pattern.

In a term rewriting system, function symbols can roughly be divided into two groups: defined symbols and constructors. Any symbol which appears as the outermost function symbol of the left-hand side of a rewrite rule is a defined symbol. Any symbol occurring within the left-hand side of a rule is a constructor. In an arbitrary term rewriting system typically only a few function symbols cannot be categorized in this way; they are both defined and constructor symbols (this is an empirical fact, theoretically there are circumstances which support this view, but there is no necessity).

Since the patterns we are considering are the left-hand sides of the rewrite rules, almost all will begin with a defined symbol, and continue with constructors only. But then a substring or suffix of a pattern almost always begins with a constructor. This means that it will rarely appear as the prefix of another pattern.

▷ Finally we should mention that the aim of reducing needed redexes only, necessitates some extra scanning after each reduction. In order to find another needed redex, we would have to start scanning again at the root of the entire term, or, if a list of states, symbols and tree locations is kept for all partially accepted patterns, at a 'safe' lower position some levels above the current position. In the second approach it is not clear if the gain achieved outweighs the time lost due to extra bookkeeping. This matter is not discussed in [36].

It should be mentioned here, that the scanning technique was originally conceived as a method to check strong left-sequentiality, but apparently proved acceptable as an implementation for term matching.

### 3.5.3 Comparing the EQI and the current method

The advantage of the method used in the EQI is its efficiency in number of reduction steps. The disadvantages are the restrictions on the pattern set and the (in fact considerable) time spent in pre-processing. This pre-processing makes incremental maintenance of the pattern set hard.

In contrast, our method spends little or no time on pre-processing (allowing for incremental maintenance), and puts almost no restrictions on the pattern set (it allows for arbitrary term rewriting systems). The disadvantage, compared to the EQI, is that it does not support any special reduction strategy, such as that implied by strong left-sequentiality. Note in this context that the lack of any particular reduction strategy makes this method an interesting tool to use in conjunction with alternative methods aiming at reducing the number of reduction steps, such as for instance heuristic techniques.

Based on the considerations brought forth in this section we conclude that our method can achieve comparable performance with an extended functionality.

## 3.6 Conclusions

We have shown how text scanning automata can be used to perform tree matching, with the following adaptations:

- Special edges labeled $\Xi$ exist. These edges are only taken if the current function symbol does not appear as a label on any of the 'normal' edges;
- All edges are labeled with a tree move. This move is used to displace the 'current' location in the tree being scanned;
- All states are labeled with the set of patterns that have been partially recognized when the automaton is in that state. For final (accepting) states this is the set of patterns the subject tree is an instance of.
- Some states are labeled with a list of variables. When the automaton visits such a state this list is saved with the current tree location. When an accepting state has been reached, all variables occurring in the patterns recognized in that state have been remembered with their associated value.

In principle, arbitrary text scanner generators can be adapted accordingly.

The scanner generator of the ASF+SDF system has been conservatively extended with these adaptations. The generator is still being used to generate all text scanners. For text scanners, both the generation process and the generated scanners have lost less than 10% in speed, due to the changes.

The same generator also generates tree pattern matchers, which are being used in the term rewriting process. In a representative example, with 80 patterns, the entire rewrite process gained a factor 5 in speed, compared to an implementation based on straightforward matching against a list of patterns.

An important property of tree pattern matchers is that the time needed to establish a match does not directly depend on the size of the pattern set. Consequently the factor 5 mentioned above can be expected to increase if larger pattern sets are considered. In this context it should be mentioned that a straightforward implementation involves a sequential search in the entire list of patterns. When often used patterns appear at the end of that list, the efficiency will deteriorate. The efficiency of tree scanning automata is not influenced by the order of patterns (although the same order can be guaranteed to allow for priorities). The factor 5 mentioned above is representative for an arbitrary ordering of the 80 patterns; it varied from 2 to 20 for specific orderings. For larger pattern sets, in realistic situations, the minimal factor (2 above) will increase, because only so-many patterns can appear at the beginning of the list. Note that in general it is not possible to determine an 'optimal' order of all patterns, if arbitrary input is considered.

By using the existing scanner generator in the ASF+SDF system several features were automatically inherited:

- Lazy construction of the automaton. Only parts which are actually used are created;

- Incremental maintenance of the scanners. That is, a single pattern can be added to or deleted from the pattern set, without the need to reconstruct the entire automaton;
- Modular use of the scanners. Parts of the automata can be de-activated, depending on which patterns should be recognized.

The term scanners and generation process can successfully be applied to the term matching problem. The method imposes almost no constraints on the set of patterns. It can support lazy generation of the scanners, and incremental maintenance of the pattern set.

# Chapter 4

# Hybrid Implementations

Implementations of algebraic specifications based on term rewriting are usually too slow. We address this problem by introducing a formal framework in which implementations based on term rewriting can be combined transparently with implementations based on conventional programming languages. This hybrid method can even be used for the implementation of parameterized datatypes, where the formal parameter may have a formal signature and where the actual parameter may be implemented in either fashion. The hybrid method also permits later addition of equations over datatypes implemented in a conventional programming language, and allows for non-canonical terms to have an 'internal representation', i.e., a representation in the data domain of the conventional programming language.

## 4.1 Introduction

An 'implementation' of an algebraic specification can be defined as a procedure which checks whether some formula is a logical consequence of that algebraic specification. Unfortunately this is in general not decidable. By restricting the formulas to equations (that is, by checking only for the equality of terms) and by restricting the algebraic specifications to specifications which can be interpreted as confluent Noetherian term rewriting systems (TRSs), a decision procedure exists: put the two given terms in normal form and check whether the normal forms are syntactically equal [32], [21].

This method has one major disadvantage: the execution time is linear in the number of applications of equations. The aim of this chapter is to create a framework for hybrid implementations, in which a conventional programming language implementation can be combined transparently with a term rewriting implementation. Transparency here means that one need not be aware of the fact that parts of a specification have been implemented using a conventional programming language, and that changes to the specification do not imply changes to the standard implementation unless, of course, the semantics of the implemented functions has been changed.

In this way, low level, often used operations such as integer arithmetic can be implemented in a conventional programming language (or **cpl** for short), whereas the higher level computations will still be handled with term rewriting.

One can imagine how sorts and functions of elementary data types reside in a library with their signatures and equations visible to the users. Separate, and hidden from the users, are their actual implementations and the proofs of correctness of these implementations relative to the equations.

In order to understand this method we will first look at a simple example: the natural numbers. Most computers and programming languages have numbers as a standard datatype. All numbers (within a bounded though for practical purposes wide range) can be represented. The addition and multiplication operators are available and are executed very efficiently. In algebraic specifications the natural numbers are often defined using a constant $z$ and a monadic function $s$ (mapping each number on its successor). Addition and multiplication can be defined as follows:

$$
\begin{aligned}
&a(z, x) \rightarrow x \\
&a(s(x), y) \rightarrow a(x, s(y)) \\
&m(z, x) \rightarrow z \\
&m(s(x), y) \rightarrow a(m(x, y), y)
\end{aligned}
$$

Note that this specification, though compact, is not very efficient. For instance, in order to compute ten times ten more than a hundred rewritings are needed.

In our hybrid method we will keep the function symbols $z$, $s$, $a$ and $m$, but we will use the machine implementation for addition and multiplication.

Conceptually, we will extend the signature with a new constant for every number at the cpl-level (the datadomain of our cpl). In this chapter the cpl values and the corresponding added constants are denoted as $c_t$, where $t$ is the term in the original signature which corresponds to that value. For instance, the term $s(s(z))$ corresponds to the value 2. Consequently $c_{s(s(z))}$ is, depending on the context, the value 2 or the newly added constant formally representing 2.

This extension is conceptual because no actual representation of the entire signature is maintained. That is, such a constant is created only for those numbers which actually occur.

Also, rewrite rules are added which allow terms in the $s$-$z$ notation to be transformed to their corresponding new constants.

$$
\begin{aligned}
&z \rightarrow c_z \\
&s(c_t) \rightarrow c_{s(t)}
\end{aligned}
$$

Furthermore rules are added to map functions such as $a$ and $m$ to the corresponding cpl-level functions.

$$
\begin{aligned}
&a(c_s, c_t) \rightarrow < c_s + c_t > \\
&m(c_s, c_t) \rightarrow < c_s * c_t >
\end{aligned}
$$

The latter two rules clearly show a problem (though it is already present in the rules above): the right-hand side cannot properly be expressed in the algebraic framework. For this reason our theory is phrased slightly more complicated. In theory we add large, potentially infinite numbers of fixed rules to the term rewriting system. The implementation of our hybrid system uses rule schemes, each capable of mimicing large sets or fixed rewrite rules as suggested in the notation above. Formally the rules include:

$$s(c_z) \to c_{s(z)}$$
$$s(c_{s(z)}) \to c_{s(s(z))}$$
$$a(c_{s(s(z))}, c_{s(z)}) \to c_{s(s(s(z)))}$$

The final particularity of our hybrid system occurs when a term is being matched with the left-hand side of a rule (in order to determine whether the rule may be applied). If the term is one of the special constants $c_t$ it must be 'unwrapped' temporarily in order to compare its intended outermost function symbol with the corresponding symbol in the left-hand side. This unwrapping is necessary in order to guarantee the correctness and transparency of our system. This unwrapping is somewhat like applying rules such as $s(c_t) \to c_{s(t)}$ in reverse order. In practice the implementor provides functions which perform this unwrapping. See Section 4.9 for a function called `unpack`.

We will now give a step by step example of the entire rewrite process. We will show the added constants such as $c_{s(z)}$ as their cpl representation (i.e., 1).

For the purpose of illustration we will add a new function which is defined in the $s$-$z$ notation and which is 'unaware' of the existence of cpl implemented functions. The cut-off-difference is defined by:

$$d(x, z) \to \quad x$$
$$d(z, x) \to \quad z$$
$$d(s(x), s(y)) \to \quad d(x, y)$$

We will now normalize the term representing $((2 \times 3) - (1 + 1)) \times 3$.

$m(d(m(s(s(z)), s(s(s(z)))), a(s(z), s(z))), s(s(s(z)))) \twoheadrightarrow$ `five rewritings`
$m(d(m(s(s(0)), s(s(s(0)))), a(s(0), s(0))), s(s(s(0)))) \twoheadrightarrow$ `at once`
$m(d(m(s(1), s(s(1))), a(1, 1)), s(s(1))) \twoheadrightarrow$
$m(d(m(2, s(2)), a(1, 1)), s(2)) \twoheadrightarrow$
$m(d(m(2, 3), a(1, 1)), 3) \twoheadrightarrow$
$m(d(6, 2), 3) \twoheadrightarrow$ `calling two cpl operators`
$m(d(5, 1), 3) \twoheadrightarrow$ `temporarily unwrapping 6 and 2 to` $s(5)$ `and` $s(1)$
$m(d(4, 0), 3) \twoheadrightarrow$
$m(4, 3) \twoheadrightarrow$
12

After this brief introduction to our method we can now indicate how its correctness and completeness is established.

We will show the equivalence of our hybrid method and a traditional term rewriting implementation. The correctness of traditional implementations is already established in, for instance, [8] and [13].

To establish this equivalence one should note that our method is parameterized with a programming language and with arbitrary functions which can be written in that language. It is not our aim to prove the correctness of the implementations of those languages and functions; merely to show that the hybrid term rewriting implementation is correct and complete if all cpl functions are correct. The correctness of the cpl functions is based on the assumption that they are (in a sense to be made precise shortly) equivalent with some specification defining the same functions. For instance, in our examples above we assume the addition and multiplication operators to be consistent with the rewrite rules we presented for those functions. Note that this does not mean that our hybrid system must have a specification for every cpl implemented function. The system works well without a corresponding specification, but formal correctness is in that case meaningless.

We can now formalize our method. We start by selecting a set $N$ of terms which have a cpl representation. In our earlier example this set contained the terms $z$, $s(z)$, $\ldots$ , $s(\ldots s(z)\ldots)$, where the last term corresponds to the largest cpl representable integer. There is a constraint on $N$: it should be closed under subterm selection. That is, all subterms of terms in $N$ should also be in $N$. This constraint is no problem for basic datatypes, such as integers, but it seems to pose a problem for parameterized datatypes such as tables: the subterms of terms of such datatypes include the terms of the parameter types, and for reasons of transparency and extendibility we cannot assume all possible parameters to have a cpl implementation. The problem is solved by allowing $N$ to contain open terms, where the variables correspond to the fields in parameterized datatype values into which parameter values may be stored. For instance, the term $add(k_1, v_1, add(k_2, v_2, empty))$ could represent a table with two entries, where the two keys and associated values have yet to be filled in. Clearly the 'subterm constraint' on $N$ does not extend to sole variables.

Allowing $N$ to contain open terms, as opposed to ground terms, results in an essentially identical, though notationally more complicated theory. For this reason the theory is presented in two phases. In earlier sections we will assume $N$ to contain only ground terms. This restriction is slackened in Section 4.8.

For every ground term $t$ in $N$ (in the non-parameterized case) a new constant $c_t$ is introduced in the signature and equations will be added to the specification such that $\vdash t = c_t$. For an open term $t$ (with $n$ variables) the $n$-ary function symbol $\phi_t$ is introduced and equations will be added such that for any $t_1, \ldots, t_n$ (of appropriate sorts), and for the substitution $\sigma$ which assigns $t_1, \ldots, t_n$ to the variables in $t$, we have $\vdash t^\sigma = \phi_t(t_1, \ldots, t_n)$. In order to avoid a proliferation of notations we will assume that $c_t$ (or $\phi_t(\ldots)$) may, depending on the context, indicate that constant or term, or the cpl value we associate with that term. For instance, given that the integer 2 is associated with the term $s(s(z))$, then $c_{s(s(z))}$ may indicate either a constant term or the integer 2. The term $c_t$ (or $\phi_t(\ldots)$) is called an **internal representation** of $t$. The cpl value $c_t$ is called the cpl representation of $t$ (and $c_t$).

Given some term $t$, **alternative representations** can be constructed by replacing subterms $t_1, \ldots, t_n$ of $t$ with $c_{t_1}, \ldots, c_{t_n}$ (or $\phi_{t_1}(\ldots), \ldots, \phi_{t_n}(\ldots)$) respectively. Terms entirely of the form $c_t$ (or $\phi_t(\ldots)$) are called (totally) internal representations and the term $t$ is referred to as the term representation.

In an implementation every term has a representation in the language in which that implementation was made. This representation is a datastructure in which the tree-like construction of the term is reflected. We will call this representation the 'tree representation' and we will use the notation $[\![t]\!]$ to indicate the tree representation of a term $t$. Note that the values $[\![c_t]\!]$ and $c_t$ are distinct. The former is the tree representation of a term. The latter is a cpl value which we associate with that term. For example, suppose in some implementation terms are represented using records (structs) with a field named `fsym`, containing the outermost function symbol (for instance kept as a string), a field named `value` which is used only in the hybrid implementation, and zero or more fields named `arg1`, `arg2`, etcetera, containing pointers to the immediate subterms. Then given the term $t = s(z)$, $c_t$ equals 1, and $[\![t]\!]$ and $[\![c_t]\!]$ are as shown in Figure 4.1.

*Figure 4.1.*

Observe that $[\![c_t]\!]$ is the representation of a constant the name of which is chosen here to reflect the associated cpl value. Our only constraint is that this representation can be obtained from the cpl value, and vice versa.

Now, given some function symbol $f$, suppose a cpl function `f` exists such that if evaluation of $\mathtt{f}(c_{t_1}, \ldots, c_{t_n})$ yields $c_t$ (or that of $\mathtt{f}(s_1, \ldots, s_n)$ yields $s$, where (some or all of) $s_1, \ldots, s_n$ and $s$ are of the form $\phi_{t_1}(\ldots), \ldots, \phi_{t_n}(\ldots)$ and $\phi_t(\ldots)$ respectively), then a reduction sequence $f(t_1, \ldots, t_n) \xrightarrow{+} t$ exists (where $\xrightarrow{+}$ is the transitive closure of $\rightarrow$). Such a function `f` is said to **implement** $f$, and it can be used to define the new reduction $f(c_{t_1}, \ldots, c_{t_n}) \rightarrow c_t$ (or $f(s_1, \ldots, s_n) \rightarrow s$). This new reduction has the effect of a number of reduction steps in the original rewrite system. For instance, the `+` operator implements the function $a$, in our example earlier, because when `x+y` yields `z`, where `x`, `y` and `z` integers, then the normal form of $a(s(\ldots s(z)\ldots), s(\ldots s(z)\ldots))$ is $s(\ldots s(z)\ldots)$ (`x`, `y` and `z` times $s$, respectively).

The common notion of strategies as functions mapping terms onto their one step reducts is used with respect to the new rewrite relation. A **superstrategy** will be a function which maps terms onto a one-step reduct in this new rewrite relation. That is, a superstrategy is a strategy with respect to this new relation, which means that the effect of many rewriting can be achieved in a single step by using a cpl function.

In an actual implementation the transformation from terms to internal representations and vice versa should be avoided as much as possible. If the result of one computation is a cpl representation which can be used in another computation, then the corresponding term representation should, of course, not be reconstructed in between.

We will present a correctness proof of the hybrid method: if a cpl function correctly implements a function defined by a number of equations in a specification, then the combination of that cpl function with a standard (term rewriting) implementation of the remainder of that specification is equivalent to an implementation of the original specification. We stress that we will not investigate how the correctness of the cpl function is established.

The ideas sketched above will be presented in two phases. In Section 4.4 a restriction will be imposed on the set of terms that can be implemented using a cpl. As a consequence only nonparameterized data types can be handled. With this restriction a relatively simple theory can be formulated. In Section 4.8 this restriction is slackened, allowing for parameterized data types. The ensuing theory is slightly more complicated, though essentially similar. In Section 4.8 only parts of certain proofs are presented.

In Section 4.2 we will first present some common terminology and established results that are needed later on. In Section 4.3 we will indicate how a standard implementation of algebraic specifications can be defined. In Section 4.4 we will introduce the notion of internal representations, to be made formal in Section 4.5. In Section 4.6 three adaptations of specifications and framework are presented. In Section 4.7 a new rewrite relation is introduced and properties equivalent to those in Section 4.2 will be established. In Section 4.8 limitations on cpl representations imposed in Section 4.4 will be removed. In Section 4.9 the standard implementation is presented in an algorithmic notation, together with a proof of its correctness. This implementation is then extended to allow for cpl-implemented functions, according to the theory developed in this chapter. In Appendix A an example TRS is presented with further details of its implementation. The standard implementation coded in Lisp can be found in Appendix A as well. Finally, the Lisp implementation with cpl extension is presented, both for parameterized and nonparameterized situations. Appendix B contains proofs of various theorems.

The ideas presented in this chapter are also investigated in [27] and [38]. Neither article addresses the problem of parameterized datatypes nor allows for non-normal forms to have an 'internal' representation, as is necessary to allow for additions of arbitrary equations. In [38] the problem of matching modulo a congruence is addressed. No formal proofs are given in [38].

## 4.2 Algebraic specifications and TRSs $\bowtie$[1]

In this section we will give a brief introduction to algebraic specifications and we will give a formal definition of what we mean by an 'implementation of an algebraic specification'. We will introduce term rewriting systems (TRSs) and discuss some of their properties. Finally we will present the definition and correctness proof of an implementation scheme based on TRSs (see also [6] and [32]).

An algebraic specification is a pair $(\Sigma, E)$ where $\Sigma$ is a signature and $E$ a set of equations. A signature contains a set of sort names and a set of function symbols. Every function symbol has an arity (the number and respective sorts

[1]There is a minor difference in notation, between this chapter and Chapter 2. See definitions 1 and 2.

of its arguments) and a result sort. A function without arguments is called a constant.

In this chapter we will not consider the problem of typechecking. We will assume every function symbol to be uniquely typed; if two function symbols are the same, then they refer to the same function.

The set $T_\Sigma$ of ground terms over $\Sigma$ is defined as the smallest set satisfying:

if $s_1, \ldots, s_n \in T_\Sigma$ and for each $i$, $s_i$ has sort $S_i$, and $f\colon S_1 \times \ldots \times S_n \mapsto S$, then $f(s_1, \ldots, s_n) \in T_\Sigma$.

The set $T_{\Sigma,V}$ of open terms, where $V$ is a set of (implicitly sort-annotated) variable names, is the smallest set satisfying both the condition above and:

$x \in V \Rightarrow x \in T_{\Sigma,V}$

In this chapter we will assume all open terms to be linear (i.e. no variable occurs more than once). Consequently, 'variable' and 'variable occurrence' may be used interchangeably. Allowing for nonlinear open terms, however, poses no essential problems.

The operator # yields the number of variables in a term. Its value is 0 for ground terms. Given an open term $t$, the notation $v_i^t$ indicates the $i$-th variable in $t$ (as encountered in the standard prefix textual representation). As an example, consider the term $t = *(+(x,y), +(0, -(z)))$. Then $\#t = 3$, $v_1^t = x$, $v_2^t = y$ and $v_3^t = z$.

An equation is a formula $s = t$, where $s$ and $t$ are open terms. An equation is (implicitly) universally quantified over all variables that occur in it.

In this chapter we will only consider the equality of terms. That is, we will consider the meaning of a specification to be the set of equations $s = t$ which can be derived using the reflexivity, symmetry and transitivity of equality, the fact that equal terms are interchangeable in any context, and the fact that equal (open) terms remain equal under any substitution. We will use the common notation $(\Sigma, E) \vdash s = t$ to indicate that the equality of two terms is a consequence of $(\Sigma, E)$ in equational logic.

An implementation of a specification is a procedure which takes two terms as arguments. The procedure may or may not terminate. Upon termination it should indicate success or failure. Furthermore, an implementation `p` should satisfy the following condition.

if `p(s,t)` terminates indicating success, then $(\Sigma, E) \vdash \mathtt{s} = \mathtt{t}$.

A term rewriting system (TRS) is a pair $(\Sigma, R)$, where $R$ is a set of rewrite rules. A rewrite rule is a formula $s \to t$, where $s$ and $t$ are open terms satisfying the following conditions.

- $s$ is not a single variable (i.e., contains at least one function symbol);
- all variables occurring in $t$ also occur in $s$.

A **substitution** assigns terms to the variables occurring in an open term; it is a function from $V$ to $T_\Sigma$. An **instantiation** $t^\sigma$ of an open term $t$ given a substitution $\sigma$ is the term we get by replacing every variable in $t$ with its $\sigma$-value. A context is a term with a hole in it. Formally, the set of contexts is the smallest set satisfying:

- $[.]$ is a context.
- if $C[.]$ is a context, and if $f(s_1, \ldots, s_n)$ is a term, then $f(s_1, \ldots, s_{i-1}, C[.], s_{i+1}, \ldots, s_n)$ is a context.

The instantiation $C[s]$ of a context is defined by

- if $C[.] = [.]$ then $C[s] = s$.
- if $C[.] = f(s_1, \ldots, s_{i-1}, C'[.], s_{i+1}, \ldots, s_n)$ then $C[s] = f(s_1, \ldots, s_{i-1}, C'[s], s_{i+1}, \ldots, s_n)$.

Given a TRS $(\Sigma, R)$, the relations $\rightarrow$ (which should not be confused with the syntactic appearance of rewrite rules), $\xrightarrow{*}$ and $\xrightarrow{+}$ can be defined as follows.

**1. Definition**

$s \rightarrow t \overset{\text{def}}{\Longleftrightarrow}$

$$\exists C[.]\ \exists s_0 \rightarrow t_0 \in R\ \exists \sigma\colon V \mapsto T_\Sigma\colon s = C[s_0^\sigma] \wedge t = C[t_0^\sigma]$$

$\xrightarrow{*}$ is the smallest set satisfying

$s \xrightarrow{*} t \overset{\text{def}}{\Longleftrightarrow}$

$$s = t \text{ or } \exists u \in T_\Sigma\colon s \rightarrow u \text{ and } u \xrightarrow{*} t$$

$\xrightarrow{+}$ is the smallest set satisfying

$s \xrightarrow{+} t \overset{\text{def}}{\Longleftrightarrow}$

$$\exists u \in T_\Sigma\colon s \rightarrow u \text{ and } u \xrightarrow{*} t$$

where in each case $s$, $t$ and $u$ are ground terms

This means that $s$ can rewrite to $t$ if a context $C[.]$, a rewrite rule $s_0 \rightarrow t_0 \in R$ (where $R$ is the set of all rewrite rules), and a substitution $\sigma$ exist, such that $t$ can be derived from $s$ by replacing a subterm (namely $s_0^\sigma$, which is called the **redex**) in $s$ which is an instantiation of the left-hand side of a rewrite rule by the corresponding instantiated right-hand side. For instance, $a(m(s(z), s(z)), s(z)) \rightarrow a(a(m(z, s(z)), s(z)), s(z))$ using $C[.] = a(., s(z))$, $s_0 \rightarrow t_0 = m(s(x), y) \rightarrow a(m(x, y), y)$ and $\sigma = \{x \mapsto z, y \mapsto s(z)\}$.

The relation $\xrightarrow{+}$ is the transitive closure of $\rightarrow$, and $\xrightarrow{*}$ is the reflexive transitive closure of $\rightarrow$.

Any occurrence $r^\sigma$ for which there are a context $C[.]$, a substitution $\sigma$ and a reduction rule $r \rightarrow r'$ such that $s = C[r^\sigma]$, is called a **redex** (reducible expression) in $s$.

**2. Definition**
A **strategy** is a function $F: T_\Sigma \mapsto T_\Sigma$ for which $dom(F) = \{s | \exists t \in T_\Sigma: s \rightarrow t\}$ and which satisfies $\forall s \in dom(F): s \rightarrow F(s)$.

This means that a strategy applies the rewrite relation. Strategies can be characterized by the redex they select for reduction and by the rewrite rule used for that reduction [38]. For instance, the leftmost innermost reduction strategy takes the leftmost redex (by any rule) which does not contain another redex.

Note that all strategies have the same domain.

For any strategy $F$ we define

- $F_0$ by $F_0(s) = s$.
- $F_n$ by $F_n(s) = F(F_{n-1}(s))$ if $n \geq 1$ and $F_i(s) \in dom(F)$ for each $i < n$.
- $F_\omega$ by $F_\omega(s) = F_n(s)$ if an $n$ exists such that $F_n(s) \notin dom(F)$, and for each $i < n$, $F_i(s) \in dom(F)$.

For any strategy $F$ and for any term $s$ in $T_\Sigma$, $F_\omega(s)$ is called a normal form of $s$.

**3. Lemma**
For every strategy $F$ the following four properties are easily verified:

1. if the sequence $s, F(s), F_2(s), \ldots$ is infinite, then $s \notin dom(F_\omega)$;
2. for $s \notin dom(F)$ we have $F_\omega(s) = s$;
3. for $n > 0$ we have $s \xrightarrow{+} F_n(s)$ (if $F_n(s)$ exists);
4. for $s \in dom(F_\omega)$ we have $s \xrightarrow{*} F_\omega(s)$.

The reason that we have looked at TRSs is that we can interpret an equation as a rewrite rule simply by applying it only from left to right. The importance of the relation $\rightarrow$ is that if $s \rightarrow t$ then $\vdash s = t$, where the proof contains the application of only one equation, the undirected version of the rewrite rule used to establish $s \rightarrow t$. This is phrased in a slightly different way in the following theorem.

**4. Theorem**
Let $(\Sigma, E)$ be an algebraic specification and let $(\Sigma, R)$ be the corresponding TRS. For all $s, t \in T_\Sigma$ such that $s \xrightarrow{*} t$ we have $(\Sigma, E) \vdash s = t$.

Since this Theorem, and the two below can be found in any standard text, we will not present proofs [32].

A TRS is called **confluent** if all diverging reduction sequences can again converge, i.e., if for all $s, t_1, t_2 \in T_\Sigma$ with $s \xrightarrow{*} t_1$ and $s \xrightarrow{*} t_2$ some $u$ exists, such that $t_1 \xrightarrow{*} u$ and $t_2 \xrightarrow{*} u$.

**5. Theorem**
Let $(\Sigma, R)$ be a confluent TRS. Let $s \in T_\Sigma$ and let $F$ and $G$ be two strategies. If $s \in dom(F_\omega) \bigcap dom(G_\omega)$ then $F_\omega(s) = G_\omega(s)$.

A TRS is called Noetherian if no infinite reduction sequences exist.

**6. Theorem**
Let $(\Sigma, R)$ be a Noetherian, confluent TRS. Let $s \in T_\Sigma$ and let $F$ and $G$ be two strategies. Then $F_\omega(s) = G_\omega(s)$.

The importance of this theorem is that it guarantees us that for a confluent Noetherian TRS all strategies have the same result.

In the rest of this chapter we will only consider confluent, possibly Noetherian TRSs.

## 4.3 Term rewriting implementations $\bowtie$

The theorems in the previous section are the theoretical basis for the implementation of algebraic specifications based on TRSs. From Theorem 4 we see that applying a strategy any number of times to some term yields a term which can be proved equal to the original term. Indeed, every application of the strategy is equivalent to one step in the formal proof.

From Theorem 5 we know that if the TRS is confluent, it does not matter which strategy we take, as long as it terminates. If the TRS is Noetherian, we know that any strategy will terminate. If $F_\omega(s)$ exists, it is the unique normal form of $s$. If two terms have the same normal form, they are provably equal; the two derivations to normal form are the two halves of the proof.

Suppose `f` is a function which implements a strategy $F$. That is, if $[\![s]\!]$ and $[\![t]\!]$ are tree representations of $s$ and $t$ respectively, and if $F(s) = t$, then the evaluation of `f(`$[\![s]\!]$`)` should yield $[\![t]\!]$, and for $s \notin dom(F)$, evaluation of `f(`$[\![s]\!]$`)` should indicate failure.

For the leftmost outermost reduction strategy, for instance, this function is constructed as follows. The term $s$, and subsequently all its subterms, are matched with the left-hand sides of all reduction rules. When a match is found, the substitution defined by that match is used to construct an instantiation of the right-hand side of the rule thus found. This instantiation is put in the same context as the matched subterm, and a reduct has been established.

Now define $\mathtt{f}_\omega$ and `p` as follows.

- $\mathtt{f}_\omega$`(s) :`
```
      while (f(s) succeeds)
           do s := f(s) od
      yield(s)
```

- `p(s,t) :`
```
      fω(s) = fω(t)
```

Procedure p is an implementation in the sense defined in the previous section:

- if p(s,t) succeeds, we have that both $\texttt{f}_\omega$(s) and $\texttt{f}_\omega$(t) terminate, and that the results are equal. Hence $(\Sigma, E) \vdash s = t$.
- otherwise p fails or doesn't terminate at all. It fails if $\texttt{f}_\omega\texttt{(s)} \neq \texttt{f}_\omega\texttt{(s)}$, it doesn't terminate if the evaluation of $\texttt{f}_\omega$(s) or $\texttt{f}_\omega$(t) doesn't terminate.

Observe that our only assumption is a confluent TRS. Any strategy will do. If the TRS is Noetherian, then we know that p always terminates.

In Appendix A a more detailed implementation in our algorithmic notation is presented, together with a proof of its correctness. Apart from the procedure p and the functions f and $\texttt{f}_\omega$, the implementation defines the functions match, which matches a term with the left-hand side of a rewrite rule, in order to recognize an instance, and to determine a substitution, and the function inst which instantiates the right-hand side of a rewrite rule, given some substitution.

## 4.4 Internal representations

In a standard TRS implementation terms are represented by trees. In this section we will argue that a good approach to representing terms by arbitrary cpl values in the context of TRS implementations, is to add new terms to the signature. For now, we will impose a restriction on the set of terms which can be represented by a cpl value. We will assume that only ground terms have a cpl representation, and that cpl-implemented functions only take cpl values as arguments. These restrictions are slackened in Section 4.8.

A cpl value, such as an integer, has no internal structure in the context of terms and TRSs, and can therefore only be formalized as a constant. The term corresponding to that value however, may not be a constant. Consequently, more than one formal object may correspond to the same term in the original TRS.

If the term $t$ corresponds to the cpl value $c_t$, then the new constant $c_t$ is added to the signature. This constant is called the (totally) internal representation of $t$. If $t$ is an arbitrary term, a more internal representation of $t$ can be found by replacing some subterms of $t$ by their internal representation. If all largest possible subterms have been replaced by their most-internal representation, the result is again a most-internal representation. Note that consequently not every most-internal representation is of the form $c_t$.

Let $N \subseteq T_\Sigma$ be an arbitrary set of ground terms closed under subterm-selection (that is, all subterms of a term in $N$ are also in $N$). For instance $N$ might contain all terms over $\{z, s(.)\}$.

For every $t$ in $N$ define a new constant $c_t$, and let $\Sigma^+$ be the set of those constants. Let $\Sigma^* = \Sigma \cup \Sigma^+$ be the union of the old signature with these constants, and let $T_{\Sigma^*}$ be the ground term space over $\Sigma^*$.

In the following section we will show that a partial order on alternative representations exists, and that every term has a unique infimum and supremum with respect to this order. The supremum is the term representation of a term and it is unique in $T_\Sigma$. The infimum is the most-internal representation of the term; it is the unique term in which as many subterms as possible have been replaced by their cpl representation.

Informally this partial order works as follows. Highest in the order are terms in the original signature. Terms lower down are obtained by replacing some subterms with their associated constants. Lowest down are the terms in which as many subterms as possible (and possibly the entire term) have been replaced. For instance, consider the following diagram, which contains all terms comparable to $m(s(z), s(s(z)))$. In this diagram constants such as $c_{s(z)}$ are shown as the corresponding integer (e.g., 1 for $c_{s(z)}$).

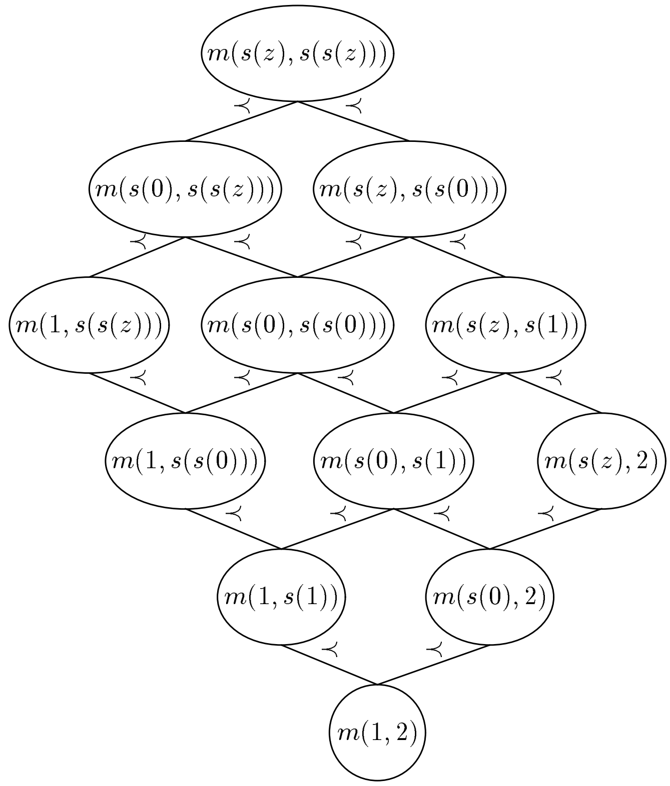



*Figure 4.2.*

In an implementation it is undesirable to find the internal representation of a term every time we need it. Much of the gain in speed achieved by using a cpl implementation would be lost in conversions between alternative representations. For the same reason it is not useful to maintain both internal and term representations; changes in the one too often necessitate the entire reconstruction of the other.

In our system any representation may occur. The system itself favors internal representations in the following sense:

- Terms are transformed to internal representation and kept in that form as much as possible;
- If part of the term representation of some internal representation is needed, as little as possible is 'unwrapped'; the part which was needed is reconstructed with immediate subterms still in internal representation.

The first property is achieved by adding rules to the TRS which perform the transformation to internal representation. The second property is achieved by redefining the concept of matching, so that an open term matches not only all its instantiations, but also all more-internal representations thereof. As a matter of fact a particular implementation is used of 'matching modulo alternative representation'.

In the following section we will extend our concept of rewriting.

- Using the rules mentioned above, the transformation of a term to a more internal representation is added to the rewrite relation;
- Applying the new definition of matching when trying to rewrite a term adds many rewritings on terms in the new signature which are consistent with the old rewrite relation;
- Cpl functions will be used to define rewritings which can transform terms in one step to a many-step reduct.

Allowing for cpl-implemented functions where some of the arguments may not have an internal representation poses a slight problem which will be covered in later sections. For the moment we will only regard cpl-implemented functions for which internal representations exist for all arguments.

The next question is: which terms should be contained in the set $N$, or in other words will have a cpl representation? For the moment we will assume the set of terms which have a cpl representation to be closed under subterm selection. That is, if a term has a cpl representation, then so do all its subterms.

In many cases only normal forms will have a cpl representation, and such an assumption allows for certain optimalizations in the implementation. For reasons of transparency, however, we will allow arbitrary terms to have an internal representation.

We will also assume that the cpl representation of terms is unique, that is, for every cpl representation a unique term can be constructed which is represented by that cpl value.

In Figure 4.3 the old and extended rewrite relations are shown. The upper part contains the term space of the original signature ($T_\Sigma$). Arrows in that part are possible rewritings according to the original rewrite relation $\rightarrow$.

Lower down in the figure are terms containing constants in $\Sigma^+$. The set $\Sigma^+$ is of course at the bottom, but terms not in $\Sigma^+$ may still be most internal representation (such as $m(c_z, c_z)$). Arrows pointing down transform terms to more internal representations (e.g. $s(c_z) \to c_{s(z)}$). The dashed arrows are parallel to the $\to$-arrows. They lead from more internal representations of terms to an alternative representation of whatever the corresponding $\to$-arrow is pointing to. These arrows are a consequence of the special kind of matching used; terms may be 'unwrapped' in order to apply the original rewrite rules. Usually terms need not to be unwrapped entirely, as is indicated by the fact that most dashed arrows do not end in $T_\Sigma$. Finally, the curved arrows represent the application of cpl functions.

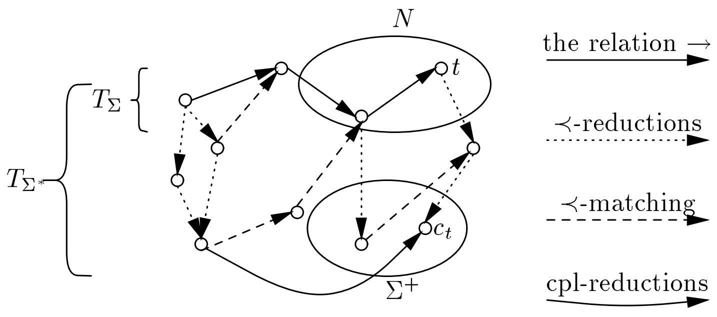



*Figure 4.3.*

## 4.5 $\preceq$ and $\prec$, $\lceil . \rceil$ and $\lfloor . \rfloor$

Let $(\Sigma, R)$ be a confluent TRS. Let $N \subseteq T_\Sigma$ and let $N$ be closed under subterm selection. Formally,

$\forall t \in N\colon t = f(t_1, \ldots, t_n) \Rightarrow t_1, \ldots, t_n \in N$

For each term $t \in N$ we define a new, unique constant $c_t$ ($c_t \notin \Sigma, c_s = c_t \Rightarrow s = t$).

Let $\Sigma^+ = \{c_t | t \in N\}$ and let $\Sigma^* = \Sigma \cup \Sigma^+$. Now define the relations $\preceq$ and $\prec$ on terms in $T_{\Sigma^*}$, where $s \prec t$ means that $s$ is a more internal representation of $t$.

**7. Definition**

$s \preceq t \stackrel{\text{def}}{\Longleftrightarrow}$

$$\begin{array}{ll} s = t \text{ or} & \\ s = f(s_1, \ldots, s_n), t = f(t_1, \ldots, t_n), s_1 \preceq t_1, \ldots, s_n \preceq t_n \text{ or} & (1) \\ \exists f(t_1, \ldots, t_n) \in N\colon s = c_{f(t_1, \ldots, t_n)} \text{ and } f(c_{t_1}, \ldots, c_{t_n}) \preceq t & (2) \end{array}$$

$s \prec t \stackrel{\text{def}}{\Longleftrightarrow} s \preceq t \wedge s \neq t$

This means that $s$ occurs below $t$ if either they have the same outermost function symbol and the immediate subterms are ordered appropriately, or $s$ is itself a constant in $\Sigma^+$, and if we unwrap $s$ once we get a term which is equal to $t$ or is smaller than $t$.

For every $\Sigma^*$-term there is exactly one $\preceq$-supremum (which is in $T_\Sigma$) and exactly one $\preceq$-infimum (in $T_{\Sigma^*}$).

Example: earlier we saw that $m(c_{s(z)}, s(c_{s(z)})) \preceq m(s(c_z), s(s(c_z)))$. We can now derive this. First we apply clause (1) and we must show that $c_{s(z)} \preceq s(c_z)$ and $s(c_{s(z)}) \preceq s(s(c_z))$. We apply clause (1) to the second condition. We must now show that $c_{s(z)} \preceq s(c_z)$. This is established immediately by applying clause (2).

Similarly, we conclude that the infimum of $m(c_{s(z)}, s(c_z))$ is $m(c_{s(z)}, c_{s(s(z))})$ and its supremum is $m(s(z), s(s(z)))$.

**8. Theorem**

$$\forall s \in T_{\Sigma^*} \exists! t \in T_{\Sigma^*} \colon s \preceq t \text{ and } \forall s' \in T_{\Sigma^*} \colon s \preceq s' \Rightarrow s' \preceq t$$

and for this $t$ we have $t \in T_\Sigma$.

**9. Theorem**

$$\forall s \in T_{\Sigma^*} \exists! t \in T_{\Sigma^*} \colon t \preceq s \text{ and } \forall t' \in T_{\Sigma^*} \colon t' \preceq s \Rightarrow t \preceq t'$$

We will denote the $\preceq$-supremum of a term $s$ by $\lceil s \rceil$ and the $\preceq$-infimum by $\lfloor s \rfloor$.

The proof of these theorems is given in Appendix B.

## 4.6 Three adaptations

In this section three adaptations of the standard implementation (see Section 4.3) will be presented, which allow us to utilize cpl functions.

### 4.6.1 $\prec$-reduction

In order to convert a term to its internal representation, we have to add rewrite rules to the TRS.

$N$ is closed under subterm selection. Consequently it is sufficient to add rules of the form $f(c_{s_1}, \ldots, c_{s_n}) \rightarrow c_{f(s_1,\ldots,s_n)}$.

Note that with these rules a normal form may no longer be a normal form. Instead, if $s$ was a normal form, then now $\lfloor s \rfloor$ will be one.

Let $R^\prec$ be the set of rules $f(c_{s_1}, \ldots, c_{s_n}) \rightarrow c_{f(s_1,\ldots,s_n)}$, where $f$ ranges over all function symbols occurring in $N$ and where $s_1$ to $s_n$ range over all terms in $N$ for which $f(s_1, \ldots, s_n) \in N$. These rules will be called $\prec$-rules.

As mentioned earlier, these rules are only added theoretically. In the implementation a single rule scheme is added for every function symbol occurring in $N$. An example of such a rule scheme is $s(c_t) \to c_{s(t)}$, where $t$ is, of course, not a normal variable. When $t$ ranges over the applicable terms in $N$ the rule scheme above ranges over all fixed rules.

### 4.6.2 $\prec$-matching

When a term has been transformed to internal representation, rewrite rules may no longer apply to it, even though they apply to the term representation of that term, because the new constants in $\Sigma^+$ do not match any function symbol in $\Sigma$. Nevertheless we would still like to apply these rules; they may be essential to reach a suitable normal form. We introduce $\prec$-matching to solve this problem.

Remember that $s \to t$ means that a context $C[.]$, a substitution $\sigma$ and a rewrite rule $s_0 \to t_0$ exists, such that $s = C[s_0^\sigma]$ and $t = C[t_0^\sigma]$.

We introduce $\prec$-matching as a re-definition of term matching, such that any $u \preceq C[s_0^\sigma]$ matches with $C[s_0]$. Obviously $\prec$-matching should also allow $\sigma$ to range over $T_{\Sigma^*}$, so that subterms which play no part in matching can be in any representation. In the situation above, $\prec$-matching $u$ with $C[s_0]$ gives rise to some (not uniquely defined) substitution $\sigma'$ for which $\lceil x^{\sigma'} \rceil = x^\sigma$ for every $x$ in $V$. Then, rewriting $u$ to $C[t_0^{\sigma'}]$ is consistent with the rewrite rule $s_0 \to t_0$.

For example, the rewrite rule $d(s(x), s(y)) \to d(x, y)$ (third rule for cut-off-difference) cannot be applied to the term $d(c_{s(z)}, c_{s(s(z))})$ using normal matching. During $\prec$-matching, however, $c_{s(z)}$ and $c_{s(s(z))}$ are temporarily unwrapped to $s(c_z)$ and $s(c_{s(z)})$, respectively, and the term is rewritten to $d(c_z, c_{s(z)})$ by $\sigma'$ such that $\sigma'(x) = c_z$ and $\sigma'(y) = c_{s(z)}$.

This alternative matching should not be applied when considering $\prec$-rules, because the right-hand side of such a rule always matches ($\prec$-matches) its left-hand side.

The definition above is suitable for our theoretical purposes, but it does not show how a particular substitution $\sigma'$ can be found.

In Appendix B a constructive match routine is defined which satisfies the description above.

### 4.6.3 The cpl connection

A cpl function `f` implements a function $f: S_1 \times \ldots \times S_n \mapsto S$ with respect to the TRS $(\Sigma^*, R)$ and the set of terms $N$ if, for all appropriate $s_1, \ldots, s_n \in N$ the following situation occurs. When evaluation of $\mathtt{f}(c_{s_1}, \ldots, c_{s_n})$ yields $c_t$, then there exists a reduction sequence $f(s_1, \ldots, s_n) \to \ldots \to t$.

Now suppose `f` implements $f$. Then we may add the rules $f(c_{s_1}, \ldots, c_{s_n}) \to c_t$, where $t$ can be computed by evaluating $\mathtt{f}(c_{s_1}, \ldots, c_{s_n})$. The special property of the function `f` mentioned above guarantees that any added reduction is only a shortcut for a reduction sequence which already existed.

All rules thus added will be called cpl rules and the set of cpl rules will be denoted by $R^{cpl}$.

At this point one may wonder why the rules concerning $f$ cannot be removed from the TRS. The reason is simple. These rules are still needed for terms not in $N$, which can only be handled in the normal fashion.

Even if $N$ does contain all terms, it is still undesirable to remove the original rules for reasons of transparency. Any extension of the signature would render the implementation invalid.

Since both the cpl rules and the original rules are present, the implementation must take care not to use an original rule when a cpl rule is applicable, or may become applicable at some later point. This latter situation occurs, for instance, when a term has subterms in $N$, which have not yet been transformed to their internal representation. In this case the cpl rules are not yet applicable, whereas the original rules may well be applicable. A way to do this is to put a term in normal form with respect to the $\prec$-rules before considering any other rule.

At this point we would like to mention that $\prec$-rules and cpl rules are very similar. Rules in $R^{\prec}$ look like $f(c_{s_1}, \ldots, c_{s_n}) \to c_{f(s_1,\ldots,s_n)}$, where $f(s_1, \ldots, s_n)$ ranges over $N$; cpl rules look like $f(c_{s_1}, \ldots, c_{s_n}) \to c_t$, where $t$ is a reduct of $f(s_1, \ldots, s_n)$. The only difference is that application of a cpl rule is equivalent with at least one reduction in the original TRS, whereas application of a $\prec$-rule yields a term $\preceq$ the argument, and is therefore equivalent to zero reductions in the original TRS. Although $\prec$-rules and cpl rules can be handled in the same way, to a large degree, and are implemented identically, as will be shown in subsequent sections, we will see that the difference mentioned above is essential.

## 4.7 Superstrategies and the relation $\searrow$

In this section we will define the alternative rewrite relation $\searrow$, which incorporates the extensions presented in the previous section. Because extended matching subsumes ordinary matching, and because the old rewrite rules are rules of $\searrow$, the relation $\searrow$ contains the relation $\to$. Since cpl-rules are added, however, this new relation contains many more term rewritings. We will define superstrategies as strategies with respect to $\searrow$, and we will establish properties of superstrategies, similar to known properties of strategies. A superstrategy is called 'super' because it can achieve the equivalent of many reduction steps in a single step, by using cpl rules.

We will consider the TRS $(\Sigma, R)$ with the extensions presented so far. Let $R^* = R \cup R^{\prec} \cup R^{cpl}$.

**10. Definition**

$$
\begin{array}{l}
s \searrow t \overset{\text{def}}{\Longleftrightarrow} \\
\qquad \exists C[.]\ \ \exists \sigma : V \mapsto T_{\Sigma^*}: \\
\qquad\qquad \exists s_0 \to t_0 \in R^{\prec}: s = C[s_0^\sigma] \wedge t = C[t_0^\sigma] \qquad \bigvee \\
\qquad\qquad \exists s_0 \to t_0 \in R \cup R^{cpl}: s \preceq C[s_0^\sigma] \wedge t = C[t_0^\sigma]
\end{array}
$$

$s \searrow^* t \stackrel{\text{def}}{\Longleftrightarrow}$

$$s = t \text{ or } \exists u \in T_{\Sigma^*}: s \searrow u \text{ and } u \searrow^* t$$

$s \searrow^+ t \stackrel{\text{def}}{\Longleftrightarrow}$

$$\exists u \in T_{\Sigma^*}: s \searrow u \text{ and } u \searrow^* t$$

We will first show that this definition incorporates all extensions presented so far.

Obviously $\prec$-rules and cpl-rules have been added. $\prec$-Matching is incorporated because now any $s \preceq s_0^\sigma$ is acceptable, which is precisely the extended match condition. Since $\sigma$ ranges over $T_{\Sigma^*}$, $s$ need not be unwrapped more than necessary. Finally, observe that extended matching is not applicable to terms which have been reduced by a $\prec$-rule, as stipulated earlier.

**11. Definition**
A superstrategy is a function $F: T_{\Sigma^*} \mapsto T_{\Sigma^*}$ for which $dom(F) = \{s | \exists t \in T_{\Sigma^*}: s \searrow t\}$ and which satisfies $\forall s \in dom(F): s \searrow F(s)$.

All superstrategies have the same domain.

For a superstrategy $F$ the functions $F_n$ and $F_\omega$ are defined in the same way as for strategies.

We will now present three theorems establishing the equivalence of strategies and superstrategies. The proofs of these theorems, and various properties of the relation $\searrow$ will be presented in Appendix B.

**12. Theorem**
If $(\Sigma, R)$ is confluent w.r.t. $\rightarrow$, then so is $(\Sigma^*, R^*)$ w.r.t. $\searrow$.

**13. Theorem**
If $(\Sigma, R)$ is Noetherian w.r.t. $\rightarrow$, then so is $(\Sigma^*, R^*)$ w.r.t. $\searrow$.

**14. Theorem**
Let $(\Sigma, R)$ be a Noetherian, confluent TRS and let $(\Sigma^*, R^*)$ be as described. Let $F$ be a strategy and let $\check{F}$ be a superstrategy.

$$\forall s \in T_\Sigma: F_\omega(s) = \lceil \check{F}_\omega(s) \rceil$$

This theorem states the equivalence of a traditional implementation scheme and our hybrid scheme.

Note that the only restrictions imposed are that the set $N$ of cpl represented term is closed under subterm selection, and that cpl functions correctly implement the corresponding equations. Soundness and completeness of our method is guaranteed by this theorem and by the soundness and completeness of traditional term rewriting implementations [8, 13].

An implementation, in our algorithmic notation, based on the theory presented above can be found in Section 4.8. This implementation supersedes and partially uses the term rewriting implementation also presented in that section. In Appendix A the Lisp version of this implementation can be found.

## 4.8 Parameterized data types

In Sections 4.4 and 4.5 a restriction was imposed on $N$, the set of terms with a cpl representation. For any term in $N$, all subterms must also be in $N$. This restriction is reasonable for primitive data types, such as integers or characters. For parameterized data types such as lists or tables, the restriction is undesirable, because, for reasons of transparency, we cannot assume every actual parameter to be implemented using a cpl.

From the implementational point of view there is no real problem. Type-checking is done separately, so at run time all objects are of the same cpl data type; they are all tree representations. Since all objects are of the same kind, one can make an implementation of a parameterized data type without knowing anything about the actual parameter. The fields in a data structure not 'belonging' to that data type all contain values of parameter types.

This description points us in the direction of a solution. Where in an implementation fields of a parameterized data structure can be set to any term, variables in an open term can be instantiated with any term of the proper sort. The sort constraint is dealt with during typechecking and need not concern us.

We will redefine $N$ to be a set of (possibly) open terms. We will assume $N$ to be closed under subterm selection, with the exception of variables. That is, all subterms of a term in $N$, which are not sole variables (they may of course contain variables), are also in $N$.

In previous sections we associated a new constant with every term in $N$. Now such a term may contain variables and may consequently yield different instantiations when subjected to different substitutions. For every open term with $n$ variables a new, unique $n$-ary function symbol is added to the signature.

A third consequence of this new approach concerns how a cpl function implements the semantics of some function symbol. Before, we could assume the immediate subterms of the left-hand side, and the entire right-hand side of a cpl-rule to be in internal representation. Now, this is no longer the case. As an example consider a parameterized specification of association lists (defining a sort *list*). Such a specification has two parameters: the sort *key*, which is used to associate values under, in a list, and the sort *value*, which contains whatever is being stored. For such a specification to be meaningful, at least an equality predicate must be defined on the sort *key*, and an undefined value on the sort *value*. We will consider the cpl rule for the function *lookup*, which, given a list and a key, returns whatever value is stored in the list under that key (or undefined if the key is not present). The left-hand side has a so-called **limited variable** for the argument of sort *list*. This means that lists must appear in internal representation. The *key* argument however, may not even have an internal representation, and must be passed to the right-hand side as a term. The right-hand side also does not yield an internal representation. It is a term of sort *value*, which is taken from one of the fields in the list argument. When we look closer at the right-hand side, we see another problem: In order to determine which value must be returned, keys in the list must be compared to the argument key. That is, the equality of keys must be determined using the equality predicate on the sort *key*. For this purpose the equational identity must be established of two terms, for which we fortunately have a decision procedure: the procedure `p`. In general the right-hand side may yield, and may depend on the equality of, terms which are

1. internal representations of sort(s) being defined;
2. arguments of the function call of any other sort, which may be in any representation;
3. values of any sort occurring, as instantiations of fields, inside the internal representations which were passed to the function;
4. terms constructed of the terms mentioned under 1-3, together with function symbols occurring in the signature of the formal parameters of the data type being defined (such as the equality predicate in the example above).

Note that 'the sort(s) being defined' is not a precise formulation, as we can see in the example above: the function *lookup* has as much to do with the sorts *key* and *value* as with the sort *list*. What should actually be read is something like 'the sort(s) for which an internal representation is being defined in the same scope or module and the existence of which may be assumed for the definition of this function'.

With these alterations, to be presented in more detail shortly, the entire construction from previous sections, and all results established so far, remain valid. Most formulations will be more complicated and for that reason we have dealt with the simpler case first.

In the remainder of this section we will rephrase definitions and properties in this new framework, highlighting proofs only occasionally. In order not to introduce more notational conventions than necessary, we will reuse those established in earlier sections. We will forget all definitions and properties in sections 4.5, 4.6 and 4.7.

### 4.8.1 Internal representation of open terms

Let $(\Sigma, R)$ be a confluent, possibly Noetherian TRS.

Let $N \subseteq T_{\Sigma,V} \backslash V$ be a set of open terms which is closed under subterm selection, with the exception of variables. Formally

$$\forall f(t_1, \ldots, t_n) \in N \quad \forall i \colon (t_i \in N \otimes t_i \in V)$$

Where $\otimes$ means either, but not both.

Observe that $N$ may contain ground terms, so our current definition covers the construction in Section 4.6. Also note that no term in $N$ can be a single variable.

Remember that we are considering linear open terms only, that $\#t$ is the number of variables in a term $t$, and that if $t$ is open, it contains the variables $v_1^t$ to $v_{\#t}^t$ as they occur in the textual (pre-order) representation of $t$.

For each term $t \in N$ which contains exactly $n$ variables, we define a new, unique $n$-ary function symbol $\phi_t : S_1 \times \ldots \times S_n \mapsto S$, where $S_i$ is the sort of $v_i^t$, and $S$ is the sort of $t$.

Informally we may think of such $n$-ary function symbols $\phi_t$ as datastructures in which fields must yet be filled in. The term $\phi_t(s_1, \ldots, s_n)$ is the datastructure which we get by putting the terms $s_1$ through $s_n$ into those fields.

For example, suppose we are considering lists. Function symbols are $empty$, for the empty list, and $add$, which adds an item to a list. The set $N$ contains $empty$, $add(x, empty)$, $add(x, add(y, empty))$, and so forth. The open term $add(x, add(y, empty))$ is associated with *all two-element lists* whatever the actual parameters are. The term $\phi_{add(x,add(y,empty))}(z, s(z))$ is the internal representation (using non cpl implemented integers as actual parameter) of the list $add(z, add(s(z), empty))$.

The notation $\Phi_i^t(\ldots)$ is used to describe the unwrapping of terms. For instance, unwrapping $\phi_{add(x,add(y,empty))}(s, t)$ once, yields $add(s, \phi_{add(y,empty)}(t))$, which can be written as $add(\Phi_1^{add(x,add(y,empty))}(s, t), \Phi_2^{add(x,add(y,empty))}(s, t))$ if $\Phi$ is used. Loosely speaking, $\Phi_i^t$ is the internal representation (inheriting variables) of the $i$-th subterm of $t$, and $\Phi_i^t(s_1, \ldots, s_n)$ is the internal representation using the appropriate instantiation (which depends on which subterms of $t$ have how many variables).

**15. Definition**

$$\Phi_i^{f(t_1,\ldots,t_n)} = \begin{cases} t_i & \text{when } t_i \in V; \\ \phi_{t_i} & \text{when } t_i \in N \text{ and } t_i \text{ is ground;} \\ \phi_{t_i}(v_1^{t_i}, \ldots, v_{\#t_i}^{t_i}) & \text{when } t_i \in N \text{ and } t_i \text{ is open.} \end{cases}$$

$$\Phi_i^t(s_1, \ldots, s_{\#t}) = (\Phi_i^t)^\sigma \quad \text{where } \sigma \text{ is such, that } (v_j^t)^\sigma = s_j \text{ for } j \leq \#t.$$

For example consider the term $t = f(g(w, x), h(y), z)$. Then, $\#t = 4$, $v_3^t = y$, $v_1^{g(w,x)} = w$, etcetera; $\Phi_1^t = \phi_{g(v,w)}(v, w)$ and $\Phi_3^t = y$, etcetera; and $\Phi_1^t(s_1, s_2, s_3, s_4) = \phi_{g(v,w)}(s_1, s_2)$ and $\Phi_3^t(s_1, s_2, s_3, s_4) = s_3$, etcetera.

Now, let $\Sigma^+ = \{\phi_t | t \in N\}$ and let $\Sigma^* = \Sigma \cup \Sigma^+$.

**16. Definition**

$s \preceq t \overset{\text{def}}{\iff}$

$$s = f(s_1, \ldots, s_n), t = f(t_1, \ldots, t_n), s_1 \preceq t_1, \ldots, s_n \preceq t_n \text{ or} \tag{1}$$

$$\begin{aligned} &\exists f(t_1, \ldots, t_n) \in N, \exists k: \\ &\quad \#f(t_1, \ldots, t_n) = k \wedge \\ &\quad (k = 0 \wedge s = \phi_{f(t_1,\ldots,t_n)} \vee \\ &\quad\; k > 0 \wedge s = \phi_{f(t_1,\ldots,t_n)}(s_1, \ldots, s_k)\ ) \wedge \\ &\quad f(\Phi_1^{f(t_1,\ldots,t_n)}(s_1, \ldots, s_k), \ldots, \Phi_n^{f(t_1,\ldots,t_n)}(s_1, \ldots, s_k)) \preceq t \end{aligned} \tag{2}$$

$s \prec t \overset{\text{def}}{\Longleftrightarrow}$

$$s \preceq t \text{ and } s \neq t$$

Intuitively this ordering is identical to the ordering in the non-parameterized case. That is, smaller terms are in their most internal representation, and progressing upwards terms are more and more unwrapped, until, at the top, they appear entirely in the original signature.

The relation $\preceq$ is again a partial order. As an example, the proof of transitivity of $\preceq$ is shown in Appendix B.

As before, every term has exactly one $\preceq$-supremum (which is in $T_\Sigma$) and exactly one $\preceq$-infimum (in $T_{\Sigma^*}$).

**8. Theorem**

$$\forall s \in T_{\Sigma^*} \exists! t \in T_{\Sigma^*} : s \preceq t \text{ and } \forall s' \in T_{\Sigma^*} : s \preceq s' \Rightarrow s' \preceq t$$

and for this $t$ we have $t \in T_\Sigma$.

**9. Theorem**

$$\forall s \in T_{\Sigma^*} \exists! t \in T_{\Sigma^*} : t \preceq s \text{ and } \forall t' \in T_{\Sigma^*} : t' \preceq s \Rightarrow t \preceq t'$$

**17. Definition**

- $\lceil f(s_1, \ldots, s_n) \rceil = f(\lceil s_1 \rceil, \ldots, \lceil s_n \rceil)$
  for $f \in \Sigma$;
- $\lceil \phi_{f(r_1,\ldots,r_n)}(s_1, \ldots, s_k) \rceil =$
  $f(\lceil \Phi_1^{f(r_1,\ldots,r_n)}(s_1, \ldots, s_k) \rceil, \ldots, \lceil \Phi_n^{f(r_1,\ldots,r_n)}(s_1, \ldots, s_k) \rceil)$
  for $\phi_{f(r_1,\ldots,r_n)} \in \Sigma^+$;
- $\lfloor s \rfloor = \phi_t(\lfloor s_1 \rfloor, \ldots, \lfloor s_k \rfloor)$
  when a $t \in N$ and a substitution $\sigma$ exist, such that for each $i$ we have $(v_i^t)^\sigma = s_i$ and finally $s = t^\sigma$;
- $\lfloor f(s_1, \ldots, s_n) \rfloor = f(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor)$
  otherwise.

The proofs of the two theorems above are straightforward. Both are structured as in Section 4.6, although now they are based on Definition 16.

### 4.8.2 The three adaptations

Let $R^{\prec}$ contain rules $f(\Phi_1^t(v_1^t,\ldots,v_{\#t}^t),\ldots,\Phi_n^t(v_1^t,\ldots,v_{\#t}^t)) \to \phi_t(v_1^t,\ldots,v_{\#t}^t)$, where $t = f(t_1,\ldots,t_n)$ ranges over $N$.

$\prec$-Matching can be phrased in precisely the same way as before. Given a term $s$ and the left-hand side of a rewrite rule $s_0 \to t_0$, matching succeeds if there exist a substitution $\sigma: V \mapsto T_\Sigma$ and a context $C[.]$ such that $s \preceq C[s_0^\sigma]$. In this case a substitution $\sigma': V \mapsto T_{\Sigma^*}$ is found, such that $s \preceq C[s^{\sigma'}]$.

As before, this alternative matching should not be applied when considering the $\prec$-rules, and as before, in our implementation we will only use strategies which reduce a term before putting it 'inside' an internal representation.

The third adaptation involves cpl functions. We must now take into consideration the fact that cpl values may be parameterized with arbitrary terms. That is, if $t$ is an open term in $N$, then the corresponding cpl value is some data structure $\phi_t(\ldots)$ which has fields corresponding to the variables in $t$, and $\phi_t(s_1,\ldots,s_{\#t})$ is the data structure in which the indicated terms have been placed in those fields.

A cpl function `f` implements the semantics of a function $f: S_1 \times \ldots \times S_n \mapsto S$ with respect to the TRS $(\Sigma^*, R)$, if the following situation occurs. Suppose $s_1,\ldots,s_n \in T_{\Sigma^*}$ are of sorts $S_1,\ldots,S_n$ respectively, and suppose for every sort $S_i$ which for the definition of $f$ must be assumed to have internal representation, a $t^i \in N$, and $t_1^i,\ldots,t_{\#t^i}^i$ exist such that $s_i = \phi_{t^i}(t_1^i,\ldots,t_{\#t^i}^i)$. Furthermore suppose that evaluation of `f(`$[\![s_1]\!]$`,...  ,`$[\![s_n]\!]$`)` yields some value $[\![s]\!]$. Then a reduction sequence $f(\lceil s_1\rceil,\ldots,\lceil s_n\rceil) \xrightarrow{+} \lceil s\rceil$ must exist in $(\Sigma, R)$.

If `f` implements the semantics of $f$, then we may add the rewrite rules $f(s_1,\ldots,s_n) \to s$. All rules thus added will be called cpl rules and the set of cpl rules will be denoted by $R^{cpl}$.

As before, these rules are only added in theory; in practice a single rule scheme exists per cpl implemented function.

Consider for example a *sort* function on lists. One of the rules added to $R^{cpl}$ is $sort(\phi_{add(x,add(y,empty))}(s(s(z)),s(z))) \to \phi_{add(x,add(y,empty))}(s(z),s(s(z)))$. The rule scheme for this function cannot easily be shown because it depends on an ordering predicate on the actual parameter sort. For integers this might be the less-than predicate; for strings the lexicographic ordering. In the parameterized case, cpl functions have a way to 'call' a function defined on the formal parameter. See Section 4.5 and Appendix A for examples.

### 4.8.3 Superstrategies and the relation $\searrow$

Let $R^* = R \cup R^{\prec} \cup R^{cpl}$.

**18. Definition**

$$
\begin{array}{l}
s \searrow t \overset{\text{def}}{\iff} \\
\qquad \exists C[.]\ \ \exists \sigma: V \mapsto T_{\Sigma^*}: \\
\qquad\qquad \exists s_0 \to t_0 \in R^{\prec}: s = C[s_0^\sigma] \wedge t = C[t_0^\sigma] \bigvee \\
\qquad\qquad \exists s_0 \to t_0 \in R \cup R^{cpl}: s \preceq C[s_0^\sigma] \wedge t = C[t_0^\sigma]
\end{array}
$$

$s \searrow^* t \stackrel{\text{def}}{\Longleftrightarrow}$

$$s = t \text{ or } \exists u \in T_{\Sigma^*}: s \searrow u \text{ and } u \searrow^* t$$

$s \searrow^+ t \stackrel{\text{def}}{\Longleftrightarrow}$

$$\exists u \in T_{\Sigma^*}: s \searrow u \text{ and } u \searrow^* t$$

**19. Definition**

A superstrategy is a function $F: T_{\Sigma^*} \mapsto T_{\Sigma^*}$ for which $dom(F) = \{s | \exists t \in T_{\Sigma^*}: s \searrow t\}$ and which satisfies $\forall s \in dom(F): s \searrow F(s)$.

**20. Theorems**

1. If $(\Sigma, R)$ is confluent w.r.t. $\rightarrow$, then so is $(\Sigma^*, R^*)$ w.r.t. $\searrow$;
2. If $(\Sigma, R)$ is Noetherian w.r.t. $\rightarrow$, then so is $(\Sigma^*, R^*)$ w.r.t. $\searrow$.
3. Let $(\Sigma, R)$ be a Noetherian, confluent TRS and let $(\Sigma^*, R^*)$ be as described. Let $F$ be a strategy and let $\breve{F}$ be a superstrategy.

$$\forall s \in T_\Sigma: F_\omega(s) = \lceil \breve{F}_\omega(s) \rceil$$

The earlier proofs of these theorems remain largely valid. Only two lemma's (as presented in Appendix B) need to be established in a different way. In Appendix B the first lemma, and its new proof are presented. The proof of the second lemma is left as an exercise.

A striking feature of the implementation of parameterized data types is that it is identical to the implementation presented earlier, of non-parameterized data types. The only difference occurs in generators and in the function `unpack`.

In Appendix A a Lisp implementation of the algorithms presented in Section 4.8 can be found, taking its rules from an example TRS.

## 4.9 An implementation $\bowtie$

In this section we will present an implementation scheme of algebraic specifications using an algorithmic notation. It is not difficult to transform this notation to actually working programs, although that transformation would involve some detail. First, we will show the standard TRS implementation scheme. Then we will present the hybrid system. In Appendix A we will present a detailed example of a particular TRS, together with all TRS-dependent algorithms, and a Lisp implementation.

The implementation scheme presented in this section assumes terms and substitutions to have been implemented in some way. We will freely use functions and operators on those data types, all with the obvious meaning.

### 4.9.1 A TRS implementation

In the formal definition of a rewrite step, an applicable context and substitution must be found. In the program below we will not explicitly look for a context; instead, we will simply attempt to match a rewrite rule on the given term and all its subterms, until we succeed. A substitution is constructed on the fly, while matching the left-hand side of a rewrite rule with some term. The first time a variable is encountered, the corresponding subterm is remembered; all further occurrences of that variable can only syntactically match the same term. Remember that the right-hand side of a rewrite rule contains only variables which are present in the left-hand side.

The example defines the following functions.

- `match`, to match an open term (the left-hand side of a rewrite rule) with a given term, which yields a substitution;
- `inst`, to instantiate an open term, given a substitution;
- `ff`, the implementation of a strategy $F$;
- $\mathtt{f}_\omega$, the implementation of $F_\omega$;
- `p`, the procedure which is a decision procedure as specified in the beginning of this section.

The `match` function has one extra parameter, the desired substitution which must initially be empty and which is extended on the fly.

```
match(pat,term,sub):
    case pat of
    variable: if pat ∈ dom(sub)
              then if sub(pat) ≠ term
                   then exit(failure)
                   else exit(sub)
              else exit(sub + <pat↦term>)
    nonvar:   with pat = f(p1,...  ,pn), term = g(t1,...  ,tk)
                   if f ≠ g              note: if f=g then n=k
                   then exit(failure)
                   else sub0 := sub
                        for i:=1 to n
                            subi := match(pi,ti,subi-1)
                            if subi = failure
                            then exit(failure)
                        exit(subn)


inst(pat,sub):
    case pat of
    variable: exit(sub(pat))
    nonvar:   with pat = f(p1,...  ,pn)
                  exit(f(inst(p1,sub),...  ,inst(pn,sub)))
```

```
ff(term,rules)
    with term = f(t1,...  ,tn)
        if n>0
        then for i:=1 to n
                 tmp := ff(ti, rules)
                 if tmp ≠ failure
                 then exit(f(t1,...  ,ti-1,tmp,ti+1,...  ,tn))
    for s→t ∈ rules
        sub := match(s, term, ∅)
        if sub ≠ failure
        then exit(inst(t,sub))
    exit(failure)
```

```
fω(term,rules)
    repeat
        tmp := term
        term := ff(tmp, rules)
    until term = failure
    exit(tmp)
```

```
p(s,t,rules)
    tmp := fω(s,rules)
    if tmp ≠ failure and tmp = fω(t,rules)
    then exit(ok)
    else exit(failure)
```

The strategy thus implemented is called the leftmost innermost strategy because the deepest possible context, with its hole occurring as far left as possible is attempted first. Different strategies can be achieved by adapting the `with` clause, in the body of `ff`, resulting in parallel strategies, and by exchanging the `for` and the `with` clause, resulting in outermost strategies [6].

**21. Property**
The procedure `p` is an implementation of $(\Sigma, R)$.

In the proof below we will use the notation introduced in Section 4.1. If $s$ is some term, then $[\![s]\!]$ is the tree which represents $s$ in the implementation (i.e., $[\![s]\!]$ is the tree representation of $s$). It is a value which is or refers to the data structure representing $s$, and which can be passed to, and can be returned by functions.

First we will show that `match` is correct. To do this, we will show that `match(`$[\![s]\!]$`,`$[\![t]\!]$`,`$\emptyset$`)` yields a substitution $[\![\sigma]\!]$ precisely when $s^\sigma = t$, and that `match(`$[\![s]\!]$`,`$[\![t]\!]$`,`$\emptyset$`)` fails precisely when $t$ is not an instantiation of $s$. In order to show this we will establish a slightly more general invariant. We will show that `match(`$[\![s]\!]$`,`$[\![t]\!]$`,`$\sigma$`)` yields the minimal substitution $[\![\sigma']\!]$ such that $s^{\sigma'} = t$, where $dom(\sigma) \subseteq dom(\sigma')$ and $\forall x \in dom(\sigma)\colon x^\sigma = x^{\sigma'}$ (that is, $\sigma'$ is a conservative extension of $\sigma$), and fails if no such substitution exists.

We will use induction on the complexity of $s$. Suppose $s$ is a constant. Then `match` will execute the `nonvar` section. Failure is indicated when $s$ and $t$ are unequal, and since the only instantiation of a constant is that constant itself, this is correct. If `match` succeeds, then it is by the `else` part in the `nonvar` section which can only be reached when $s$ and $t$ are equal. For constant $s$, we have $n = 0$, and the substitution returned is the unaltered argument $\sigma$.

Now suppose $s$ is a variable. Failure is indicated when $s \in dom(\sigma)$, but $t \neq s^\sigma$. Clearly no suitable substitution exists. If $t = s^\sigma$ then $\sigma$ itself suffices and is indeed returned. If $s \notin dom(\sigma)$ then $\sigma'$ is taken to be the extension of $\sigma$ which maps $s$ to $t$.

Finally, suppose $s$ is some compound term. If the outermost function symbols are unequal, then failure is indicated. Otherwise, the sequence of substitutions $\sigma = \sigma_0 \ldots \sigma_n$ is constructed, such that `match` on the $i$-th sub-pattern and sub-term, with $\sigma_{i-1}$ as argument substitution yields $\sigma_i$. Failure results when any recursive call yields failure. If a substitution $\sigma_n$ is found, then by induction it instantiates every subpattern to the corresponding subterm, and consequently $s^{\sigma_n} = t$. If failure results, one of the subterm matches fails. Since all substitutions are minimal, no suitable substitution exists.

We have now established the invariant. Since `match` is called with $\emptyset$ as argument substitution, `match` succeeds precisely when $t$ is an instantiation of $s$, and the minimal substitution is returned.

In a similar fashion we can establish that `inst(`$[\![s]\!]$`,`$[\![\sigma]\!]$`)` yields $[\![s^\sigma]\!]$.

Now we will first show that if evaluation of `ff(`$[\![u]\!]$`)` yields $[\![v]\!]$, then $u \rightarrow v$, and, for any $u$ for which a $v$ exists, such that $u \rightarrow v$ we have that `ff(`$[\![u]\!]$`)` succeeds. Evaluation of `ff(`$[\![u]\!]$`)` indicates failure precisely when $u$ is a normal form.

We will again use induction on the complexity of $u$. If $u$ is a constant, then `ff(`$[\![u]\!]$`)` can only succeed by the `exit` in the `for` clause. Then, apparently there is a rule the left-hand side of which matches with $u$. Since the right-hand side of a rewrite rule cannot introduce a new variable, that right-hand side must be a ground term, which has itself as instantiation. We have established that $u \rightarrow v$, where $[\![v]\!]$ `=ff(`$[\![u]\!]$`)`. If `ff` fails, no rule with matching left-hand side exists and $u$ is indeed a normal form.

Now suppose $u$ is a compound term. If `ff` succeeds by the for clause, then $u$ matches the left-hand side of some rule $s \rightarrow t$, by some substitution $\sigma$. We have established that $u = s^\sigma \rightarrow t^\sigma$, and the latter value is indeed returned. If `ff` succeeds by the `with` clause, then a recursive call to `ff` on one of the subterms apparently succeeds. By induction we have established a context $C[.]$ such that $u = C[s^\sigma]$ and $v = C[t^\sigma]$ and we have shown that $u \rightarrow v$. If `ff` fails, no context, rule and substitution exist, and hence $u$ is a normal form.

We have shown that `ff` implements a strategy.

It is obvious that $\mathtt{f}_\omega$`(`$[\![s]\!]$`)` succeeds always, yielding some $[\![t]\!]$, such that, $t = F_\omega(s)$.

Finally we conclude that `p` is an implementation in the sense of Section 4.2.

### 4.9.2 A hybrid implementation

In this section we will present a cpl-implementation (non-parameterized) based on the theory developed in Section 4.7. In Appendix A a Lisp implementation along these lines can be found.

We will first repeat some notions introduced earlier. Every term $t$ may have an internal representation $c_t$, which is another term. In an implementation, every term ($t$, but also $c_t$) has a tree representation ($[\![t]\!]$ and $[\![c_t]\!]$ respectively), but only terms such as $c_t$ have a cpl value (also denoted by $c_t$).

For our implementation we will assume that the cpl value $c_t$ and the tree representation $[\![c_t]\!]$ can be trivially determined from each other. A possibility to achieve this is, for instance, to inject all relevant cpl values into the set of names of constants. Furthermore we will assume that from the data structure $[\![c_{f(s_1,\ldots,s_n)}]\!]$, the function symbol $f$ and the values $c_{s_1}, \ldots, c_{s_n}$ can be derived, and vice versa.

Finally, in the previous section we have seen that $\prec$-matching should not be applied when considering rules in $R^{\prec}$. For this reason we will assume that all rules in $R^{\prec}$ are marked. The function `match` gets one extra parameter, a boolean value indicating whether the rule under consideration is marked.

Furthermore, we will assume the existence of the following functions.

- `internal`, where `internal(`$[\![t]\!]$`)` yields $true$ when t is an internal representation in $\Sigma^+$, and $false$ otherwise;
- `val`, where `val(`$[\![c_s]\!]$`)` yields $c_s$;
- `make`, where `make(`$c_s$`)` yields $[\![c_s]\!]$;
- `unpack`, where `unpack(`$[\![c_{f(s_1,\ldots,s_n)}]\!]$`)` yields $[\![f(c_{s_1}, \ldots, c_{s_n})]\!]$.

In earlier sections large, potentially infinite numbers of rules were added to the TRS under consideration. It is undesirable and impossible to add these rules explicitly, so in our implementation we will use rule schemes. A rule scheme is an implementational device which simulates a large number of rules.

For every set of rules $f(c_1, \ldots, c_n) \to c$ for $c_1, \ldots, c_n, c \in \Sigma^+$, where $c$ can be derived from $c_1, \ldots, c_n$ using some computable function, a rule scheme is added which in the implementation acts as if it is a single rule, with variables as immediate subterms of its left-hand side, but which upon activation computes the proper right-hand side, using the actual instantiations of those variables. Clearly these variables are very special, in that they should only match terms in $\Sigma^+$. For this purpose we will introduce the notion of *limited variables*. A limited variable will only match a term in $\Sigma^+$. As a consequence of the context in which limited variables are used, we may assume that such a variable occurs not more than once in a left-hand side.

The right-hand side of a rule scheme is also non-standard. It must be computed given the substitution which defines the values of the (limited) tree variables in the left-hand side. For this purpose we will introduce a new kind of right-hand side: a *generator*. A generator is simply a callable function which, when given some substitution which assigns internal values to limited variables, returns the right-hand side corresponding to those values. Note that a generator exists for every $\prec$-rule scheme and for every cpl-rule scheme. This means that a generator exists for every function symbol in $N$ and every cpl-implemented function.

It may be assumed that these rules are linear (i.e., that every variable occurs exactly once). Non-linear rules are not as useful as in normal algebraic specifications. To implement non-linear rules, either the function `match` should be adapted, or the generator should check for equality.

Below we will only show the definitions of the functions `match` and `inst`. Generators cannot be exhibited without looking at a particular example. See Appendix A. The functions `p`, `ff` and $\mathtt{f}_\omega$ remain as they were, with the exception of the marking of rules in $R^{\prec}$, and the fact that this value must be supplied to `match`. Particular examples of generators can be found in Appendix A.

```
match(pat,term,sub,inR≺):
    case pat of
    variable: if pat ∈ dom(sub)
              then if sub(pat) ≠ term
                   then exit(failure)
                   else exit(sub)
              else exit(sub + <pat↦term>)
    limited:  if not internal(term)
              then exit(failure)
              else exit(sub + <pat↦term>)
    otherwise:if internal(term)
              then if inR≺
                   then exit(failure)
                   else term := unpack(term)
              with pat = f(p1,...  ,pn), term = g(t1,...  ,tk)
                   if f ≠ g
                   then exit(failure)
                   else sub0 := sub
                        for i:=1 to n
                            subi := match(pi,ti,subi-1,inR≺)
                            if subi = failure
                            then exit(failure)
                        exit(subn)


inst(rhs,sub):
    case rhs of
    variable: exit(sub(rhs))
    generator:exit(rhs(sub))
    otherwise:with rhs = f(p1,...  ,pn)
                  exit(f(inst(p1,sub),...  ,inst(pn,sub)))
```

A remark on efficiency is in order. As can be seen, the function `match` 'unwraps' internal representations always, when they do not match a limited variable, and when a rule not in $R^{\prec}$ is being considered. In the lisp implementation in Appendix A this loss in efficiency is somewhat reduced by annotating tree representations of internal representations with the outermost function symbol of the unpacked version. The result is that a term is only actually unpacked when its outermost function symbol is as required.

## 4.10 Conclusions

We have presented a method for the hybrid implementation of algebraic specifications. We have used an algorithmic notation which can easily be transformed to a real programming language. In Appendix A, a prototype based on our method and written in Lisp (LeLisp 15.22, INRIA) is presented. The prototype implements the left-most innermost reduction strategy.

In Figure 4.4 some measurements using this prototype are shown. Measurements are performed on a Sun 4 sparcstation 1+. Execution times shown are user seconds.

The term rewriting system used contains rules for Booleans, natural numbers and lists. On Booleans and natural numbers the usual operators and predicates are defined; on lists a sort and a union operator are defined. Three hybrid versions exist.

1. For hybrid natural numbers the operators $*$ and $+$ are coded in Lisp. Packing rules are provided for zero and for the successor function.

2. For hybrid lists the sort operator is coded in Lisp (using bubble sort).

3. A combination of these two specifications can be made by simply merging the rules. Note that in this combined version the two separate hybrid versions are still 'unaware' of each other.

The entire rewriting system contains 37 rules. A single input term is used in all cases. This term is normalized in 4836 reduction steps in the standard implementation.

| implementation | rewrite rules | time (sec) | n. of reductions | | n. of unpacks | |
|---|---|---|---|---|---|---|
| | | | total | hybrid | total | superfluous |
| standard | standard | 723 | 4836 | | | |
| hybrid | Lists | 738 | 4385 | 527 | | |
| | Numbers | 240 | 4527 | 85 | | |
| | both | 115 | 4007 | 612 | 8631 | 2509 |

*Figure 4.4.*

Some observations about these figures are:

- The total number of reduction steps is roughly the same in every version. The reason is that the input term was chosen such, that during its normalization no 'deep' terms occur. In this way we have attempted to create a fair balance between 'real' rewritings and packing and unpacking rewritings (it is easy to supply some term such as $1000 * 1000$, which will be normalized in 2000 steps in the hybrid case, but in 1000000 steps in the normal case.

- The execution time in the hybrid list situation is higher than that in the standard situation. There is a small overhead in the hybrid implementation. In our input term only a moderate use was made of list sorting. Consequently, the gain did not outweigh the price.

More extensive measurements using a larger example are presented in Chapter 5.

# Appendix A: A detailed example

In this section we will present a detailed example. We will first present an algebraic specification of Booleans, non-negative Integers modulo 10, and Lists of such integers. Then we will show how various function can be implemented using a cpl and we will discuss most functions and rule schemes, the existence of which was assumed in Section 4.7. Finally, we will present two Lisp implementations; the standard term rewriting implementation and our hybrid implementation.

Note that Lists are a parameterized data type, although they are presented here with one particular actual parameter.

## A.1 An example TRS

Boolean functions are $t$ (true), $f$ (false), $\vee$ (or), $\wedge$ (and), and $\neg$ (not); Integer functions are 0 (zero), $s$ (the successor function mapping 0 to 1, 1 to 2 etc.), $+$ (addition) and $*$ (multiplication); $zero$ is the predicate which determines if an Integer is equal to zero (which is needed because on Integers modulo 10 the syntactic appearance of a number cannot be used to check for equality to zero), and $\leq$ is the less-than-or-equal predicate on Integers; List functions are $\epsilon$ (the empty list), $|$ (putting an item at the head of a list) and $sort$ (which sorts a list of integers, using a merge-sort algorithm). Auxiliary functions are $odd$ and $even$, which return the sublist containing the entries occurring in odd and even positions respectively; $merge$ which merges two sorted sublists into a sorted list; and $if$, which selects one of two possibilities given a Boolean value.

The specification contains the following rewrite rules.

(1) $t \vee x = t$
(2) $f \vee x = x$
(3) $t \wedge x = x$
(4) $f \wedge x = f$
(5) $\neg t = f$
(6) $\neg f = t$

(7) $s(s(s(s(s(s(s(s(s(s(x)))))))))) = x$
(8) $x + 0 = x$
(9) $x + s(y) = s(x) + y$
(10) $x * 0 = 0$
(11) $x * s(y) = (x * y) + x$
(12a) $zero(0) = t$
(12b) $zero(s(0)) = f$
(12c) $zero(s(s(0))) = f$
(12d) $zero(s(s(s(0)))) = f$
(12e) $zero(s(s(s(s(0))))) = f$
(12f) $zero(s(s(s(s(s(0)))))) = f$
(12g) $zero(s(s(s(s(s(s(0))))))) = f$
(12h) $zero(s(s(s(s(s(s(s(0)))))))) = f$
(12i) $zero(s(s(s(s(s(s(s(s(0))))))))) = f$
(12j) $zero(s(s(s(s(s(s(s(s(s(0)))))))))) = f$
(13) $0 \leq x = t$
(14) $s(x) \leq 0 = zero(s(x))$
(15) $s(x) \leq s(y) = zero(s(x)) \vee if(zero(s(y)), f, x \leq y)$
(16) $if(t, l, m) = l$
(17) $if(f, l, m) = m$
(18) $odd(\epsilon) = \epsilon$
(19) $odd(x|l) = x|even(l)$
(20) $even(\epsilon) = \epsilon$
(21) $even(x|l) = odd(l)$
(22) $merge(\epsilon, l) = l$
(23) $merge(l, \epsilon) = l$
(24) $merge(x|l, y|m) = if(x \leq y,\ x|merge(l, y|m)\ ,\ y|merge(x|l, m))$
(25) $sort(\epsilon) = \epsilon$
(26) $sort(x|\epsilon) = x|\epsilon$
(27) $sort(x|y|l) = merge(sort(odd(x|y|l)), sort(even(x|y|l)))$

Observe that the specification for lists is effectively parameterized. It only assumes the existence of a $\leq$ predicate on parameter arguments.

The specification above may be viewed as a TRS. This TRS is confluent and Noetherian. The proof of confluence and termination of this TRS is beyond the scope of this chapter.

The functions we would like to implement using a cpl are $+$, $*$ and *sort*. In the non-parameterized implementation, $N$ will contain the terms 0 and $s(\ldots s(0)\ldots)$. In the parameterized implementation $N$ will contain $\epsilon$ and $x_1|\ldots|x_k|\epsilon$, where each $x_i$ is some integer variable. In the parameterized implementation $N$ may or may not contain the integers. Note that $N$ does contain terms which are not in normal form, due to equation (7).

The functions `internal`, `val` and `make` rely on a low level coding scheme, which we will investigate later in this section. The function unpack can be described as follows.

```
unpack(term):
   case val(term) of
     int n:  if n=0
             then exit(⟦0⟧)
             else with ⟦pred⟧=make(n-1)
                    exit(⟦s(pred)⟧)
     list l: if empty(l)
             then exit(⟦ε⟧)
             else with ⟦lst⟧=make(tail(l))
                       ⟦item⟧=head(l)
                    exit(⟦item|lst⟧)
```

Finally, we can present the various rule schemes. Below, variable names with a capital letter indicate limited variables. Generators are denoted as `sub` $\mapsto$ `body`, where `sub` is the parameter to the generator, which is always a substitution, and where `body` is the function body of the generator. In this example we will use a Prolog-like notation for Lists, where `[]` is the empty list, and where `[` *head* `|` *tail* `]` is the list with *head* as first element, and *tail* the list with remaining elements.

**The non-parameterized situation**
Rule schemes in $R^{\prec}$.

$0 =$ `sub` $\mapsto$ `exit(make(0))`
$s(X) =$ `sub` $\mapsto$ `exit(make(val(sub(`$X$`))+1))`

Rule schemes in $R^{cpl}$.

$X + Y =$ `sub` $\mapsto$ `exit(make(val(sub(`$X$`))+val(sub(`$Y$`))))`
$X * Y =$ `sub` $\mapsto$ `exit(make(val(sub(`$X$`))*val(sub(`$Y$`))))`

**The parameterized situation**
Rule schemes in $R^{\prec}$.

$\epsilon =$ `sub` $\mapsto$ `exit(make([]))`
$x|L =$ `sub` $\mapsto$ `exit(make([ x | val(sub(`$L$`)) ]))`

Rule schemes in $R^{cpl}$.

```
sort(L) = sub ↦
   with l=val(sub(L))
        [x_1... x_n]=l
      for i=1 to n-1
         for j=i+1 to n
            if p(⟦x_i ≤ x_j⟧, t)
            then swap(x_i,x_j)
   exit(make(l))
```

Observe that the *sort* generator uses a bubble sort algorithm.

## A.2 Term rewriting implementation of algebraic specifications

In this section we will present a standard TRS implementation coded in Lisp. The Lisp version used is Le-Lisp version 15.22 by INRIA [25]. The implementation below has two parts. The first part contains the general implementation, which has a set of rules as parameter. The second part contains the lisp version of the reduction rules in the TRS presented earlier. The Lisp code below is very similar to the algorithmic code shown before.

```
; The following conventions are used:
;    variable       - symbol
;    function symb. - symbol
;    term           - struct *trm (ofs (outerm f symb) and args)
;    sub            - list of (key . value) pairs
;                     or 'empty-sub, since nil indicates failure
;    rules          - list of structs *rle (lhs, rhs and unp)


(de match (pat term sub)
  (let (x)
       (cond
        ((symbolp pat)
         (if (setq x (cassq pat sub))
             (if (equal x term)
                 sub)
             (acons pat term (consp sub))
         ))
        ((typep term '*trm)
         (if (eq (#:*trm:ofs pat) (#:*trm:ofs term))
             (if (#:*trm:args pat)
                 (every
                   (lambda (pt tm)
                     (setq sub (match pt tm sub)))
                   (#:*trm:args pat)
                   (#:*trm:args term))
                 sub))))))

(de inst (pat sub)
  (cond
   ((symbolp pat)
    (cassq pat sub))
   ((typep pat '*trm)
    (*trm (#:*trm:ofs pat)
      (mapcar (lambda (tp) (inst tp sub))
          (#:*trm:args pat))))))

(de ff (term rules)
  (or
   (if (typep term '*trm)
```

```
      (let* ((exit ())
             (trms (mapcar (lambda (tp)
                              (if exit
                                  tp
                                  (or (setq exit (ff tp rules)) tp)))
                           (#:*trm:args term))))
         (if exit
             (*trm (#:*trm:ofs term) trms))))
   (any (lambda (rule)
          (let ((sub (match (#:*rle:lhs rule)
                            term
                            'empty-sub)))
             (if sub (inst (#:*rle:rhs rule) sub))))
        rules)))

(de f_omg (term rules)
  (let ((tmp (ff term rules)))
     (if tmp
         (f_omg tmp rules)
         term)))

(de p (term1 term2 rules)
  (equal (f_omg term1 rules)
     (f_omg term2 rules)))

(defstruct *trm ofs args)
(defstruct *rle lhs rhs)

(de *trm (ofs args)
  (let ((tmp (new '*trm)))
       (#:*trm:ofs tmp ofs)
       (#:*trm:args tmp args)
       tmp))
(de *rle (lhs rhs)
  (let ((tmp (new '*rle)))
       (#:*rle:lhs tmp lhs)
       (#:*rle:rhs tmp rhs)
       tmp))
```

Below is a function call, which when evaluated yields the datastructure that can be passed to the implementation above.

```
(*rules
`(
  ( (or (t) x)   (t) )
  ( (or (f) x)   x )
  ( (and (t) x)   x )
  ( (and (f) x)   (f) )
  ( (not (t))   (f) )
```

```
( (not (f))   (t) )
( (s (s (s (s (s (s (s (s (s (s x))))))))))   x )
( (+ x (0))   x )
( (+ x (s y))   (+ (s x) y) )
( (* x (0))   (0) )
( (* x (s y))   (+ (* x y) x) )
( (zero (0))   (t) )
( (zero (s (0)))   (f) )
( (zero (s (s (0))))   (f) )
( (zero (s (s (s (0)))))   (f) )
( (zero (s (s (s (s (0))))))   (f) )
( (zero (s (s (s (s (s (0)))))))   (f) )
( (zero (s (s (s (s (s (s (0))))))))   (f) )
( (zero (s (s (s (s (s (s (s (0)))))))))   (f) )
( (zero (s (s (s (s (s (s (s (s (0))))))))))   (f) )
( (zero (s (s (s (s (s (s (s (s (s (0)))))))))))   (f) )
( (<= (0) x)   (t) )
( (<= (s x) (0))   (zero (s x)) )
( (<= (s x) (s y))  (or (zero (s x)) (if (zero (s y)) (f) (<= x y))) )
( (if (t) l m)   l )
( (if (f) l m)   m )
( (odd (empty))   (empty) )
( (odd (ap x l))   (ap x (even l)) )
( (even (empty))   (empty) )
( (even (ap x l))   (odd l) )
( (merge (empty) l)   l )
( (merge l (empty))   l )
( (merge (ap x l) (ap y m))   (if (<= x y)
                                   (ap x (merge l (ap y m)))
                                   (ap y (merge (ap x l) m))) )
( (sort (empty))   (empty) )
( (sort (ap x (empty)))   (ap x (empty)) )
( (sort (ap x (ap y l)))   (merge (sort (odd (ap x (ap y l))))
            (sort (even (ap x (ap y l))))) )
))
```

## A.3 Hybrid implementation of algebraic specifications

In this section we will present an implementation coded in Lisp which is based on the theory presented in this chapter. As before, the Lisp version used is Le-Lisp version 15.22 by INRIA [25]. The first part contains the general implementation, which is very similar to the algorithmic code presented earlier, to which we refer for comment. The second part contains the actual rules and rule schemes as presented earlier.

The only essential difference with the term rewriting implementation is the function `unpack`. Earlier, a single function `unpack` was defined, which handled the two sorts `integer` and `list`. In an actual implementation this is undesirable, because it is not modular. Below, an `unpack` function exists for every sort which has internal representation, and every internal value contains a reference to its own unpack function.

```
; The following conventions are used:
; variable          - symbol
; function symb.    - symbol
; term              - struct *trm (ofs (outerm f symb) and args)
; sub               - list of (key . value) pairs
;                     or 'empty-sub, since nil indicates failure
; rules             - list of structs *rle (lhs, rhs and unp)
; limited var's     - struct *lim (symb)
; generators        - struct *gen (fun)
; internal values   - (*int), a constant with non-symbol function,
;                     which is a struct *int (val: the actual internal
;                     value, unp: a function which unpacks this value,
;                     and ofs: the outermost function symbol of
;                     the unpacked term.

(de match (pat term sub unp)
  (let (x)
       (cond
tree reptypep pat '*lim)
      (if (typep term '*int)
          (acons pat term (consp sub))))
   ((symbolp pat)
      (if (setq x (cassq pat sub))
          (if (equal x term)
              sub)
          (acons pat term (consp sub))))
   ((typep term '*int)
      (and (not unp)
           (eq (#:*int:ofs term) (#:*trm:ofs pat))
           (match pat (unpack term) sub unp)))
   ((typep term '*trm)
      (if (eq (#:*trm:ofs pat) (#:*trm:ofs term))
          (if (#:*trm:args pat)
            (every
              (lambda (pt tm)
                (setq sub (match pt tm sub unp)))
              (#:*trm:args pat)
              (#:*trm:args term))
           sub)))))))

(de inst (pat sub)
  (cond
   ((symbolp pat)
      (cassq pat sub))
   ((typep pat '*gen)
      (funcall (#:*gen:fun pat) sub))
   ((typep pat '*trm)
      (*trm (#:*trm:ofs pat)
```

```
        (mapcar (lambda (tp) (inst tp sub))
           (#:*trm:args pat))))))

(de ff (term rules)
  (or
   (if (typep term '*trm)
       (let* ((exit ())
              (trms (mapcar (lambda (tp)
                              (if exit
                                  tp
                                  (or (setq exit (ff tp rules)) tp)))
                          (#:*trm:args term))))
          (if exit
              (*trm (#:*trm:ofs term) trms))))
   (any (lambda (rule)
           (let ((sub (match (#:*rle:lhs rule)
                             term
                             'empty-sub
                             (#:*rle:unp rule))))
             (if sub (inst (#:*rle:rhs rule) sub))))
        rules)))

(de f_omg (term rules)
  (let ((tmp (ff term rules)))
     (if tmp
         (f_omg tmp rules)
         term)))

(de p (term1 term2 rules)
  (equal (f_omg term1 rules)
         (f_omg term2 rules)))


(de unpack (internal)
tree repncall (#:*int:unp internal)
           (#:*int:ofs internal)
           (#:*int:val internal)))


(defstruct *trm ofs args)
(defstruct *rle lhs rhs unp)
(defstruct *lim sym)
(defstruct *gen fun)
(defstruct *int ofs val unp)

(de *trm (ofs args)
  (let ((tmp (new '*trm)))
       (#:*trm:ofs tmp ofs)
       (#:*trm:args tmp args)
```

```
       tmp))
(de *rle (lhs rhs unp)
  (let ((tmp (new '*rle)))
       (#:*rle:lhs tmp lhs)
       (#:*rle:rhs tmp rhs)
       (#:*rle:unp tmp unp)
       tmp))
(de *lim (sym)
  (let ((tmp (new '*lim)))
       (#:*lim:sym tmp sym)
       tmp))
(de *gen (fun)
  (let ((tmp (new '*gen)))
       (#:*gen:fun tmp fun)
       tmp))
(de *int (ofs val unp)
  (let ((tmp (new '*int)))
       (#:*int:ofs tmp ofs)
       (#:*int:val tmp val)
       (#:*int:unp tmp unp)
       tmp))

(de *rules(rules)
    (if rules
        (cons
           (*rule(car rules))
           (*rules(cdr rules)))))
(de *rule(rule)
    (*rle (*term(car rule))
          (*term(cadr rule))
          (caddr rule) ))
(de *term(term)
    (if (consp term)
        (*trm (car term) (*terms(cdr term)))
        term))
(de *terms(tl)
    (if tl
        (cons (*term(car tl))
              (*terms(cdr tl)))))

(de int-unpack (ofs val)
  (selectq ofs
      ('0   (*trm '0 ()))
      ('s   (let ((nval (1- val)))
            (*trm 's (list (*int (if (eq nval 0) '0 's)
                  nval
                  'int-unpack)))))))
(*rules
`(
```

```
( (or (t) x)   (t) )
( (or (f) x)   x )
( (and (t) x)   x )
( (and (f) x)   (f) )
( (not (t))   (f) )
( (not (f))   (t) )
( (0)
  ,(*gen (lambda (sub) (*int '0 0 'int-unpack)))
  **no-unpack)
( (s ,(*lim 'x))
  ,(*gen `(lambda (sub)
    (*int 's
          (1+ (#:*int:val (cassoc ',(*lim 'x) sub)))
          'int-unpack)))
  **no-unpack)
( (s (s (s (s (s (s (s (s (s (s x))))))))))   x )
( (+ ,(*lim 'x) ,(*lim 'y))
  ,(*gen `(lambda (sub)
            (let ((sum (+ (#:*int:val (cassoc ',(*lim 'x) sub))
                          (#:*int:val (cassoc ',(*lim 'y) sub)))))
               (*int (if (eq sum 0) '0 's)
                     sum
                     'int-unpack)))) )
( (+ x (0))   x )
( (+ x (s y))   (+ (s x) y) )
( (* ,(*lim 'x) ,(*lim 'y))
  ,(*gen `(lambda (sub)
            (let ((prod (* (#:*int:val (cassoc ',(*lim 'x) sub))
                           (#:*int:val (cassoc ',(*lim 'y) sub)))))
               (*int (if (eq prod 0) '0 's)
                     prod
                     'int-unpack)))) )
( (* x (0))   (0) )
( (* x (s y))   (+ (* x y) x) )
( (zero (0))   (t) )
( (zero (s (0)))   (f) )
( (zero (s (s (0))))   (f) )
( (zero (s (s (s (0)))))   (f) )
( (zero (s (s (s (s (0))))))   (f) )
( (zero (s (s (s (s (s (0)))))))   (f) )
( (zero (s (s (s (s (s (s (0))))))))   (f) )
( (zero (s (s (s (s (s (s (s (0)))))))))   (f) )
( (zero (s (s (s (s (s (s (s (s (0))))))))))   (f) )
( (zero (s (s (s (s (s (s (s (s (s (0)))))))))))   (f) )
( (<= (0) x)   (t) )
( (<= (s x) (0))   (zero (s x)) )
( (<= (s x) (s y))  (or (zero (s x)) (if (zero (s y)) (f) (<= x y))) )
( (if (t) l m)   l )
( (if (f) l m)   m )
```

```
  ( (odd (empty))   (empty) )
  ( (odd (ap x l))   (ap x (even l)) )
  ( (even (empty))   (empty) )
  ( (even (ap x l))   (odd l) )
  ( (merge (empty) l)   l )
  ( (merge l (empty))   l )
  ( (merge (ap x l) (ap y m))   (if (<= x y)
                                    (ap x (merge l (ap y m)))
                                    (ap y (merge (ap x l) m))) )
  ( (sort (empty))   (empty) )
  ( (sort (ap x (empty)))   (ap x (empty)) )
  ( (sort (ap x (ap y l)))   (merge (sort (odd (ap x (ap y l))))
                                    (sort (even (ap x (ap y l))))))
  ))
```

## A.4 Hybrid implementation of parameterized algebraic specifications

In this section we will present the unpack function for lists and the rewrite rules and rule schemes for the parameterized implementation. As mentioned before, the general implementation in the parameterized case is identical to the non-parameterized version presented in the previous section, and will not be shown here.

```
(de list-unpack (ofs val)
  (selectq ofs
      ('empty   (*trm 'empty ()))
      ('ap   (let ((nval (cdr val)))
                (*trm 'ap (list (car val)
                                (*int (if nval 'ap 'empty)
                                      nval
                                      'list-unpack)))))))
(*rules
`(
  ( (or (t) x)   (t) )
  ( (or (f) x)   x )
  ( (and (t) x)   x )
  ( (and (f) x)   (f) )
  ( (not (t))   (f) )
  ( (not (f))   (t) )
  ( (s (s (s (s (s (s (s (s (s (s x))))))))))   x )
  ( (+ x (0))   x )
  ( (+ x (s y))   (+ (s x) y) )
  ( (* x (0))   (0) )
  ( (* x (s y))   (+ (* x y) x) )
  ( (zero (0))   (t) )
  ( (zero (s (0)))   (f) )
  ( (zero (s (s (0))))   (f) )
```

```
( (zero (s (s (s (0)))))   (f) )
( (zero (s (s (s (s (0))))))   (f) )
( (zero (s (s (s (s (s (0)))))))   (f) )
( (zero (s (s (s (s (s (s (0))))))))   (f) )
( (zero (s (s (s (s (s (s (s (0)))))))))   (f) )
( (zero (s (s (s (s (s (s (s (s (0))))))))))   (f) )
( (zero (s (s (s (s (s (s (s (s (s (0)))))))))))   (f) )
( (<= (0) x)   (t) )
( (<= (s x) (0))   (zero (s x)) )
( (<= (s x) (s y))  (or (zero (s x)) (if (zero (s y)) (f) (<= x y))) )
( (empty)
  ,(*gen (lambda (sub) (*int 'empty () 'list-unpack)))
  **no-unpack)
( (ap x ,(*lim 'y))
  ,(*gen `(lambda (sub)
      (*int 'ap
            (cons (cassoc 'x sub)
                  (#:*int:val (cassoc ',(*lim 'y) sub)))
            'list-unpack)))
  **no-unpack)
( (if (t) l m)   l )
( (if (f) l m)   m )
( (odd (empty))   (empty) )
( (odd (ap x l))   (ap x (even l)) )
( (even (empty))   (empty) )
( (even (ap x l))   (odd l) )
( (merge (empty) l)   l )
( (merge l (empty))   l )
( (merge (ap x l) (ap y m))   (if (<= x y)
                                  (ap x (merge l (ap y m)))
                                  (ap y (merge (ap x l) m))) )
( (sort ,(*lim 'x))
  ,(*gen `(lambda (sub)
             (let ((sorted (sort (lambda (x y)
                                   (p (*trm 't ())
                                      (*trm '<= (list x y))
                                      rules))
                              (#:*int:val (cassoc ',(*lim 'x) sub)))))
                (*int (if sorted 'ap 'empty)
                      sorted
                      'list-unpack)))) )
( (sort (empty))   (empty) )
( (sort (ap x (empty)))   (ap x (empty)) )
( (sort (ap x (ap y l)))   (merge (sort (odd (ap x (ap y l))))
                                  (sort (even (ap x (ap y l))))) )
))
```

# Appendix B: Miscellaneous results

In this section various proofs skipped earlier are presented.

## B.1 $\preceq$ is a partial order

The relation $\preceq$ is a partial relation as can be seen as follows.

- $\forall t{:}\, t \preceq t$ follows immediately; clause (1) is always applicable.
- $\forall s, t{:}\, s \preceq t \wedge t \preceq s \Rightarrow s = t$: Observe that for any constant $c \in \Sigma^+$ we have $t \preceq c \rightarrow t = c$ as is easily verified. Now assume that $s \neq t$ and let $s_0$ be the leftmost outermost subterm of $s$, whose outermost function symbol doesn't match the corresponding symbol in $t$ (heading a subterm which we will call $t_0$). If $s \preceq t$ then $s_0 \in \Sigma^+$, and if $t \preceq s$ then $t_0 \in \Sigma^+$. We may conclude that $s_0 = t_0$, which is a contradiction.
- $\forall s, t, u{:}\, s \preceq t \wedge t \preceq u \Rightarrow s \preceq u$: Let $u$ be a smallest (in depth) term for which such an $s$ and $t$ can be found, that $s \preceq t$ and $t \preceq u$ but $s \not\preceq u$. Apparently the outermost symbol of $t$ differs from that of $s$ or that of $u$. If it differs from that of $u$ however, we have that $t \in \Sigma^+$, which, together with the observation in the previous proof, leads to a contradiction. Hence, $s = c_{f(s_1,\ldots,s_n)}$ and $f(c_{s_1}, \ldots, c_{s_n}) \preceq t$. The outermost function symbols of $t$ and $u$ are both $f$. Now however, we have that $f(c_{s_1}, \ldots, c_{s_n}) \preceq t$ and $t \preceq u$, and apparently $f(c_{s_1}, \ldots, c_{s_n}) \not\preceq u$. Since $t$ and $u$ have the same outermost function symbol, there must be an index $i$ for which immediate subterms $t_i$ and $u_i$ of $t$ and $u$, respectively, exist such that $c_{s_i} \preceq t_i$ and $t_i \preceq u_i$ and $c_{s_i} \not\preceq u_i$. This is in contradiction with the assumption that $u$ has minimal depth.

## B.2 Every term has a unique $\preceq$ infimum and supremum

For every $\Sigma^*$-term there is exactly one $\preceq$-supremum (which is in $T_\Sigma$) and exactly one $\preceq$-infimum (in $T_{\Sigma^*}$).

**22. Theorem**

$$\forall s \in T_{\Sigma^*} \exists! t \in T_{\Sigma^*}: s \preceq t \text{ and } \forall s' \in T_{\Sigma^*}: s \preceq s' \Rightarrow s' \preceq t$$

and for this $t$ we have $t \in T_\Sigma$.

**23. Theorem**

$$\forall s \in T_{\Sigma^*} \exists! t \in T_{\Sigma^*}: t \preceq s \text{ and } \forall t' \in T_{\Sigma^*}: t' \preceq s \Rightarrow t \preceq t'$$

We will denote the $\preceq$-supremum of a term $s$ by $\lceil s \rceil$ and the $\preceq$-infimum by $\lfloor s \rfloor$.

For any $s \in T_{\Sigma^*}$ the term $\lfloor s \rfloor$ is the most internal representation of $s$ and $\lceil s \rceil$ is the term representation, it is the unique term in $T_\Sigma$ of which $s$ (and $\lfloor s \rfloor$) are alternative representations.

In order to prove Theorem 22, we will first define $\lceil . \rceil$.
Let

1. $\lceil c_s \rceil = s$ for $c_s \in \Sigma^+$;
2. $\lceil f(s_1, \ldots, s_n) \rceil = f(\lceil s_1 \rceil, \ldots, \lceil s_n \rceil)$ for $f \in \Sigma$.

Observe that $\lceil s \rceil \in T_\Sigma$ for any $s \in T_{\Sigma^*}$ and that for $s \in T_\Sigma$ we have $\lceil s \rceil = s$ (and consequently $s \preceq \lceil s \rceil$). We will now show that for all $s$: $s \preceq \lceil s \rceil$.

- $\forall c_s \in \Sigma^+ \colon c_s \preceq \lceil c_s \rceil$. Proof by induction on the complexity of $s$. If $s$ is a constant, clause 2 of Definition 7, and the observation above are applicable. If $s = f(s_1, \ldots, s_n)$, then clause 2, Definition 7 is applicable precisely if $f(c_{s_1}, \ldots, c_{s_n}) \preceq f(s_1, \ldots, s_n)$, which is a consequence of our induction hypothesis.
- $\forall f \in \Sigma\ \forall f(s_1, \ldots, s_n) \in T_{\Sigma^*} \colon f(s_1, \ldots, s_n) \preceq \lceil f(s_1, \ldots, s_n) \rceil$. We will use induction on the complexities of $s_1$ to $s_n$. Clearly $f(s_1, \ldots, s_n) \preceq f(\lceil s_1 \rceil, \ldots, \lceil s_n \rceil)$, as a consequence of our induction hypothesis, or of the property derived above. The desired equation now follows immediately.

Now, in order to prove Theorem 22, we only need to show that for any term $s$, $\lceil s \rceil$ is the only term in $T_{\Sigma^*}$ which is a $\preceq$-supremum of $s$. Suppose that $s \in T_{\Sigma^*}$ and that $s \preceq t$ for some $t \in T_{\Sigma^*}$. We will use induction on the complexity of $t$ to show that $t \preceq \lceil s \rceil$. If $t$ is a constant, then either $s = t$ (clause 1) or $s = c_t$ (clause 2). In both cases $t = \lceil s \rceil$. If $t = f(t_1, \ldots, t_n)$, then either

- $s = f(s_1, \ldots, s_n)$ and $s_1 \preceq t_1, \ldots, s_n \preceq t_n$. According to the induction hypothesis we have $t_1 \preceq \lceil s_1 \rceil, \ldots, t_n \preceq \lceil s_n \rceil$, and we conclude that $t \preceq f(\lceil s_1 \rceil, \ldots, \lceil s_n \rceil) = \lceil s \rceil$;
- $s = c_{g(s_1, \ldots, s_k)}$ and $g(c_{s_1}, \ldots, c_{s_k}) \preceq f(t_1, \ldots, t_n)$, which can only be satisfied if $f = g$ (and $n = k$). But then we are in the same situation as in the previous case.

This concludes the proof of Theorem 22.
Theorem 23 is shown likewise. We start with the definition of $\lfloor . \rfloor$:

1. $\lfloor s \rfloor = c_s$ for $s \in N$;
2. $\lfloor f(s_1, \ldots, s_n) \rfloor = f(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor)$ for $f(s_1, \ldots, s_n) \in T_{\Sigma^*} \setminus N$, where $n \geq 0$.

Note that since $n \geq 0$, all constants are handled.
First we will show that for all $s$ we have $\lfloor s \rfloor \preceq s$. We will use induction on the complexity of $s$. If $s$ is a constant, then $\lfloor s \rfloor = s$ ($s \notin N$) or $\lfloor s \rfloor = c_s \preceq s$ (by clause 2).
If $s = f(s_1, \ldots, s_n) \notin N$, then $\lfloor s \rfloor = f(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor) \preceq f(s_1, \ldots, s_n) = s$ (applying clause 1 and using the induction hypothesis). If $s = f(s_1, \ldots, s_n) \in$

$N$, then $\lfloor s \rfloor = c_s$. Clause 2 may be applied precisely if $f(c_{s_1}, \ldots, c_{s_n}) \preceq f(s_1, \ldots, s_n)$ (*). Now we can use the fact that $N$ is closed under subterm selection, as a consequence of which $\lfloor s_i \rfloor = c_{s_i}$ for each $i$. We can now apply clause 1 and the induction hypothesis to establish fact (*).

Now suppose $t \preceq s$ for some $s, t \in T_{\Sigma^*}$. Using induction on the complexity of $s$, we will show that $\lfloor s \rfloor \preceq t$. If $s$ is a constant either $s \in \Sigma^+$, in which case only clause 1 can be applicable, hence $s = t$, or $s \in T_\Sigma$, in which case $t = s$ or $t = c_s \preceq s$. If $s = f(s_1, \ldots, s_n)$ then either

- $t = f(t_1, \ldots, t_n)$ and $t_i \preceq s_i$ for each $i$, but then $\lfloor s_i \rfloor \preceq t_i$ (induction hypothesis) and hence $\lfloor s \rfloor \preceq t$;

- $t = c_{g(t_1,\ldots,t_k)}$ and $g(t_1, \ldots, t_k) \preceq f(s_1, \ldots, s_n)$, but then $g = f$ (and $k = n$) and $t_i \preceq s_i$, and hence, using the induction hypothesis, $\lfloor s \rfloor \preceq t$.

We have now established Theorem 23.

## B.3 An implementation of $\prec$-matching

In order to recognize a term $u \prec C[s_0^\sigma]$ as a ($\prec$-)instantiation of $C[s_0]$, we have to 'unwrap' $u$ until it is an instantiation. This unwrapping is done by applying the reverse of the $\prec$-rules on internal representations in $u$. The reverse rules cannot simply be added to the TRS as rewrite rules, because cycles would then be introduced, but we can temporarily apply the reverse of these rules during matching. Performing this reverse may seem a waste of work already done, but usually it is not; internal representations are in many cases the result of calculations done by cpl functions, and the gain in efficiency achieved by using cpl functions outweighs this loss.

If $C[s_0]$ is known, we can simply match $u$ with $C[s_0]$ in the traditional way, until an internal representation occurs in $u$. This internal representation is then converted, using a reverse $\prec$-rule, to a less internal representation, and matching can continue as before.

In our situation however, $C[s_0]$ is not known, because we are looking for an arbitrary redex in an arbitrary context. Now, we do not know with which internal representation in $u$ to begin.

In a sense the problem arises because terms containing redexes can be put in internal representation. Otherwise, no subterm in $C[s_0^\sigma]$ containing $s_0^\sigma$ would have been put in internal form, and we would be able to recognize the instantiation of $s_0$ without unwrapping a single function symbol in $C[.]$.

In our implementations we will assume that an internal representation (in $\Sigma^+$), the term representation of which is not itself (at the root) a redex, does not contain a redex either. Observe that innermost strategies have this property automatically, because immediate subterms are normalized beforehand, and consequently a $\prec$-rule will only be applied if the arguments are in normal form. Non-innermost strategies have to verify that the immediate subterms of term are in normal form, before a $\prec$-rule is applied, in order to achieve this property.

Assuming this property we can, as before, match the left-hand side of a rewrite rule with each subterm of a term. When an internal representation is matched with a non-variable, the internal representation is unwrapped once, temporarily, and matching continues. Matching an internal representation with a variable is dealt with in the usual fashion. If no match is found in a term, it is in normal form. If a term is not a normal form, then a match ($\prec$-matching) will be found on one of the subterms.

Note that an implementation based on this property is not correct for arbitrary terms in $T_{\Sigma^*}$. It is only correct when we start with a term in $T_\Sigma$ and rewrite it using strategies which do not create internal representation with 'contained' redexes. As mentioned before, the theoretical construction neither assumes nor uses this property.

## B.4 Equivalence of strategies and superstrategies

**24. Lemma**

$$\forall s,t \in T_{\Sigma^*}\colon s \searrow t \Rightarrow \lceil s \rceil \xrightarrow{*} \lceil t \rceil$$

Apparently there are a context $C[.]$, a substitution $\sigma$ ranging over $T_{\Sigma^*}$ and a rule $s_0 \to t_0$ in $R^*$, such that $s \preceq C[s_0^\sigma]$ (or $s = C[s_0^\sigma]$ for $s_0 \to t_0 \in R^{\prec}$) and $t = C[t_0^\sigma]$. Now define the context $C'[.] = \lceil C[.] \rceil$ and the substitution $\sigma' = \lceil \sigma \rceil$ (where $\lceil . \rceil$ is extended for substitutions in the obvious manner).

Suppose $s_0 \to t_0 \in R^{\prec}$. Then $s_0 = f(c_{s_1},\ldots,c_{s_n})$ and $t_0 = c_{f(s_1,\ldots,s_n)}$ for some $f(s_1,\ldots,s_n) \in N$. Now we have $\lceil s \rceil = \lceil C[f(c_{s_1},\ldots,c_{s_n})] \rceil = C'[f(s_1,\ldots,s_n)] = C'[\lceil t_0 \rceil]$, and consequently an (empty) reduction sequence as required has been established.

Now suppose $s_0 \to t_0 \in R$. Then $\lceil s \rceil = C'[\lceil s_0^\sigma \rceil] = C'[s_0^{\sigma'}]$ and $\lceil t \rceil = C'[t_0^{\sigma'}]$, and we have established $\lceil s \rceil \to \lceil t \rceil$.

Finally suppose $s_0 \to t_0 \in R^{cpl}$. Then $s_0$ is of the form $f(c_{s_1},\ldots,c_{s_n})$ and $t_0$ is of the form $c_{t_1}$, and a reduction sequence $f(s_1,\ldots,s_n) \to \ldots \to t_1$ exists. We now have $\lceil s \rceil = C'[f(s_1,\ldots,s_n)]$ and $\lceil t \rceil = C'[t_1]$ and we have established the desired reduction sequence.

**25. Lemma**

$$\forall s,t \in T_{\Sigma^*}\colon s \preceq t \Rightarrow t \overset{*}{\searrow} s$$

We will prove this lemma using induction on the complexity of $t$. If $t$ is a constant, then either $s = t$ or $s = c_t$. If $s = t$ a reduction sequence of length 0 has been established. If $s = c_t$, then $t \in N$ (otherwise $c_t$ would not exist) and hence the rule $t \to c_t$ is in $R^{\prec}$. A reduction sequence of length 1 has been established.

If $t = f(t_1,\ldots,t_n)$, then either $s = f(s_1,\ldots,s_n)$ and for each $i$ we have $s_i \preceq t_i$, or $s = c_{f(\lceil t_1 \rceil,\ldots,\lceil t_n \rceil)}$. In the first case according to the induction hypothesis, a reduction sequence exists for each $i$ from $t_i$ to $s_i$. Any merge of these sequences defines a reduction sequence from $t$ to $s$ as required. In the second case the rule $f(c_{\lceil t_1 \rceil}, \ldots, c_{\lceil t_n \rceil}) \to s$ is in $R^{\prec}$. Now, a reduction sequence as required exists precisely when for each $i$ such a sequence exists from $t_i$ to $c_{\lceil t_i \rceil}$. This however, is a consequence of our induction hypothesis, because clearly $c_{\lceil t_i \rceil} \preceq t_i$.

**26. Lemma**

$$\forall s \in T_{\Sigma^*}, t \in T_{\Sigma}\colon \lceil s \rceil \to t \Rightarrow s \searrow^{+} \lfloor t \rfloor$$

Apparently there is a context $C[.]$, a rule $s_0 \to t_0 \in R$ and a substitution $\sigma$, such that $\lceil s \rceil = C[s_0^\sigma]$ and $t = C[t_0^\sigma]$.

Clearly $\lfloor s \rfloor \preceq C[s_0^\sigma]$, and by Lemma 25 there is a reduction sequence $C[s_0^\sigma] \searrow^{*} \lfloor s \rfloor$. Furthermore $\lfloor s \rfloor \preceq \lceil s \rceil$, so by definition 10 we conclude that $\lfloor s \rfloor \searrow C[t_0^\sigma]$. Again by Lemma 25 we have $C[t_0^\sigma] \searrow^{*} \lfloor t \rfloor$. We have now established a reduction sequence $s \searrow^{+} \lfloor t \rfloor$.

**27. Lemma**

$$\forall s, t \in T_{\Sigma^*}\colon s \searrow t \Rightarrow \lfloor s \rfloor \searrow^{*} \lfloor t \rfloor$$

According to Lemma 24 we have $\lceil s \rceil \xrightarrow{*} \lceil t \rceil$. We will use induction on the length of that derivation. If $\lceil s \rceil = \lceil t \rceil$ then clearly $\lfloor s \rfloor = \lfloor t \rfloor$ and consequently $\lfloor s \rfloor \searrow^{*} \lfloor t \rfloor$.

If $\lceil s \rceil \xrightarrow{+} \lceil t \rceil$ there must be a $u \in T_{\Sigma}$, such that $\lceil s \rceil \to u$ and $u \xrightarrow{*} \lceil t \rceil$. Consequently we have $\lceil \lfloor s \rfloor \rceil \to \lceil \lfloor u \rfloor \rceil$ and $\lceil \lfloor u \rfloor \rceil \xrightarrow{*} \lceil t \rceil$. By Lemma26 we now have $\lfloor s \rfloor \searrow^{+} \lfloor u \rfloor$ and by our induction hypothesis we also have $\lfloor u \rfloor \searrow^{*} \lfloor t \rfloor$, and we conclude $\lfloor s \rfloor \searrow^{*} \lfloor t \rfloor$.

**28. Lemma**

$$\forall s, t \in T_{\Sigma^*}\colon s \searrow^{*} t \Rightarrow \lfloor s \rfloor \searrow^{*} \lfloor t \rfloor$$

Using induction on the length of the reduction sequence $s \searrow^{*} t$, this is an immediate consequence of Lemma 27.

**29. Lemma**

$$\forall s \in T_{\Sigma^*}, t \in T_{\Sigma}\colon \lceil s \rceil \xrightarrow{*} t \Rightarrow s \searrow^{*} \lfloor t \rfloor$$

We will use induction on the length of the reduction sequence $\lceil s \rceil \xrightarrow{*} t$. If $\lceil s \rceil = t$ then $s \searrow^{*} \lfloor t \rfloor$ follows from Lemma 25.

Otherwise a $u \in T_{\Sigma}$ exists, such that $\lceil s \rceil \to u$ and $u \xrightarrow{*} t$. By Lemma 26 we have $s \searrow^{+} \lfloor u \rfloor$ which implies $s \searrow^{*} \lfloor u \rfloor$. Since $\lceil u \rceil = u$ we have $u \searrow^{*} \lfloor t \rfloor$ by our induction hypothesis, and by Lemma 28 we have $\lfloor u \rfloor \searrow^{*} \lfloor t \rfloor$ and we have established $s \searrow^{*} \lfloor t \rfloor$.

**30. Lemma**

$$\forall t \in T_{\Sigma}\colon \{s | s \prec t\} \text{ is finite.}$$

From Definition 7 we see that for $s \prec t$, all function symbols in $s$ are equal to the corresponding symbol in $t$, apart from the constants from $\Sigma^{+}$. These constants are $\preceq$ than the corresponding subterm, and according to Theorem 23 they are uniquely defined. Since $t$ has only finitely many subterms we have established this lemma.

**31. Lemma**

$$\forall s \in T_{\Sigma^*} \colon R^{\prec} \vdash s = \lceil s \rceil$$

This is a direct consequence of Lemma 25 because in its proof only $\prec$-rules are used.

**12. Theorem**
If $(\Sigma, R)$ is confluent w.r.t. $\rightarrow$, then so is $(\Sigma^*, R^*)$ w.r.t. $\searrow$.

Let $s$, $t_1$ and $t_2$ be such, that $s \overset{*}{\searrow} t_1$ and $s \overset{*}{\searrow} t_2$. According to Lemma 24 we have $\lceil s \rceil \xrightarrow{*} \lceil t_1 \rceil$ and $\lceil s \rceil \xrightarrow{*} \lceil t_2 \rceil$. Consequently a $u \in T_\Sigma$ exists, with reduction sequences $\lceil t_1 \rceil \xrightarrow{*} u$ and $\lceil t_2 \rceil \xrightarrow{*} u$. By Lemma 29 we have $t_1 \overset{*}{\searrow} \lfloor u \rfloor$ and $t_2 \overset{*}{\searrow} \lfloor u \rfloor$ and we have established Theorem 12.

**13. Theorem**
If $(\Sigma, R)$ is Noetherian w.r.t. $\rightarrow$, then so is $(\Sigma^*, R^*)$ w.r.t. $\searrow$.

Suppose an infinite reduction sequence $s_1 \searrow s_2 \searrow \ldots$ exists. Then, according to Lemma 24 infinitely many reduction sequences $\lceil s_i \rceil \xrightarrow{*} \lceil s_{i+1} \rceil$ exist. Since $(\Sigma, R)$ is Noetherian w.r.t. $\rightarrow$, only finitely many of these can be non-empty and consequently an index $i$ must exists after which all reduction sequences $\lceil s_i \rceil \xrightarrow{*} \lceil s_{i+1} \rceil$ are empty. From the proof of Lemma 24 we can see that this implies that for each $j > i$ we have $s_j \prec s_i$. According to Lemma 30 however, there can be only finitely many such $s_j$.

**32. Theorem**
Let $(\Sigma, E)$ be an algebraic specification, $(\Sigma, R)$ the corresponding TRS and $(\Sigma^*, R^*)$ as described. For all $s, t \in T_{\Sigma^*}$ such that $s \overset{*}{\searrow} t$ we have $(\Sigma, E) \vdash \lceil s \rceil = \lceil t \rceil$.

This follows immediately from Lemma 24 and Theorem 4(Section 4.2). Theorem 32 is thus the analogon of Theorem 4.

**33. Theorem**
Let $(\Sigma, R)$ be a confluent TRS and let $(\Sigma^*, R^*)$ be as described. Let $s \in T_{\Sigma^*}$, and let $F$ and $G$ be two superstrategies. If $s \in dom(F_\omega) \bigcap dom(G_\omega)$ then $F_\omega(s) = G_\omega(s)$.

Suppose $F_\omega(S) \neq G_\omega(s)$. By definition we have $s \overset{*}{\searrow} F_\omega(S)$ and $s \overset{*}{\searrow} G_\omega(s)$, and as a consequence of Theorem 12 there must be some $t$ such that $F_\omega(s) \overset{*}{\searrow} t$ and $G_\omega(s) \overset{*}{\searrow} t$. By definition of $F_\omega$ and $G_\omega$ however, both these reduction sequences must be empty, and we have established the analogon of Theorem 5.

**34. Theorem**
Let $(\Sigma, R)$ be a Noetherian, confluent TRS and let $(\Sigma^*, R^*)$ be as described. Let $s \in T_{\Sigma^*}$, and let $F$ and $G$ be two superstrategies. Then $F_\omega(s) = G_\omega(s)$.

According to Theorem 13, $(\Sigma^*, R^*)$ is also confluent and Noetherian. Consequently $dom(F_\omega) = T_{\Sigma^*}$ for any superstrategy $F$, and Theorem 32 is always applicable. This theorem is the analogon of Theorem 6.

**14. Theorem**
Let $(\Sigma, R)$ be a Noetherian, confluent TRS and let $(\Sigma^*, R^*)$ be as described. Let $F$ be a strategy and let $\breve{F}$ be a superstrategy.

$$\forall s \in T_\Sigma \colon F_\omega(s) = \lceil \breve{F}_\omega(s) \rceil$$

This theorem is a consequence of Lemma 24, Lemma 26 and Theorem 34. This theorem states the equivalence of the traditional implementation scheme and a scheme based on TRSs, incorporating conventional programming languages.

## B.5 Parameterized relation $\preceq$ is a transitive relation

The relation $\preceq$, as defined in definition 16 is a transitive relation.

Let $u$ be a smallest (in depth) term for which an $s$ and $t$ can be found, such that $s \preceq t$ and $t \preceq u$ but $s \not\preceq u$. Now $t$ is either of the form $f(t_1, \ldots, t_n)$ (for $f \in \Sigma$) or of the form $\phi_{f(t'_1,\ldots,t'_n)}(t_1, \ldots, t_k)$.

In the first case we have $u = f(u_1, \ldots, u_n)$ and ($u$ has minimal depth) $s = \phi_{f(s'_1,\ldots,s'_n)}(s_1, \ldots, s_l)$, and for all $i$ we have $\Phi_i^{f(s'_1,\ldots,s'_n)}(s_1, \ldots, s_l) \preceq t_i \preceq u_i$. Again because $u$ has minimal depth, we may assume transitivity for all $u_i$, and consequently for each $i$ we have $\Phi_i^{f(s'_1,\ldots,s'_n)}(s_1, \ldots, s_l) \preceq u_i$. From this we may conclude $s \preceq u$.

In the second case we have that $u = f(u_1, \ldots, u_n)$ (because $u$ has minimal depth), and that $s = \phi_{f(t'_1,\ldots,t'_n)}(s_1, \ldots, s_k)$ and for all $i \leq k$ and $j \leq n$ we have $\Phi_j^{f(t'_1,\ldots,t'_n)}(t_1, \ldots, t_k) \preceq u_j$ and $s_i \preceq t_i$, and hence (Definition 15) $\Phi_j^{f(t'_1,\ldots,t'_n)}(s_1, \ldots, s_k) \preceq \Phi_j^{f(t'_1,\ldots,t'_n)}(t_1, \ldots, t_k)$. Since $u$ has smallest depth we may again conclude that $s \preceq u$, which concludes the proof of transitivity of $\preceq$.

## B.6 The equivalence of cpl implementations and TRS implementation for parameterized data types

**35. Lemmas**

1. $\forall s, t \in T_{\Sigma^*} \colon s \searrow t \Rightarrow \lceil s \rceil \xrightarrow{*} \lceil t \rceil$
2. $\forall s, t \in T_{\Sigma^*} \colon s \preceq t \Rightarrow t \searrow^* s$

As an example, we will show the proof of Lemma 35.1. The proof is structured as the proof of the equivalent lemma, Lemma 24.

Apparently there are a context $C[.]$, a substitution $\sigma$ ranging over $T_{\Sigma^*}$ and a rule $s_0 \rightarrow t_0$ in $R^*$, such that $s \preceq C[s_0^\sigma]$ (or $s = C[s_0^\sigma]$ for $s_0 \rightarrow t_0 \in R^\prec$) and $t = C[t_0^\sigma]$. Now define the context $C'[.] = \lceil C[.] \rceil$ and the substitution $\sigma' = \lceil \sigma \rceil$.

Suppose $s_0 \rightarrow t_0 \in R^\prec$. Then $s_0 = f(\Phi_1^u(v_1^u, \ldots, v_{\#u}^u), \ldots, \Phi_n^u(v_1^u, \ldots, v_{\#u}^u))$ and $t_0 = \phi_u(v_1^u, \ldots, v_{\#u}^u)$ for some $u = f(u_1, \ldots, u_n) \in N$. Now we have $\lceil s \rceil = C'[f(u_1^{\sigma'}, \ldots, u_n^{\sigma'})] = \lceil t \rceil$, and consequently an (empty) reduction sequence as required has been established.

Now suppose $s_0 \rightarrow t_0 \in R$. Then $\lceil s \rceil = C'[\lceil s_0^\sigma \rceil] = C'[s_0^{\sigma'}]$ and $\lceil t \rceil = C'[t_0^{\sigma'}]$, and we have established $\lceil s \rceil \rightarrow \lceil t \rceil$.

Finally suppose $s_0 \rightarrow t_0 \in R^{cpl}$. Then $s_0$ is of the form $f(vecs1n)$, and a reduction sequence $f(\lceil s_1 \rceil, \ldots, \lceil s_n \rceil) \xrightarrow{+} \lceil t_0 \rceil$ exists. We now have $\lceil s \rceil = C'[f(\lceil s_1 \rceil, \ldots, \lceil s_n \rceil)]$ and $\lceil t \rceil = C'[\lceil t_0 \rceil]$ and we have established the desired reduction sequence.

We will leave the proof of Lemma 35.2. as an exercise. The earlier proofs of the other lemma's remain valid.

The earlier proofs of all other lemma's and theorems in Section 4.7 all remain valid.

# Chapter 5

# Efficiency of hybrid implementations

We present a single example specification and observe the effect on execution speed when various parts of this specification are implemented in a conventional programming language using the hybrid specification technique.

In this chapter we will look at an extended example of hybrid specifications, and we will focus our attention on execution speed. The example concerns the dynamic semantics of a small programming language. We will provide hybrid implementations of some of the datatypes in the specification, and we will discuss various measurements of the execution times.

We observe that hybrid implementation of carefully selected basic datatypes reduces the total execution time by a factor 3.

## 5.1 The programming language Nano

In [3] the programming language Pico was introduced. Pico is a small programming language with strings and natural numbers as datatypes. Statements are assignment, an if-construct and a while-construct. There are no procedures or functions in Pico, apart from the standard operators on strings (concatenation) and numbers (addition and subtraction). Booleans are simulated with numbers, where 0 indicates *false* and any other value indicates *true*. Pico is strongly typed, and all variables must be declared with a type, either `string` or `natural`. In [3] the syntax and static and dynamic semantics of Pico are defined.

As it is, Pico is not suitable for our purposes. The key datatype, natural numbers with addition and subtraction, is unrealistically simple. For instance, multiplication in Pico has to be done by repeated addition, using an auxiliary variable. Furthermore, we cannot expect much gain in efficiency from a hybrid implementation of that datatype. The complexity of addition on natural numbers is linear in the size of its arguments, but the worst case complexity of any hybrid operator is also linear in the complexity of its arguments, because their internal representation has to be established.

We will present an extension of Pico, resulting in the – still tiny – language Nano. We will extend the natural numbers with various operators such as multiplication, and we will add a new datatype: array of natural number.

In the remainder of this section we will present the syntax of Nano, and briefly discuss its static semantics (type checking). Then we will present a specification of the dynamic semantics (execution) of Nano.

We will not give a specification of the static semantics, because in Section 5.2 we only look at the dynamic semantics. Measurements of the execution speed using a hybrid specification of the static semantics of Nano lead to results quite similar to those presented in Section 5.2. In fact this is not surprising, because the key datastructure used for typechecking, the symbol table, is also used for the dynamic semantics.

### 5.1.1 The syntax of Nano

In this section we will present the syntax of Nano. First, we will discuss some basic notions. We will not show all modules.

In the module `Identifiers` Nano identifiers are defined by the following rule.

```
[a-z][a-z0-9]* -> ID
```

That is, a NANO identifier is a lower case letter followed by zero or more lower case letters or digits.

In the modules `Strings` and `Naturals` the basic datatypes are defined. These modules export the two sorts `STR-CON` and `NAT-CON`, representing constants of the two types (strings of symbols, quotes excluded, and surrounded by quotes, and sequences of one or more digits, respectively).

The module `Types` defines the three Nano type names: `natural`, `string` and `array`.

We can now present the module `Syntax`.

```
module Syntax
imports Identifiers Strings Naturals Types
exports
   sorts PROGRAM DECLS ID-TYPE STATEMENT EXP
   context-free syntax
      begin DECLS {STATEMENT ";"}* end      -> PROGRAM
      declare {ID-TYPE  ","}* ";"           -> DECLS
      ID ":" TYPE                           -> ID-TYPE
      ID ":=" EXP                           -> STATEMENT
      ID "[" EXP "]"  ":=" EXP              -> STATEMENT
```

```
      if EXP then {STATEMENT ";"}*
         else {STATEMENT ";"}* fi           -> STATEMENT
      while EXP do {STATEMENT ";"}* od      -> STATEMENT
      ID                                    -> EXP
      NAT-CON                               -> EXP
      STR-CON                               -> EXP
      EXP "+" EXP                           -> EXP {left}
      EXP "-" EXP                           -> EXP {left}
      EXP "*" EXP                           -> EXP {left}
      EXP "div" EXP                         -> EXP {left}
      EXP "mod" EXP                         -> EXP {left}
      EXP "and" EXP                         -> EXP {left}
      EXP "=" EXP                           -> EXP {left}
      EXP "<=" EXP                          -> EXP {left}
      EXP "||" EXP                          -> EXP {left}
      ID "[" EXP "]"                        -> EXP
      "(" EXP ")"                           -> EXP {bracket}
   variables
      Decls [0-9']*      -> DECLS
      Id-type-list       -> {ID-TYPE ","}*
      Stat [0-9']*       -> STATEMENT
      Stat-list [0-9']*  -> {STATEMENT ";"}*
      Stat-plus [0-9']*  -> {STATEMENT ";"}+
      Exp [0-9']*        -> EXP
priorities
   EXP "and" EXP -> EXP <
   { EXP "=" EXP -> EXP, EXP "<=" EXP -> EXP} <
   { EXP "+" EXP -> EXP, EXP "-" EXP -> EXP} <
   { EXP "*" EXP -> EXP, EXP "div" EXP -> EXP,
     EXP "mod" EXP -> EXP}
```

Some comments are in order. All differences between Nano and Pico are indicated.

- A program is a list of declarations followed by a list of statements. A declaration contains an identifier and a type. The type may be `natural`, `string` or `array`.
- There are two kinds of assignment. An ordinary assignment to a plain variable, and an assignment to an array entry. In Pico only the former kind exists.
- Only one if-statement exists, but the 'then-statements-list' and 'else-statement-list' may be empty.
- Natural number operators `*`, `div`, `mod`, `and`, `=`, and `<=` have been added, all with the obvious meaning. The 'boolean' operators will behave consistent. That is, they will return 0 to indicate failure, and another value

to indicate success. Note that in Pico (and also in Nano) subtraction means cut-off difference: if the integer difference is negative, then the cut-off difference is zero.

- There is only one string operator: concatenation.
- One other expression has been added: retrieval of an entry from an array.

### 5.1.2 Static semantics

For Nano, just as for Pico, the static semantics contains very few rules.

- All variables occurring in a program must have been declared.
- Both sides of an ordinary assignment must have the same type. This type should not be `array`.
- In an array-entry assignment, the identifier must be of type `array`, and both expressions appearing in it must be of type `natural`.
- In an if-statement, or while-statement, the expression must be of type `natural`.
- Expressions must be correct, and only then they do have a type:
  - ▷ `NAT-CON` and `STR-CON` are always correct, and their type is evident.
  - ▷ A variable in an expression is only correct if it has been declared `string` or `natural`, in which case its type is evident.
  - ▷ An operator expression is only correct if both its arguments are correct and of the proper type. The types of the expression and of its arguments follow directly from the operator used.

### 5.1.3 Dynamic semantics

In this section we will present the dynamic semantics of Nano. It is assumed that any program under consideration has already successfully been typechecked.

The module `strings` is straightforward. Is is identical to the same module in the specification of Pico, and since we will not use strings, the module is not shown.

The module `Naturals` defines natural numbers in the common `S`-`Z` representation (capitals are used to avoid confusion with Nano variables).

```
module Naturals
imports Layout
exports
   sorts NAT NAT-CON DIG
   lexical syntax
```

```
      [0-9] -> DIG
    context-free syntax
      "Z"           -> NAT
      "S" NAT       -> NAT
      NAT "+" NAT   -> NAT {assoc}
      NAT "-" NAT   -> NAT {left}
      NAT "*" NAT   -> NAT {assoc}
      NAT "div" NAT -> NAT {left}
      NAT "mod" NAT -> NAT {left}
      NAT "and" NAT -> NAT {assoc}
      NAT "<=" NAT  -> NAT
      NAT "=" NAT   -> NAT
      "(" NAT ")"   -> NAT {bracket}
      DIG+          -> NAT-CON
      NAT-CON       -> NAT
    variables
      Nat [0-9']*     -> NAT
      Nat-con [0-9']* -> NAT-CON
      Dig [0-9']*     -> DIG
      Digs [0-9']*    -> DIG+
  priorities
    {"=","<="} <
    {"+","-"} <
    {"*","div","mod",NAT "and" NAT -> NAT} <
    "S"
  equations
  [a] Nat + Z = Nat
  [b] Nat1 + S Nat2 = S Nat1 + Nat2
  [c] Nat * Z = Z
  [d] Nat1 * S Nat2 = Nat1 + Nat1 * Nat2
  [e] Nat - Z = Nat
  [f] Z - Nat = Z
  [g] S Nat1 - S Nat2 = Nat1 - Nat2
  [h] S Nat1 - Nat2 = Z
      =================
      Nat1 div Nat2 = Z
  [i] Nat2 != Z,
      Nat2 - Nat1 = Z
      ==========================================
      Nat1 div Nat2 = S ((Nat1 - Nat2) div Nat2)
  [j] S Nat1 - Nat2 = Z
      ====================
      Nat1 mod Nat2 = Nat1
  [k] Nat2 != Z,
      Nat2 - Nat1 = Z
```

```
    ======================================
    Nat1 mod Nat2 = (Nat1 - Nat2) mod Nat2
[l] Nat and Z = Z
[m] Z and Nat = Z
[n] S Nat1 and S Nat2 = S Z
[o] Z <= Nat = S Z
[p] S Nat <= Z = Z
[q] S Nat1 <= S Nat2 = Nat1 <= Nat2
[r] (Nat = Nat) = S Z
[t] Nat1 != Nat2
    =================
    (Nat1 = Nat2) = Z
[0] 0 = Z
[1] 1 = S Z
[2] 2 = S S Z
[3] 3 = S S S Z
[4] 4 = S S S S Z
[5] 5 = S S S S S Z
[6] 6 = S S S S S S Z
[7] 7 = S S S S S S S Z
[8] 8 = S S S S S S S S Z
[9] 9 = S S S S S S S S S Z
[10] Digs Dig = S S S S S S S S S S Z * Digs + Dig
```

Syntactic constants are represented as sequences of digits. These can then be transformed to the S-Z notation using equations [1] to [10]. All equations are self-explanatory.

The module Arrays defines arrays, with natural numbers both as index and as entry values. The lowest index is Z, and there is no upper bound. Arrays are extended (zero-filled) when an index higher than the current upper bound is used.

```
module Arrays
imports Layout Naturals
exports
   sorts ARR NAT-ARR
   context-free syntax
      null                        -> ARR
      set ARR "@" NAT "to" NAT    -> ARR
      ARR "@" NAT                 -> NAT
   variables
      Arr [0-9']*  -> ARR
hiddens
   context-free syntax
      NAT "," ARR -> ARR
equations
[a] null @ Nat = Z
```

```
[b] Nat, Arr @ Z = Nat
[c] Nat, Arr @ S Nat' = Arr @ Nat'
[d] set null @ Z to Nat = Nat, null
[e] set null @ S Nat' to Nat = Z, set null @ Nat' to Nat
[f] set Nat, Arr @ Z to Nat' = Nat', Arr
[g] set Nat1, Arr @ S Nat2 to Nat3
    = Nat1, set Arr @ Nat2 to Nat3
```

The module `Values` imports `Naturals`, `Strings` and `Arrays`, and injects the relevant sorts into a single sort `VALUE`. Arrays do not appear as the value of expressions in Nano programs, but they do appear as the current value of variables, and they can be modified by altering individual entries.

The module `Value-environments` defines tables which maintain the values of all variables during execution.

```
module Value-environments
imports Identifiers Values
exports
   sorts VENV VPAIR
   context-free syntax
      "(" ID ":" VALUE ")"          -> VPAIR
                                    -> VENV
      VPAIR ";" VENV                -> VENV
      lookup ID in VENV             -> VALUE
      update ID in VENV "by" VALUE -> VENV
   variables
      Venv [0-9']*       -> VENV
equations
[VE1]    lookup Id in (Id:Value); Venv = Value
[VE2]       Id != Id'
         ==============================
         lookup Id in (Id':Value); Venv
         = lookup Id in Venv
[VE3]    update Id in (Id:Value); Venv by Value'
         = (Id:Value'); Venv
[VE4]       Id != Id',
            update Id in Venv by Value' = Venv'
         ========================================
         update Id in (Id':Value); Venv by Value'
         = (Id':Value); Venv'
```

Note that there is a constructor function '`;`', for which no equations are given, and a function `update`, which removes the previous entry with the same key.

Finally, we can present the module where the evaluation of Nano programs is defined.

```
module Eval
imports Syntax Value-environments
```

```
exports
   context-free syntax
      ev "[" PROGRAM "]" -> VENV
hiddens
   context-free syntax
      ev "[" DECLS "]"                       -> VENV
      ev "[" {STATEMENT ";"}* "]" in VENV    -> VENV
      ev "[" EXP "]" in VENV                 -> VALUE
equations
[Ev1]    ev[begin Decls Stat-list end] = ev[Stat-list] in ev[Decls]
[Ev2a]      ev[declare Id-type-list ;] = Venv
         =====================================================
         ev[declare Id:natural, Id-type-list ;] = (Id:Z); Venv
[Ev2b]      ev[declare Id-type-list ;] = Venv
         ====================================================
         ev[declare Id:string, Id-type-list;] = (Id:""); Venv
[Ev2c]      ev[declare Id-type-list ;] = Venv
         =====================================================
         ev[declare Id:array, Id-type-list;] = (Id:null); Venv
[Ev3]    ev[declare ;] =
[Ev4]    ev[ ] in Venv = Venv
[Ev5a]   Arr1 = lookup Id in Venv,
         Nat1 = ev[Exp1] in Venv,
         Nat2 = ev[Exp2] in Venv,
         Arr2 = set Arr1 @ Nat1 to Nat2
         ========================================================
         ev[Id[Exp1] := Exp2] in Venv = update Id in Venv by Arr2
[Ev5]    ev[Id := Exp] in Venv = update Id in Venv by ev[Exp] in Venv
[Ev6a]      ev[Exp] in Venv != Z
         =====================================================
         ev[if Exp then Stat-list1 else Stat-list2 fi] in Venv
         = ev[Stat-list1] in Venv
[Ev6b]      ev[Exp] in Venv = Z
         =====================================================
         ev[if Exp then Stat-list1 else Stat-list2 fi] in Venv
         = ev[Stat-list2] in Venv
[Ev7a]      ev[Exp] in Venv = Z
         ============================================
         ev[while Exp do Stat-list od] in Venv = Venv
[Ev7b]      ev[Exp] in Venv != Z,
            ev[Stat-list] in Venv = Venv'
         ========================================
         ev[while Exp do Stat-list od] in Venv
         = ev[while Exp do Stat-list od] in Venv'
[Ev8]       ev[Stat] in Venv = Venv'
```

```
        ====================================================
        ev[Stat; Stat-plus] in Venv = ev[Stat-plus] in Venv'
[Ev9]   ev[Id] in Venv = lookup Id in Venv
[Ev10]  ev[Nat-con] in Venv = Nat-con
[Ev11]  ev[Str-con] in Venv = Str-con
[Ev12]    ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ==================================
        ev[Exp1 + Exp2] in Venv = Nat1 + Nat2
[Ev13]    ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ==================================
        ev[Exp1 - Exp2] in Venv = Nat1 - Nat2
[Ev13a]   ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ==================================
        ev[Exp1 * Exp2] in Venv = Nat1 * Nat2
[Ev13b]   ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ========================================
        ev[Exp1 div Exp2] in Venv = Nat1 div Nat2
[Ev13c]   ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ========================================
        ev[Exp1 mod Exp2] in Venv = Nat1 mod Nat2
[Ev13d]   ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ========================================
        ev[Exp1 and Exp2] in Venv = Nat1 and Nat2
[Ev13f]   ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ==================================
        ev[Exp1 = Exp2] in Venv = Nat1 = Nat2
[Ev13g]   ev[Exp1] in Venv = Nat1,
          ev[Exp2] in Venv = Nat2
        ====================================
        ev[Exp1 <= Exp2] in Venv = Nat1 <= Nat2
[Ev14]    ev[Exp1] in Venv = Str1,
          ev[Exp2] in Venv = Str2
        ====================================
        ev[Exp1 || Exp2] in Venv = Str1 || Str2
[Ev15]    ev[Id] in Venv = Arr,
          ev[Exp] in Venv = Nat
        ==============================
        ev[Id[Exp]] in Venv = Arr @ Nat
```

As a single example we will discuss equation [Ev5a], in which the execution of the array assignment is defined. The evaluation of `Id[Exp1] := Exp2 in Venv`, is defined by changing the value of `Id` in `Venv` to `Arr2`, where the current value of `Id` in `Venv` is some array `Arr1`, where `Nat1` and `Nat2` are the values of `Exp1` and `Exp2`, respectively, using `Venv`, and where `Arr2` is obtained by setting the entry in `Arr1` with index `Nat1` to `Nat2`

### 5.1.4 An example program

We will now present an example program. The program computes all odd prime numbers between 11 and 33.

```
begin
  declare prime : natural,
          index : natural,
          next : natural,
          primes : array,
          stillprime : natural,
          inrange : natural,
          factor : natural;
  primes[0] := 3;
  primes[1] := 5;
  primes[2] := 7;
  next := 3;
  prime := 11;
  while prime <= 33 do
    index := 0;
    stillprime := 1;
    inrange := 1;
    while stillprime and inrange do
      factor := primes[index];
      stillprime := prime mod factor;
      inrange := factor * factor <= prime;
      index := index + 1
    od;
    if stillprime then
      primes[next] := prime;
      next := next + 1
    else fi;
    prime := prime + 2
  od
end
```

This program will be used in the next section to perform all measurements upon.

## 5.2 Hybrid specifications and measurements

In this section we will present figures indicating the execution speed of the specification of the dynamic semantics of Nano presented above. Using these figures, we will indicate bottle-necks and present hybrid implementations of parts of the specification, each time discussing the effect on the execution speed.

The measurements presented in this section were done at a time when the ASF+SDF system did not yet support hybrid implementations. All measurements were done on a prototype implementation of hybrid term rewriting. Similarly, the notation used for hybrid equations was not yet fully accepted by the ASF+SDF system. We feel however, that this notation is much more readable than the one used in the prototype, so we will use the ASF+SDF notation with the understanding that all code shown has not been checked by the system in that form.

It cannot be assumed that every reader is well acquainted with Lisp. For that reason most Lisp code has been modified in such a way that it is functionally the same as the original, without using obscure Lisp constructs.

As mentioned earlier, all measurements will be performed on the same test case: the primes program presented in Section 5.1.4. That is, all measurements concern the normalization of the term `ev[`$\langle primes \rangle$`]`, where $\langle primes \rangle$ is the primes program as presented in Section 5.1.4. This will be referred to as **the test**.

The term rewriting implementation uses the left-most innermost reduction strategy. A reduction step occurs when an instantiation of the left-hand side of a rule is replaced by the corresponding instance of the right-hand side of that rule, after verifying the conditions of course. In the hybrid case this instantiation involves the evaluation of a generator (the expression in the right-hand side of the rule).

Reconstruction of terms in the interior of which a reduction step has taken place is not considered as a separate reduction step. Reduction steps performed during evaluation of conditions, whether the condition succeeds or not, are counted. Finally, the hybrid implementation sometimes needs to 'unwrap' internal representations. Such an unwrapping is also counted as a rewriting step.

Apart from the measurements of the execution time we will attempt to indicate the effectiveness of our method. For this purpose it is necessary to assume that every reduction step takes roughly the same time, and that the time spent searching redexes is uniformly distributed over reduction steps. In fact this is a simplifying assumption.

- The complexity of rules – both left-hand sides and right-hand sides – differs. Hence, the time spent for successful matches, and for instantiations, differs.

- The number of rules per head-symbol (i.e., the outermost function symbol of the left-hand side of the rule) varies, so lists of different lengths must be searched. Both our prototype and the ASF+SDF system divide the rules into immediately accessible sublists with the same head-symbol.
- The average time spent in unsuccessful matches – until a redex is found – depends on the 'average' term, and on the given term rewriting system.

In our implementation it is impossible to make precise measurements at this level, but based on the assumption mentioned above, we can compare numbers of reduction steps. All figures obtained in this fashion will be marked †, to indicate that they are not derived from real time measurements.

All measurements are done on a Sun Sparc station 1+ under SunOs 4.1, running LeLisp 15.22. Seconds shown are user seconds as supplied by LeLisp. All Lisp code is interpreted. Compiling the Lisp code gains a constant factor in speed, but otherwise does not significantly influence our findings.

### Ordinary rewriting

Before looking at hybrid specifications we will look at our basis of comparison: ordinary rewriting. The first measurement is established using a standard term rewriting implementation. That is, an implementation which does not support hybrid specifications, and which consequently does not have any overhead.

In this system the test executes in 16.5 seconds and requires 7299 reduction steps.

The second measurement we will look at illustrates the worst case. It is the execution time for the same – ordinary – specification using the hybrid implementation. This figure is worst-case, in the sense that it is burdened by the entire overhead of the hybrid method, without using any of its benefits.

In this situation the execution time is 20.5 seconds. The number of reduction steps is of course identical: 7299.

This figure suggests that 'the price to pay' for a hybrid implementation, whether it is used or not, does not exceed 25%. This figure must be considered in the context of the ASF+SDF system, where a user using a library may even be unaware of using hybrid implementations.

We can now make a hybrid implementation for the specification presented earlier. The first question is: which datatypes would benefit from a hybrid implementation? Obvious choices are natural numbers and tables. In general, strings would also be a good choice, but they are not used in our test, and we will not look at them. We will create hybrid implementations of numbers and tables in two stages, and then we will discover yet another datatype which may benefit from hybrid implementation.

The next question is: which datatype shall we start with. In principle, the two implementations can be altered independently, so we will start with the one which yields the most benefits. For this purpose, Table 5.1 contains a breakdown of all reduction steps categorized by head-symbol (i.e., the outermost function symbol of the subterm being replaced).

| `+` | `-` | `*` | `mod` | `&` | `<=` | $\odot$ | 0—9 |
|---|---|---|---|---|---|---|---|
| 640 | 1818 | 118 | 144 | 56 | 934 | 2 | 17 |
| naturals | | | | | | | |
| 3729 | | | | | | | |
| `insert` | `lookup` | `set` | `get` | `ev` prog. | `ev` decl's | `ev` stat's | `ev` exp's |
| 673 | 1519 | 55 | 34 | 1 | 8 | 617 | 663 |
| arrays and tables | | | | `ev` | | | |
| 2281 | | | | 1289 | | | |
| total | | | | | | | |
| 7299 | | | | | | | |

*Table 5.1.*

The operator marked $\odot$ is the invisible operator which strings together digits into natural number constants. This operator is reduced in equation [10] in module `Naturals`.

From this table we can see that of the 7299 reductions in total, 3729 concern numbers, 2281 concern tables and arrays, and the remaining 1289 concern the function `ev`. We will start with natural numbers.

### The hybrid specification of natural numbers

Below is our hybrid specification of natural numbers. The notation used has already been discussed in Chapter 1. Note that equation [U] is an unwrapping equation, used for matching only.

```
module Naturals
imports Lisp Layout
exports
   sorts NAT NAT-CON DIG
   lexical syntax
      [0-9] -> DIG
   context-free syntax
      NAT           -> LISP {alien}
      "Z"           -> NAT
      "S" NAT       -> NAT
      NAT "+" NAT   -> NAT {assoc}
      NAT "-" NAT   -> NAT {left}
      NAT "*" NAT   -> NAT {assoc}
      NAT "div" NAT -> NAT {left}
      NAT "mod" NAT -> NAT {left}
      NAT "and" NAT -> NAT {assoc}
```

```
        NAT "<=" NAT  -> NAT
        NAT "=" NAT   -> NAT
        "(" NAT ")"   -> NAT {bracket}
        DIG+          -> NAT-CON
        NAT-CON       -> NAT
     variables
        Nat [0-9']*     -> NAT
        Nat-con [0-9']* -> NAT-CON
        Dig [0-9']*     -> DIG
        Digs [0-9']*    -> DIG+
  priorities
     {"=","<="} <
     {"+","-"} <
     {"*","div","mod",NAT "and" NAT -> NAT} <
     "S"
  equations
  [a] Z = LISP 0
  [b] S LISP x = LISP (1+ x)
  [c] LISP x + LISP y = LISP (+ x y)
  [d] LISP x - LISP y = LISP (or (plusp (- x y)) 0)
  [e] LISP x * LISP y = LISP (* x y)
  [f] LISP x div LISP y = LISP (div x y)
  [g] LISP x mod LISP y = LISP (modulo x y)
  [h] LISP x and LISP y = LISP (or (zerop x) (zerop y) 1)
  [i] LISP x = LISP y = LISP (if (eq x y) 1 0)
  [j] LISP x <= LISP y = LISP (if (<= x y) 1 0)
  [U] LISP x = NAT (if (zerop x) <<0>> <<S LISP (1- x) >>)
  [0] 0 = Z
  [1] 1 = S Z
  [2] 2 = S S Z
  [3] 3 = S S S Z
  [4] 4 = S S S S Z
  [5] 5 = S S S S S Z
  [6] 6 = S S S S S S Z
  [7] 7 = S S S S S S S Z
  [8] 8 = S S S S S S S S Z
  [9] 9 = S S S S S S S S S Z
  [10] Digs Dig = S S S S S S S S S S Z * Digs + Dig
```

The execution time of our test, using this specification is 12.0 seconds. It required 4270 reductions (including ordinary and hybrid reductions, and unwrappings). Table 5.2 contains the breakdown in this case.

| `+` | `*` | `mod` | `&` | `<=` | $\odot$ | `0—9` | `Z` | `S` |
|---|---|---|---|---|---|---|---|---|
| 43 | 24 | 22 | 56 | 47 | 2 | 17 | 138 | 152 |
| naturals | | | | | | | | |
| 501 | | | | | | | | |
| {unwrap} | `insert` | `lookup` | `set` | `get` | (p) | (d) | (s) | (e) |
| | 673 | 1519 | 55 | 34 | 1 | 8 | 617 | 663 |
| | arrays and tables | | | | `ev` | | | |
| 199 | 2281 | | | | 1289 | | | |
| total | | | | | | | | |
| 4270 | | | | | | | | |

*Table 5.2.*

The number of unwrappings indicated is the number of times the unwrapping equation was applied: this may be more than once during a single match, and does not depend on the final success of the match.

From the table we can see that now only 700 rewritings (including unwrappings) concerned numbers. The `S` and `Z` rewritings are needed to convert terms to their internal representation. We can also see that 211 reductions concerning numbers are essential. They correspond to 'calls' to functions such as `+` or `*`, or to the conversion of digits, and no implementation of natural numbers could avoid these calls. Consequently a conceptual, optimal implementation would have reduced the total number of steps to 211 (reducing it by 3518 steps).

With these figures, and using the assumption discussed in the introduction, we can estimate the maximal possible decrease in execution time, which is 48%† (3518 over 7299). In fact, the reduction achieved was 41%, when comparing to the hybrid implementation, or 27%, when comparing to the plain implementation. We can express the effectiveness of our method as the ratio between the optimal and actual gain, which is 85%†.

Curiously enough we can to some degree verify the assumption used here. Based on this assumption the decrease in execution time should be 3029 (7299 - 4270) over 7299, which is precisely 41%†, the actual figure measured.

### Arrays and Tables

We will now present a hybrid implementation of arrays and tables (value environments). In fact adding this implementation of arrays is quite arbitrary, because in Table 5.2 we can see that the use of arrays is relatively limited.

```
module Arrays
imports Lisp Layout Naturals
exports
   sorts ARR NAT-ARR
   context-free syntax
      ARR                        -> ARR {alien}
```

```
      null                       -> ARR
      set ARR "@" NAT "to" NAT   -> ARR
      ARR "@" NAT                -> NAT
hiddens
   context-free syntax
   NAT "," ARR -> ARR
equations
[a] null = LISP nil
[b] LISP x, LISP l = LISP (cons x l)
[c] LISP l @ LISP x = LISP (or (nth x l) 0)
[d] set LISP l @ LISP x to LISP y
    = LISP (prog (while (<= (length l) x) (nconc1 l 0))
                 (rplaca (nthcdr x l) y)
                 (return l))

module Value-environments
imports Lisp Identifiers Values
exports
   sorts VENV VPAIR
   context-free syntax
      VENV                         -> VENV {alien}
      "(" ID ":" VALUE ")"         -> VPAIR
                                   -> VENV
      VPAIR ";" VENV               -> VENV
      lookup ID in VENV            -> VALUE
      update ID in VENV "by" VALUE -> VENV
   variables
      Venv [0-9']*       -> VENV
equations
[VE1]  = LISP nil
[VE2] (Id : LISP v); LISP env
      = LISP (acons (token << Id >>) v env)
[VE3] lookup Id in LISP env
      = LISP (cassoc (token << Id >>) env)
[VE4] update Id in LISP env by LISP v
      = LISP (prog (rplacd (assoc (token << Id >>) env) v)
                   (return env))
```

Some comments are in order.

- In module `Arrays` the hybrid implementation of numbers is a prerequisite. It is possible to create parameterized hybrid specifications, but there is an extra overhead in execution time which may outweigh the possible gain for simple datatypes.

- The Lisp function `token` gets the representation of a constant defined by some lexical-grammar rule, and returns the actual lexical token for that constant. This token uniquely identifies the constant.

- The standard Lisp functions `acons`, `cassoc` and `assoc` create an association list and retrieve a value or a key-value pair, respectively. The function `rplacd` modifies the value component of a key-value pair.

Our test, using this specification, takes 5.8 seconds.

In Table 5.3 we will again find a breakdown of head-symbols.

<table>
<tr><td>+</td><td>*</td><td>mod</td><td>&</td><td><=</td><td>$\odot$</td><td>0—9</td><td>Z</td><td>S</td></tr>
<tr><td>43</td><td>24</td><td>22</td><td>56</td><td>47</td><td>2</td><td>17</td><td>138</td><td>152</td></tr>
<tr><td colspan="9">naturals</td></tr>
<tr><td colspan="9">501</td></tr>
<tr><td rowspan="3">{unwrap}</td><td>insert</td><td>lookup</td><td>set</td><td>get</td><td>null</td><td>$\otimes$</td><td>;</td><td rowspan="3">ev</td></tr>
<tr><td>155</td><td>375</td><td>10</td><td>22</td><td>1</td><td>1</td><td>7</td></tr>
<tr><td colspan="7">arrays and tables</td></tr>
<tr><td>110</td><td colspan="7">571</td><td>1289</td></tr>
<tr><td colspan="9">total</td></tr>
<tr><td colspan="9">2471</td></tr>
</table>

*Table 5.3.*

The symbol $\otimes$ indicates the empty value environment.

Observe that the number of unwrappings has decreased. The reason is that arrays use the internal representation of numbers directly.

From the table we see that 1182 rewritings concern the datatypes with a hybrid implementation (numbers, tables and arrays), as opposed to 6010 using plain rewriting. Of these, 773 are essential. The 'optimal' implementation would decrease the execution time by 72%† (6010-773 over 7299). The actual decrease amounted to 71%, compared to the hybrid implementation, and 65% compared to the plain implementation. Note, however, that our assumption leading to the estimate of 72%†, does not seem to be reasonable now. Using it, the actual decrease in time would be estimated at 66%†, but proved to be 71%. Using this knowledge, a better estimate of the optimal decrease would be 77%†.

Keeping the demonstrated inaccuracy of the estimates in mind, the effectiveness of our method now becomes 92%†.

### Beyond basic datatypes

Looking at the breakdown in Table 5.3 we can see that all basic datatypes have been given a hybrid implementation. All reductions not yet dealt with concern the operator `ev[...]`. Consequently, the only way to enhance the execution time of our specification even further, is to give this operator a hybrid implementation.

Let us consider this operator in some more detail. In fact there are four distinct operators taking, respectively, a program, a list of declarations, a list of statements and an environment, and an expression and an environment as arguments. The first three operators yield an environment, and the last yields a value. Disregarding strings for the moment, all output sorts already have a hybrid implementation (values are numbers, strings or arrays). Likewise, the environment arguments are hybrid. Consequently, the only sort suitable would be the syntactic constructs. But there is no representation of syntactic constructs, which is more compact or efficient than abstract syntax trees (terms). We conclude that the representation itself is already optimal for a hybrid implementation. If we assume that the term representation could easily be accessed from Lisp, we do not need to have a separate hybrid representation, but we use the term representation itself. Currently this cannot be done in the ASF+SDF system. For the moment we will use a notation which is intuitively clear.

Remember that the measurements are obtained by transliteration of this specification to a format suitable for our prototype system.

In the hybrid module `Eval` below, a hybrid implementation has been given for two of the four operators: concerning expressions and concerning statements.

```
module Eval
imports Lisp Syntax Value-environments
exports
   context-free syntax
      ev "[" PROGRAM "]" -> VENV
hiddens
   context-free syntax
      ev "[" DECLS "]"                        -> VENV
      ev "[" {STATEMENT ";"}* "]" in VENV    -> VENV {alien}
      ev "[" EXP "]" in VENV                  -> VALUE {alien}
   variables
      Stat [0-9']*                            -> STATEMENT
equations
[Ev1]    ev[begin Decls Stat-list end] = ev[Stat-list] in ev[Decls]
[Ev2a]      ev[declare Id-type-list ;] = Venv
         ====================================================
         ev[declare Id:natural, Id-type-list ;] = (Id:Z); Venv
[Ev2b]      ev[declare Id-type-list ;] = Venv
         ====================================================
         ev[declare Id:string, Id-type-list;] = (Id:""); Venv
[Ev2c]      ev[declare Id-type-list ;] = Venv
         ====================================================
         ev[declare Id:array, Id-type-list;] = (Id:null); Venv
[Ev3]    ev[declare ;] =
[Ev4]    ev[Stat] in LISP env =
  LISP
  (prog
```

```
      (selectq (token << Stat >>)
        ('|Stat-list-operator|
           (let ((stats (args << Stat >>)))
             (while stats
               (evs (nextl stats) env))))
        ('|:=|
           (rplacd (assoc (token (arg1 << Stat >>)) env)
                   (eve (arg2 << Stat >>) env)))
        ('|[]:=|
           (let ((array (assoc (token (arg1 << Stat >>)) env))
                 (index (eve (arg2 << Stat >>) env))
                 (value (eve (arg3 << Stat >>) env)))
             (while (<= (length array) index)
               (nconc1 array 0))
             (rplaca (nthcdr index array) value)))
        ('|if|
          (ifn (zerop (eve (arg1 << Stat >>) env))
            (evs (arg2 << Stat >>) env)
            (evs (arg3 << Stat >>) env)))
        ('|while|
          (while (posp (eve (arg1 << Stat >>) env))
            (evs x (val (evs (cadr args) env))))))
      (return env))
[Ev5]   ev[Str-con] in Venv = Str-con
[Ev6]   ev[Nat-con] in Venv = Nat-con
[Ev7]   ev[LISP x] in Venv = LISP x
[Ev8]   ev[Id] in LISP env = LISP (cassoc (token << Id >>) env)
[Ev9]     ev[Exp1] in Venv = Str1,
          ev[Exp2] in Venv = Str2
        =======================================
        ev[Exp1 || Exp2] in Venv = Str1 || Str2
[Ev10]  ev[Exp] in LISP env =
    LISP
    (prog
      (selectq (token << Exp >>)
        ('|+|
          (return (+ (eve (arg1 << Exp >>) env)
                     (eve (arg2 << Exp >>) env))))
        ('|-|
          (return (or (plusp (- (eve (arg1 << Exp >>) env)
                                (eve (arg2 << Exp >>) env)))
                      0)))
        ('|*|
          (return (* (eve (arg1 << Exp >>) env)
                     (eve (arg2 << Exp >>) env))))
```

```
('|div|
  (return (div (eve (arg1 << Exp >>) env)
               (eve (arg2 << Exp >>) env))))
('|mod|
  (return (modulo (eve (arg1 << Exp >>) env)
                  (eve (arg2 << Exp >>) env))))
('|and|
  (return (or (zerop (eve (arg1 << Exp >>) env))
              (zerop (eve (arg2 << Exp >>) env))
              1)))
('|=|
  (if (= (eve (arg1 << Exp >>) env)
         (eve (arg2 << Exp >>) env))
    (return 1)
    (return 0)))
('|<=|
  (if (<= (eve (arg1 << Exp >>) env)
          (eve (arg2 << Exp >>) env))
    (return 1)
    (return 0)))
('|[]|
  (return (or (nth (eve (arg2 << Exp >>) env)
                   (cassoc (token (arg1 << Exp >>)) env))
              0)))))
```

Some comments are in order.

- In equation [Ev4] the evaluation of statements is defined.
  - ▷ The function `token`, as before, yields a token uniquely identifying the outermost function symbol of its argument.
  - ▷ The `ev` operator may have a single statement or a list of statements as its argument. The clause '`|Stat-list-operator|`' defines the evaluation of statement lists. The function `args` yields the (Lisp) list of statements, which are then evaluated one by one. The function `evs`, used for that purpose, is precisely the Lisp body of equation [Ev4]. In reality that Lisp body is defined as a Lisp function, which is called from the equation, and which also calls itself recursively. Since external functions cannot be denoted in ASF+SDF we have chosen to list the body within the specification.
  - ▷ The functions `arg1` and `arg2` retrieve the first and second immediate subterms of a given term, respectively.
  - ▷ The function `eve` is similar to the function `evs`. It is the body of equation [Ev10], to be discussed below.
  - ▷ All remaining clauses are straightforward. At the end of the Lisp body the environment is returned. The evaluation of the statement

may have altered this environment. Note that internal recursive calls to `evs` do not create ASF+SDF representations of the environment. Consequently the making of a 'new' internal representation of the environment, as implied by the ASF+SDF function `LISP`, only happens when the entire program has been evaluated.

- Equations [Ev5] to [Ev9] deal with the exceptional cases for expressions. Note that in fact we are using the order of equations, because any term which matches with one of the equations [Ev5] to [Ev9], also matches equation [Ev10]. This can, of course, be solved by adding a negative condition to equation [Ev10] which excludes all these possibilities, but the intention is clear as it is.

- In equation [Ev10] the evaluation of expressions is defined. This equation is straightforward.

Using this specification, our test takes 0.42 seconds. A breakdown of headsymbols is given in Table 5.4.

<table>
<tr><td>+</td><td>*</td><td>$\odot$</td><td>0—9</td><td>Z</td><td>S</td></tr>
<tr><td>2</td><td>2</td><td>2</td><td>17</td><td>25</td><td>55</td></tr>
<tr><td colspan="6">naturals</td></tr>
<tr><td colspan="6">103</td></tr>
<tr><td>null</td><td>$\otimes$</td><td>;</td><td>ev (p)</td><td>ev (d)</td><td>ev (s)</td></tr>
<tr><td>1</td><td>1</td><td>7</td><td>1</td><td>8</td><td>1</td></tr>
<tr><td colspan="3">arrays and tables</td><td colspan="3">ev</td></tr>
<tr><td colspan="3">9</td><td colspan="3">10</td></tr>
<tr><td colspan="6">total</td></tr>
<tr><td colspan="6">122</td></tr>
</table>

*Table 5.4.*

Note that there are no unwrappings. Once the internal representation of a term is established, there is no reason to unwrap it again. All operators 'know' about the internal representation. In fact, all remaining rewritings only serve to establish the internal representation of all values occurring.

Also note that there are no rewritings concerning `ev` on expressions. Equation [Ev10] is never used directly, but its body is called, as the Lisp function `eve`, from equation [Ev4].

The hybrid implementation shown above is in a sense not a term rewriting implementation at all. It is the 'fastest' possible implementation of the original specification, written in Lisp. The Lisp program uses the hybrid framework as a link to syntax oriented surroundings. Furthermore, some lesser used datatypes are given a term rewriting implementation. We will comment upon this in our conclusions.

For the reason mentioned above, it is not useful to compare the number of reduction steps used by this method with those used by the other methods. Consequently we will not perform the calculations done earlier.

## 5.3 Conclusions

In Table 5.5 some measurements and estimates presented earlier are collected. For completeness, the execution times for compiled Lisp have been added.

| impl. | hybrid datatypes | execution time | | gain vs. plain | gain vs. hybrid | reductions |
|---|---|---|---|---|---|---|
| | | int. | comp. | | | |
| plain | - | 16.5 | 2.5 | - | - | 7299 |
| hybr. | - | 20.5 | 3.0 | -24% | - | 7299 |
| hybr. | nats | 12 | 2 | 27% | 48% | 4270 |
| hybr. | basic | 5.8 | 1 | 65% | 71% | 2471 |
| hybr. | all | 0.425 | 0.17‡ | - | - | 122 |

*Table 5.5.*

The figure marked ‡ is very small, and cannot be considered as very accurate.

From the small, representative example presented in this chapter, we may conclude that by using hybrid implementations for standard datatypes, an increase in execution speed may be expected of a factor 3. When considering larger specifications, it is to be expected that this factor will increase.

- When more basic datatypes are used, the overhead due to the hybrid system will relatively decrease.
- When more complicated datastructures are used the gain can be even greater. Some examples are: hash tables, B-trees, 'real' numbers and very large texts.

The transparent nature of our system allows one to add hybrid implementations as the need arises. Non critical datatypes, such as strings in our example, can retain any representation. Eventually, tools could be created to pin-point bottle-necks and to help creating hybrid specifications.

We have also seen that an even greater increase in speed can be achieved by implementing a larger part of the specification directly. The advantages of that approach, as opposed to writing the entire program directly, are twofold.

- Datatypes which are not time-critical can be implemented using term rewriting.
- The entire front-end of the ASF+SDF system can be used. Using ASF+SDF, arbitrary context-free grammars can be defined, and incremental syntax directed editors are available.

# Bibliography


[1] A.V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, Vol. 18, pp. 333—340, 1975.

[2] *The Programming Language Ada Reference Manual*, American National Standards Institute, document ANSI/MIL-STD-1815A-1983, Lecture Notes in Computer Science, vol. 155, Springer Verlag, 1983.

[3] J.A. Bergstra, J. Heering, P. Klint (Eds.), *Algebraic Specification*, The ACM press, in cooperation with Addison Wesley, 1989.

[4] J.A. Bergstra and J.W. Klop, "Conditional rewrite rules: confluence and termination" *Journal of Computer and System Sciences*, vol. 32.3, pp. 323—362, 1986.

[5] J.A. Bergstra and J.V. Tucker, "Algebraic specifications of computable and semicomputable data types," *Theoretical Computer Science*, vol. 50, pp. 137—181, 1987.

[6] L.G. Bouma and H.R. Walters, "Implementing algebraic specifications," Chapter 5, pp. 199—282, in [3].

[7] R.S. Boyer and J.S. Moore, "A fast string searching algorithm," *Communications of the ACM*, Vol. 20, 762—772, 1977.

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

[9] N.W.P. van Diepen, "Implementation of modular algebraic specifications," Report CS-R8801, CWI, Amsterdam, 1988.

[10] N.W.P. van Diepen, "SMALL – dynamic semantics of a language with GOTOs," Chapter 3, 134—161, in [3].

[11] N. Dershowitz, M. Okada and G. Sivakumar, "Canonical conditional rewrite systems," *9th Conference on Automated Deduction*, ed. E. Lusk and R. Overbeek, Lecture Notes in Computer Science, vol. 310, pp. 538—549, Springer Verlag, 1988.

[12] M. van Eekelen and R. Plasmeijer, "Specifications of reduction strategies in term rewriting systems," *Proceedings of the Graph Reduction Workshop*, ed. J.H. Fasel and R.M. Keller, Lecture Notes in Computer Science, vol. 279, pp. 215—239, Springer Verlag, 1987.

[13] M.H. van Emden and K. Yukawa, "Logic programming with equations," *Journal of logic programming*, Vol. 4, pp. 265—288, 1987.

[14] R. Forgaard and J.V. Guttag, "REVE: a term rewriting system generator with failure-resistant Knuth-Bendix," Report, Laboratory of Computer Science, MIT, 1984.

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

[16] P.R.H. Hendriks, *ASF system user's guide*, Report CS-R8823, CWI, Amsterdam, 1988. Extended abstract in *Conference Proceedings of Computing Science in the Netherlands CSN'88 1*, pp. 83—94, SION, 1988.

[17] P.R.H. Hendriks, *Implementation of modular algebraic specifications*, PH.D. Thesis, CWI, Amsterdam, 1991.

[18] J. Heering, P.R.H. Hendriks, P. Klint and J. Rekers, "The syntax definition formalism SDF – reference manual," *SIGPLAN Notices*, 24(11), pp. 43—75, 1989.

[19] J. Heering, P. Klint and J. Rekers, "Incremental generation of lexical scanners," Report CS-8761, CWI, Amsterdam, 1987.

[20] G. Huet and J.-J. Lévy, "Call by need computations in non-ambiguous linear term rewriting systems," Report no. 359, INRIA, Rocquencourt, 1979.

[21] G. Huet and D.C. Oppen, "Equations and rewrite rules, a survey," *Formal Language Theory, Perspectives and Open Problems*, R.V. Book (Ed.), Academic Press, pp. 349—405, 1980.

[22] C.M. Hoffmann and M.J. O'Donnell, "Pattern matching in trees," *Journal of the ACM*, Vol. 29, pp. 68—95, 1982.

[23] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation,* Addison Wesley, 1979.

[24] G. Huet, "A complete proof of correctness of the Knuth-Bendix completion algorithm," *Journal of Computer and System Sciences*, vol. 23, pp. 11—21, 1981.

[25] *Le-Lisp 15.21 reference manual*, INRIA, Domaine de Voluceau, Rocquencourt, France, 1987.

[26] D.E. Knuth and P. Bendix, "Simple word problems in universal algebras," *Computational Problems in Abstract Algebra*, J. Leech (Ed.), Pergamon, pp. 263—297, 1970.

[27] Stéphane Kaplan and Christine Choppy, "Abstract rewriting with concrete operators," *Rewriting Techniques and Applications*, N. Dershowitz (Ed.), Lecture Notes in Computer Science 355, Springer Verlag, pp. 178—186, 1989.

[28] P. Klint, "A meta environment for generating programming environments," Report CS-R9064, CWI, Amsterdam, 1990, to appear in *Proc. of the METEOR workshop on Methods Based on Formal Specification*, Mierlo, the Netherlands, 1989, *Lecture Notes in Computer Science*, Springer Verlag.

[29] P. Klint, "Scanner generation for modular regular grammars," Report, CWI, Amsterdam, 1991. To appear, preliminary version appeared in: *J.W. de Bakker, 25 Jaar Semantiek*, Liber Amicorum, CWI, Amsterdam, pp. 291—305, 1989.

[30] D.E. Knuth, J.H. Morris and V.R. Pratt, "Fast pattern matching in strings," *SIAM Journal of Computing*, Vol. 6, pp. 323—350, 1977.

[31] R.M. Karp, R.E. Miller and A.L. Rosenberg, "Rapid identification of repeated patterns in strings, trees and arrays," *Proc. 4th Annual ACM Symp. on Theory of Computing*, Denver, 1972.

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

[33] P. Lescanne, "Computer experiments with the REVE term rewriting system generator," *Conference record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pp.99—108, ACM, 1983.

[34] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Transactions on Electronic Computers*, Vol. 9, 39—47, 1960.

[35] M.J. O'Donnell, "Computing in systems described by equations," *Lecture Notes in Computer Science*, vol. 58, Springer Verlag, 1977.

[36] M.J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press, 1985.

[37] M.H. Overmars and J. van Leeuwen, "Rapid subtree identification revisited," Tech. Rep. CS-79-3, Univ. of Utrecht, 1979.

[38] Tony Rush and Derek Coleman, "Architecture for conditional term rewriting," *Conditional Term Rewriting Systems*, S. Kaplan, J.P. Jouannaud (Eds.), Lecture Notes in Computer Science 308, Springer Verlag, 266—278, 1988.

[39] N. Wirth, *Programming in Modula-2*, Springer Verlag, 1982.

# Samenvatting

(Summary in Dutch)

In de afgelopen jaren zijn algebraïsche specificaties een steeds grotere rol gaan spelen bij het ontwerpen van programmatuur. Het centrale begrip dat aan deze ontwikkeling ten grondslag ligt is *abstractie*.

Binnen algebraïsche specificaties kunnen bepaalde gradaties van abstractie onderscheiden worden, die elk hun specifieke bijdrage leveren in het programmatuur-ontwikkelingsproces:

1. In eerste instantie kan het opdelen van een probleem in delen – het ontwerpen van een modulaire structuur – een eerste aanzet zijn tot de oplossing van een probleem.

2. Een volgend abstractieniveau is het definiëren van een signatuur voor een gegeven probleem. Dit komt neer op het bepalen van de afzonderlijk te onderscheiden klassen van objecten, en van de diverse functies, met bijbehorende argument- en resultaatklassen. Gereedschappen, in de vorm van programmatuur, zijn voor handen, om de interne consistentie van dergelijke signaturen te verifiëren.

3. Het derde abstractieniveau is de totale algebraïsche specificatie. Aan de functies in de signatuur wordt betekenis gehecht door vergelijkingen over die functies toe te voegen. Men kan dit zien als het formuleren van gewenste eigenschappen. Merk op dat er, in het algemeen, nog steeds sprake is van een hoge graad van abstractie:

    - Er wordt niet vastgelegd om welke objecten het gaat. Elke klasse van objecten die volgens een bepaalde interpretatie de beschreven eigenschappen heeft, is bruikbaar als model voor de specificatie. In het bijzonder wordt niet vastgelegd of, en zo ja, hoe die objecten in een computer gerepresenteerd kunnen worden.
    - Er wordt niet vastgelegd hoe de eigenschappen tot stand komen. Dit is in scherp contrast met, bijvoorbeeld, algorithmische programmeertalen, waar juist wordt beschreven hoe een resultaat tot stand

moet komen, maar niet aan welke eigenschappen dit resultaat voldoet. In de praktijk speelt dit voordeel van algebraïsche specificaties slechts een beperkte rol, daar aan de specificaties allerlei beperkingen worden opgelegd om het automatisch genereren van efficiënte prototypen mogelijk te maken.

Overigens is het niet noodzakelijk om de beschreven objecten concreet te representeren, alvorens men met behulp van een computer berekeningen omtrent die objecten kan uitvoeren. Het is immers mogelijk consequenties van de gegeven vergelijkingen te beschouwen, die dan voor alle modellen gelden. Dit is het terrein van de 'Theorem Proving'.

Helaas is het via deze weg niet mogelijk, of niet praktisch, om alle berekeningen omtrent een specificatie uit te voeren. Soms volgt een bepaald feit niet uit de vergelijkingen, maar geldt het desalniettemin in een voor ons relevant model; soms vergt verificatie van een feit in algemene zin te veel tijd, terwijl het voor een bepaald model direct volgt.

4. In overeenstemming met het laatstgenoemde is het noodzakelijk beperkingen op te leggen aan de klasse van modellen die men wil beschouwen, en aan de vergelijkingen die men toestaat. Uiteraard bestaan er vele verschillende benaderingen, die tot verschillende onderzoeksgebieden kunnen leiden.

   Dit laatste abstractieniveau kan zeker niet als het concrete implementatie niveau gezien worden. We bevinden ons nog steeds op een theoretisch vlak, maar creëren nu de mogelijkheid om met behulp van geschikte programmatuur concrete berekeningen uit te voeren.

In dit proefschrift interesseren we ons voornamelijk voor het laatste abstractieniveau. We beperken ons tot equationele theorieën (de vergelijkingen zijn universeel gequantificeerde, al dan niet conditionele gelijkheden) en we beschouwen alleen initiële modellen. Dit zijn modellen die twee bijzondere eigenschappen hebben:

- In het model komen alleen objecten voor die uitdrukbaar zijn in de gegeven signatuur.
- Alle equationele eigenschappen van het model zijn afleidbaar uit de gegeven vergelijkingen.

Merk op dat hiermee voor een groot deel aan de onder 3 genoemde bezwaren tegemoet is gekomen. Daar alle objecten uitdrukbaar zijn in de signatuur, zijn ze ook representeerbaar in een computer, met behulp van bomen, en daar alle eigenschappen afleidbaar zijn, zijn ze in een equationele theorie ook automatisch afleidbaar door middel van boomsubstitutie.

Overigens bestaat bij dit laatste nog een probleem. Weliswaar zijn eigenschappen automatisch afleidbaar, maar daarmee is niet gezegd dat een willekeurige veronderstelling automatisch verifiëerbaar is. Anders gezegd, om een feit te verifiëren kunnen we proberen het af te leiden, maar zolang dit nog niet gelukt is weten we niet of het nog ooit zal lukken, of dat de veronderstelling onjuist is.

Om dit laatste probleem te ondervangen beperken we ons verder. We interpreteren de algebraïsche specificaties als termherschrijfsystemen, waarbij vergelijkingen alleen in één richting mogen worden toegepast, en we beschouwen alleen termherschrijfsystemen die **confluent** en **terminerend** zijn, waardoor we van alle mogelijke afleidingen er slechts één hoeven te onderzoeken, en waardoor alle afleidingen eindig zijn.

Bovendien beperken we ons in de klasse te verifiëren feiten. We bekijken alleen het al dan niet semantisch gelijk zijn van objecten, of, in het kader van confluente terminerende termherschrijfsystemen daarmee equivalent, het vinden van de normaalvorm van een object. De normaalvorm is een uniek object in een verzameling semantisch gelijke objecten, dat die verzameling ook eenduidig identificeert.

Onder deze voorwaarden kan men binnen eindige tijd elke veronderstelling onomstotelijk bevestigen of ontkennen.

Overigens is het niet vanzelfsprekend dat de beschreven beperkingen nog voldoende uitdrukkingskracht overlaten. Gelukkig is dit wel het geval. Binnen het gegeven kader heeft men bijvoorbeeld een zelfde uitdrukkingskracht als in de meeste programmeertalen, waarbij alleen invoer en uitvoer, en 'real-time' aspecten afwijkend zijn. Merk op dat, strikt genomen, terminatie hier een te sterke eis is, daar een dergelijke eigenschap in de meeste programmeertalen niet gegarandeerd wordt. Onder andere om deze reden zullen terminatie en ook confluentie vooral in praktijk niet dwingend geëist worden, hoewel deze eigenschappen theoretisch nodig zijn om bepaalde resultaten af te leiden. Merk op dat het in het algemeen onmogelijk is om automatisch te beslissen of een termherschrijfsysteem confluent of terminerend is.

Bij het ontwikkellen van programmatuur heeft implementatie van algebraïsche specificaties middels termherschrijfsystemen globaal twee toepassingen:

- Aan de hand van een specificatie wordt de functionaliteit van de beoogde programmatuur getest, waarbij eventueel een prototype gebruikt wordt. Zodra de specificatie naar wens is, wordt met de hand een implementatie in een of andere programmeertaal gemaakt, waarbij men probeert het gedefiniëerde gedrag te implementeren. De specificatie wordt als gereedschap gebruikt, maar formeel is er geen verband tussen specificatie en implementatie.

- Aan de hand van de specificatie wordt automatisch de uiteindelijke implementatie gegenereerd. De specificatie is nu bouwsteen in plaats van gereedschap. Uiteraard is deze benadering een speciaal geval van de vorige, maar nu dienen zich twee nieuwe vragen aan: automatische generatie, en uiteindelijke efficiëntie van de implementatie. Het is voor het eindproduct immers van groter belang hoe de bouwstenen tot stand komen, en met welke efficiëntie ze werken, dan wat de gebruikte gereedschappen zijn.

In ons onderzoek worden systemen ontwikkeld die zich voornamelijk richten op deze tweede benadering. Aan de hand van specificaties worden automatisch geïntegreerde implementaties gegenereerd.

Helaas is efficiëntie een zwak punt van op termherschrijving gebaseerde programmatuur. Indien men de snelheid van zulke programmatuur vergelijkt met die van direct geïmplementeerde programmatuur, dan moet men toch met een factor van honderden tot duizenden rekenen, ten nadele van termherschrijving.

In dit proefschrift zal daarom aandacht besteed worden aan technieken die de efficiëntie van termherschrijving verhogen. Efficiëntie wordt bepaald door twee grootheden: het aantal uit te voeren herschrijfstappen, en de gemiddelde tijd nodig voor een herschrijfstap. Met hybride specificaties, besproken in hoofdstukken 1, 4 en 5, wordt beoogd de eerste grootheid te reduceren, terwijl met 'tree-scanners', besproken in hoofdstukken 1 en 3, de tweede grootheid gereduceerd wordt.

Dit onderzoek is uitgevoerd in het kader van het door de EEG (ESPRIT) gesubsidiëerde project GIPE (Generation of Interactive Programming Environments) waarin onder andere het ASF+SDF systeem ontwikkeld is. Dit systeem is een programmeeromgevingsgenerator, waarmee aan de hand van formele specificaties diverse interactieve, geïntegreerde gereedschappen gegenereerd kunnen worden, zoals bijvoorbeeld syntax-gerichte tekstverwerkers.

Hoofdstuk 1 van dit proefschrift gaat over de Equation Manager, de component uit dit systeem die implementaties van termherschrijving genereert. In dit hoofdstuk presenteren we eerst een formele definitie van conditioneel termherschrijven, en vervolgens schetsen we een implementatie ervan, zoals deze ook in de EQM gebruikt is. Tenslotte bekijken we drie bijzondere technieken die in de EQM zijn toegepast, maar die men niet, of niet in deze vorm in andere termherschrijfimplementaties kan vinden. Twee van de drie technieken bespreken we zeer gedetailleerd in latere hoofdstukken.

In Hoofdstuk 2 bekijken we de theorie van het termherschrijven nader. Vervolgens beschouwen we enkele bestaande implementaties van termherschrijving die gebaseerd zijn op Prolog. Een en ander resulteert in een compact overzicht van reductiestrategieën. Tenslotte besteden we enige aandacht aan de efficientie van Prolog implementaties.

Het herhaaldelijk analyseren of ergens in een gegeven term één van de vergelijkingen kan worden toegepast, kost het meeste tijd in implementaties van termherschrijving. In Hoofdstuk 3 beschrijven we een techniek waarmee een grote hoeveelheid 'dubbel werk' wordt uitgespaard, door deze analyse voor alle vergelijkingen quasi-parallel uit te voeren.

In Hoofdstuk 4 beschrijven we een techniek waarmee men binnen het kader van termherschrijving gebruik kan maken van programma's in conventionele programmeertalen. Zo kan men voor basis datatypen direct gebruik maken van de onderliggende computer machinerie, daarmee vele termherschrijfstappen vermijdend, zonder dat de correctheid of de algebraïsche context worden aangetast.

In Hoofdstuk 5, ten slotte, geven we een uitgebreid voorbeeld van de in Hoofdstuk 4 besproken methode. Aan de hand van dat voorbeeld worden diverse metingen van de executiesnelheid verricht, waarbij steeds grotere delen van de voorbeeldspecificatie geïmplementeerd worden in een conventionele programmeertaal.

Diverse hoofdstukken uit dit proefschrift zijn afzonderlijk gepubliceerd. Daardoor bestaat er enige mate van overlap. Waar mogelijk is door middel van het symbool $\bowtie$, bij een sectietitel, aangegeven dat de inhoud reeds eerder in enige vorm behandeld is. Zulke secties zijn niet essentieel voor begrip van het er op volgende, indien men het proefschrift van voor naar achter leest.