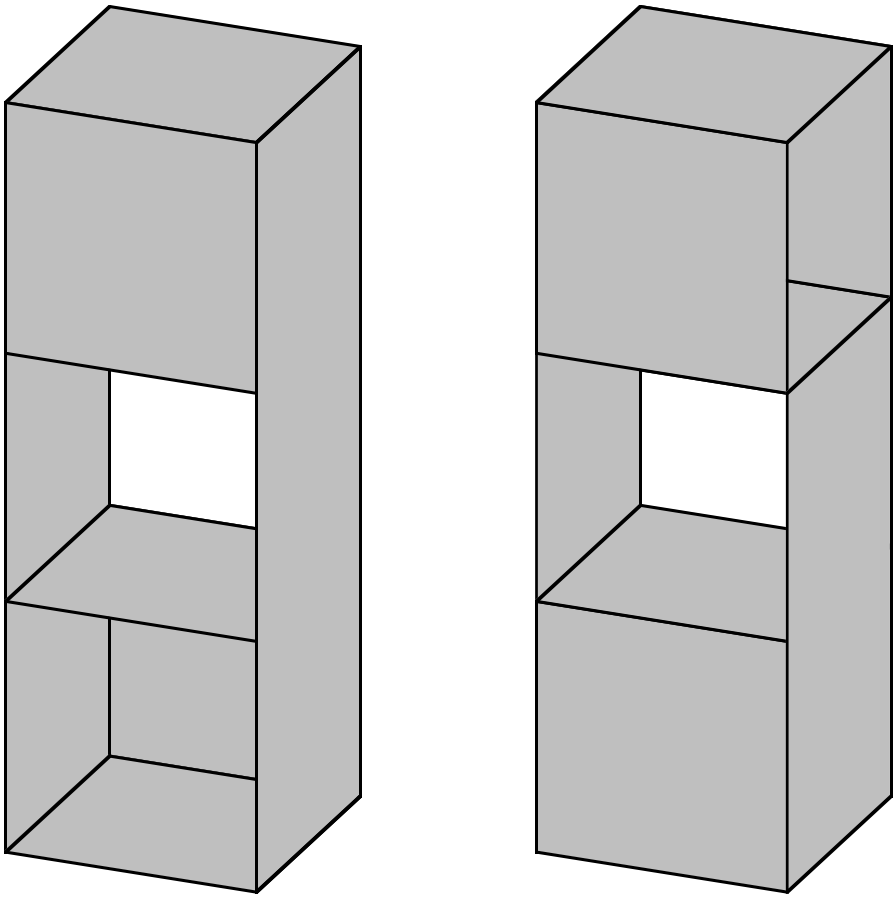
Persistence in Algebraic Specifications



Freek Wiedijk

Persistence in Algebraic Specifications

Persistence in Algebraic Specifications

Academisch Proefschrift

ter verkrijging van de graad van doctor aan
de Universiteit van Amsterdam, op gezag van
de 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 donderdag 12 december 1991 te 12.00 uur

door Frederik Wiedijk

geboren te Haarlem

Promotores: prof. dr. P. Klint & prof. dr. J.A. Bergstra
Faculteit Wiskunde en Informatica

Het hier beschreven onderzoek werd mede mogelijk gemaakt door steun van de Nederlandse organisatie voor Wetenschappelijk Onderzoek (voorheen de nederlandse organisatie voor Zuiver-Wetenschappelijk Onderzoek) binnen project nummer 612-317-013 getiteld *Executeerbaar maken van algebraïsche specificaties*.

voor Jan Truijens

Contents

Acknowledgements

My greatest debt is to Paul Klint, my promotor. I admire his excellent taste, and I consider it a great honor that he allowed me to work under his supervision.

I want to thank Jan Bergstra, my second promotor, for interesting and stimulating discussions. His ability to transform himself into an expert, whatever the subject may be, never ceases to amaze me.

My friend Jesse de Does has been a great support from the start. Despite his unfamiliarity with the field of algebraic specifications, he is one of the few people who read all my work and understood it, in spite of my hermetic style. I am indebted to him for numerous corrections.

My roommate and friend Sjouke Mauw has been a great help on many occasions. He has always showed great interest in my work, and never failed to point out where my texts were less than clear.

Of all people working near me I feel closest intellectually to my friend Hans Mulder. He is a mathematician as well as a real hacker. He has often shown me where to look for excitement. His knowledge of computer science, in particular its folklore, is unsurpassed.

I finally would like to express my thanks to the following people, who made the four years that I worked at the University of Amsterdam an unforgettable experience. They are: Pum Walters, Jos Vrancken, Chris Verhoef, Gert Veltink, John Tromp, Leen Torenvliet, Dolf Starreveld, Joost Schraag, Piet Rodenburg, Karst Koymans, Wilco Koorn, Jan Willem Klop, Jan Hellings, Jan Heering, Linda van der Gaag, Peter van Emde Boas, Madelon Drolsbach, Casper Dik, Felix Croes, Richard Carels, Maarten Carels, Jacob Brunekreef, Jeroen Brouwer, Wiet Bouma, Jurjen Bos, Chris Borton and Jos Baeten.

The picture on the cover shows two views on the one missing configuration (3L 2FB 3F) in Sol LeWitt's *All three-part variations on three different kind of cubes*, an ink drawing from 1960 that I saw on the exhibition of LeWitt's work held in the Stedelijk Museum in Amsterdam in 1984. The mottos in front of the chapters are taken from various articles and letters received from Usenet.

Chapter 1

Introduction

If it's not correct, it's worthless.
I would even agree that if it does not have rigorous definitions, it's worthless.

This thesis investigates some aspects of the notion of persistence in the context of modular initial algebra specifications. A specification is called persistent when the sets of elements of the specified datatypes do not depend on the module in which they are referred to (for the precise definition of persistence, refer to section 2.5.1.2).

The three main goals of this study are:

- To argue that verification of the persistence of an algebraic specification can enhance our confidence in the correctness of that specification, and to give an approach for verifying a form of persistence automatically.
- To show that a specification can be implemented using logic programming in such a way that the implementation consists of the natural definition of the semantics of that specification.
- To show that a persistent specification can be implemented in a modular procedural programming language in such a way that the modularity of the implementation corresponds to the modularity of the specification.

This thesis, then, consists of four parts.

- Chapter 1 contains an introduction that motivates the use of modular initial algebra specifications and the study of persistence.
- Chapter 2 gives a formal treatment of the theory of initial algebra specifications and techniques that can be used to verify the persistence of those specifications.
- Chapter 3 gives a concrete framework for checking persistence in the form of the specification language Perspect. A number of examples in this language are presented.
- Chapter 4 shows that persistent specifications can be cleanly integrated with both a logic programming language and a modular procedural programming language. Algorithms for compiling Perspect specifications to both Prolog and Modula-2 programs are described.

Two specification languages will be used for the expression of algebraic specifications. The first one is *ASF*, which is an acronym for 'Algebraic Specification Formalism'. It was developed in Amsterdam as part of the specification language for the GIPE project. For the definition of ASF, see chapter 1 of [Bergstra, Heering, Klint, 1989]. The

second specification language that is used is *Perspect*, a dialect of ASF, which is the specification language introduced in this thesis. It will be the subject of chapter 3.

1.1. Motivation and general overview

In this thesis we study the theory of initial algebra specifications. Specification languages that use initial algebra semantics have both the characteristics of a formal specification language as well as those of a programming language. This dualistic nature of a language that uses initial algebra semantics is both its weakness and its strength.

1.1.1. Simplicity of semantics. The main advantage of a language that uses initial algebra semantics, an advantage it shares with most other formal specification languages, is its clean semantics. Programming languages on the contrary, are almost always ridden with special cases in the definition of their semantics, and often contain aspects that are left unspecified or refer to system dependent parameters. Unfortunately this departure from an abstract model behind the semantics of the language is necessary when one defines a programming language: even the most formal of programming languages has to give some description of what should happen when there is, for instance, not enough storage in the computer to run the program. A specification language, of course, need not bother with such problems.

For a rather extreme example of the undefinedness and system dependency of texts in conventional programming languages, consider the following program in the C programming language [Kernighan, Ritchie, 1988].

```
#include <stdio.h>
int
main()
{
  long l;
  int i;
  short s;
  char c;
  i = 65537;
  l = i * i;
  for (s = 0; s < l; s++)
    ;
  c = 255;
  if (c == -1)
    (void) puts("0");
  else
    (void) puts("1");
  return 0;
}
```

This text is a valid C program. However, when someone tries to compile this program and subsequently tries to run it, there are (at least) five possibilities:

(i) The compiler refuses to compile the program.
(ii) The program compiles, but crashes when it is run.
(iii) The program compiles and does not crash, but will not terminate.
(iv) The program compiles, does not crash, and terminates after printing '`0`'.
(v) The program compiles, does not crash, and terminates after printing '`1`'.

Furthermore, for all five cases, the compiler and runtime system can justify why they behave like they do. It all depends on how many bits there are in the various kinds of integer (which is a system dependency of the compiler), and what happens on integer overflow (which is an undefined aspect in the C programming language). The most natural outcome of this program, in the sense that overflow is something we do not want to happen, is that this program prints '`1`' (this is case (v)). However, when we compile this program using a contemporary compiler, it probably will not terminate when we run it (which is case (iii)).

Admittedly, a language like C is a bit extreme in its system dependencies and undefinednesses, but even a clean functional language like Scheme has many special cases in its definition, and leaves a large part of its semantics undefined.

In contrast with this, the semantics of an initial algebra specification is extremely elegant and simple. It has no special cases, and needs no primitive datatypes. Even a simple datatype like the Boolean truth values can be constructed *within* the formalism, and is not needed as a primitive in the definition of the semantics. Because of this simplicity of semantics, initial algebra specifications avoid a lot of the problems that are usually encountered when writing programs in conventional programming languages.

1.1.2. Executability. Another advantage of a language with initial algebra semantics, one that it shares with programming languages, is the possibility of executability. It turns out that it is easy to write initial algebra specifications in such a way that they can be executed as a term rewriting system. This makes it possible to use the specifications as a, albeit not very efficient, prototype of the system it specifies.

However, one should realize that it is not always the case that an initial algebra specification can be executed as a term rewriting system. (If one does not mind inefficiency, an algebraic specification *can* always be executed in some sense by performing an exhaustive breadth first search, but this is generally considered not to be a viable alternative.) This means that only a subclass of the initial algebra specifications can be executed as a term rewriting system. One can argue that specifications that are written to be executable as a term rewriting system, are often not the most simple or natural way to describe something.

As an example, suppose that we try to specify the sorting of a list of elements from some datatype, using the specification language ASF.

Suppose that there are given two auxiliary modules called `Booleans` and `Lists`, which specify the Booleans and the lists over some datatype `ITEM` that has some inherent total order given by the operator `.less-or-equal.`. These modules are omitted here for brevity, except the export section of module `Lists` which is:

```
exports
 begin
  sorts LIST
  functions
   empty-list:              → LIST
   ++ _       : ITEM        → LIST
   _ ++ _     : ITEM × LIST → LIST
   _ ++ _     : LIST × ITEM → LIST
   _ ++ _     : LIST × LIST → LIST
 end
```

Clearly lists are constructed using the (overloaded) operator ++. This operator is used to construct a one-item list from some item, and to concatenate lists and items to longer lists.

Now, a natural specification of sorting, one that is unfortunately not executable as a term rewriting system, is given by the following module:

```
module Sorting
 begin
  exports
   begin
    functions
     sort: LIST → LIST
   end
  imports Booleans, Lists
  sorts BAG
  functions
   bag:    LIST → BAG
   sorted: LIST → BOOL
  variables
   i, j: → ITEM
   l, m: → LIST
  equations
   [1] bag(l ++ i ++ j ++ m)     = bag(l ++ j ++ i ++ m)
   [2] sorted(empty-list)        = true
   [3] sorted(++ i)              = true
   [4] sorted(l ++ i ++ j ++ m) = sorted(l ++ i) .and.
                                   (i .less-or-equal. j) .and.
                                   sorted(j ++ m)
   [5] sort(l) = m when bag(l) = bag(m), sorted(m) = true
 end Sorting
```

Module `Sorting` defines sorting in three steps:

- First a datatype called `BAG` is specified. This datatype is defined by equation `1` to be a copy of the datatype called `LIST`, in which lists that only differ in the order of

their elements are identified.
- Second, a predicate called `sorted` is defined that tells whether a list is already sorted.
- Finally, for some list `l`, the sorted list `sort(l)` is defined to be the unique list `m` that is sorted, and that consists of the same elements as `l`, i.e., that has the same associated bag as `l`.

There exists another specification of sorting that, in contrast to the previous specification, has the advantage that it can be executed as a term rewriting system. The reason for the fact that this specification is executable is that it does not give the abstract notion of sorting, but instead gives some specific sorting algorithm:

```
module Sorting′
 begin
  exports
   begin
    functions
     sort: LIST → LIST
   end
  imports Booleans, Lists
  functions
   _ .select-small. _: LIST × ITEM → LIST
   _ .select-large. _: LIST × ITEM → LIST
   sort-and-append:    LIST × LIST → LIST
  variables
   i, j: → ITEM
   l, m: → LIST
  equations
   [1] empty-list .select-small. j = empty-list
   [2] (i ++ l)   .select-small. j =
                  if(i .less-than. j, (i ++ m), m)
                  when l .select-small. j = m
   [3] empty-list .select-large. j = empty-list
   [4] (i ++ l)   .select-large. j =
                  if(i .less-than. j, m, (i ++ m))
                  when l .select-large. j = m
   [5] sort-and-append(empty-list,m) = m
   [6] sort-and-append((i ++ l),m) =
                  sort-and-append(l .select-small. i,
                  i ++ sort-and-append(l .select-large. i, m))
   [7] sort(l) = sort-and-append(l,empty-list)
 end Sorting′
```

The intended meaning of the expression '`sort-and-append(l,m)`' is 'first sort list `l`, and then append list `m` after it'.

The algorithm that is used in this specification is a simple version of the quicksort

algorithm (which [Knuth, 1973] calls 'partition-exchange sort'. Note though, that due to the functional nature of the specification, no 'exchanges' take place in this version of quicksort.) To enhance the efficiency of the algorithm, it is expressed in a tail-recursive way.

It can be argued that one should not give a specification of an algorithm for sorting, when a specification of the concept of sorting is required. One argument for this point of view is the observation that an algorithm is in general not as readable as a conceptual exposition. Note however, that the algebra that is specified by module `Sorting'` does not differ from the algebra that is specified by module `Sorting`. So, although the specification uses a specific algorithm to specify what should be done, it is not specified that this algorithm should be used when implementing the specification!

1.1.3. Problems. There are two – clearly related – classes of problems with the usage of initial algebra specifications:

- It is too difficult to write an initial algebra specification.
- Most of the initial algebra specifications that can be found in the literature are incorrect: the meaning of those specifications differs from the meaning that was intended by the author of the specification.

Note that the two problems are not the same. For example, in the empty language it is even more difficult to write down something. However, for the empty language it is obvious when a text is not correct. On the other hand, as is well known today, almost all texts in a conventional programming language are somewhat incorrect ('contain bugs'). Still, when writing a program in a conventional programming language, one generally feels less restricted than when writing an initial algebra specification.

In this thesis, we only work on the solution of the second problem, and, after this section, we will not be concerned with the first problem. However, our 'solution' makes the first problem worse. While the use of the techniques that are proposed here reduces the chance that an error in a specification will go undetected, it will make it much harder to write a specification at all.

While it is not the goal of this thesis to offer ideas for making it easier to write algebraic specifications, this is of course a fascinating topic. One can conceive two approaches for solving this problem: both lead to giving up the middle position between specification language and programming language, and move to one end of the spectrum between formal specification and program.

The first approach consists of adding more expressive power to the language. For example, one can replace the conventional conditional equational logic of initial algebra semantics by first order predicate logic. When one takes this step, it clearly becomes easier to write specifications because one has now more tools at one's disposal. Also, due to this greater expressive power, the specification can be more intuitive, and because of that, we can hope that the chance that one makes an error is reduced. However, the disadvantage of this approach is that a specification that uses the additional expressive power probably will not be executable.

The second approach consists of replacing the simple initial algebra semantics by a more operational semantics that is closer to the semantics of a programming language. Examples of this are the addition of priorities to term rewriting systems, or the auto-

matic handling of 'error' elements. The disadvantage of this approach is that one gets less nice semantics. For example, one tends to loose the property that every syntactically valid text in the language has a meaning. Or, even worse, one can loose the property that it is decidable whether a specification has a meaning at all.

1.1.4. Erroneous specifications. We will now look at the problem that most specifications from the literature are incorrect. A number of examples to demonstrate this point will be presented in section 1.2.

It is interesting to note that the notion of incorrectness has no formal significance. When a specification does not specify the intended algebra, it is not objectively incorrect but only in relation to the – informal – intention. If one would try to remedy this informality by giving a 'meta' specification, it is not clear what the use of the 'normal' specification is. And even if one gives a meta specification, then that meta specification can again be in error.

The problem that it is not clear what the term 'correct' exactly means – namely either: valid according to the formal definition of the system, or: correct according to the intention of the author of the specification – has confused some people who were using the interim ASF type checker. This type checker checked whether a specification was valid according to the definition of ASF, i.e., it checked whether a specification did specify *an* algebra at all. If this turned out to be the case, the type checker printed:

```
--- specification correct
```

Some people interpreted this output to mean that the specification reflected correctly their intentions, which was generally not the case.

It is not the case that incorrectness of a specification can always be easily discovered by executing the specification as a term rewriting system. For example, consider the following specification of stacks, in which the problem of what happens when one pops from an empty stack is solved by having 'error elements' in the algebra (a supposedly correct specification that uses this idea can be found as the first specification in section 3.3.4).

```
module error-Stacks
 begin
  parameters Items
   begin
    sorts ITEM
    functions
     error-item: → ITEM
   end Items
  exports
   begin
    sorts STACK
    functions
```

```
    empty-stack:                → STACK
    error-stack:                → STACK
    push:          ITEM × STACK → STACK
    top:           STACK        → ITEM
    pop:           STACK        → STACK
  end
 variables
  i, j: → ITEM
  s, t: → STACK
 equations
  [1] push(top(s),pop(s)) = s
  [2] top(push(i,s))      = i
  [3] pop(push(i,s))      = s
  [4] top(empty-stack)    = error-item
  [5] pop(empty-stack)    = error-stack
  [6] push(error-item,s)  = error-stack
  [7] push(i,error-stack) = error-stack
  [8] top(error-stack)    = error-item
  [9] pop(error-stack)    = error-stack
end error-Stacks
```

This specification is incorrect. One can prove that all elements of `ITEM` are equal to each other by:

```
i                          = [2]
top(push(i,error-stack))   = [7]
top(error-stack)           = [7]
top(push(j,error-stack))   = [2]
j
```

In the same fashion, equality of all elements of `STACK` can be proved:

```
s                          = [3]
pop(push(error-item,s))    = [6]
pop(error-stack)           = [6]
pop(push(error-item,t))    = [3]
t
```

However, this specification will always behave acceptably when executed as a term rewriting system. The incorrectness of the specification is reflected by the fact that the corresponding term rewriting system is not confluent, which means that a number of terms has more than one normal form. However, each of those normal forms is a plausible reduct.

For example, one can reduce:

```
push(top(empty-stack),pop(empty-stack))
```

to `empty-stack` but also to `error-stack`. Both results are plausible. Note that it is on the other hand not possible to reduce:

```
push(error-item,error-stack)
```

to `empty-stack`, which would be strange.

As this example shows, 'debugging' an executable specification by executing it as a term rewriting system will not always be sufficient to find all specification errors in the specification. It will be necessary to look for other ways to uncover errors in a specification.

1.1.5. Checking persistence. As will be shown by the examples in section 1.2 specification errors often lead to impersistent specifications (for the definition of persistence, see section 2.5.1.2). For example, in most incorrect specifications, the sort of the Booleans, that should contain only the two elements 'true' and 'false', will somehow be damaged. This may either mean that 'true' has become equal to 'false', or that there exists some element different from both 'true' and 'false', or even both anomalies can occur at the same time. In a sense, one can call a specification in which the sort of the Booleans turns out to contain only one element because all Booleans are identified, *inconsistent*. However, the Booleans have no special status within the framework of initial algebra semantics, so one can often better use a more intrinsic concept, and call instead such a specification *impersistent*.

It might be argued that impersistent specifications have some applications (for a rather extreme form of this point of view, see the last chapter of [Mulder, 1990]). There are situations in which the most elegant specification of some algebra is not persistent. For example, consider the following Perspect specification of the integers modulo 7.

```
external N
 sort N
 function 0, S(N): N
internal N
 [empty]

external 7a
 import N
internal 7a
 variable n: N
 equation
  S(S(S(S(S(S(S(n)))))))= n
```

In module `N` one first specifies the natural numbers. After that, the equation (in module `7a`):

```
S(S(S(S(S(S(S(n))))))): n
```

which should be read as:

```
n + 7 = n
```

is sufficient to convert this specification to a specification of the integers modulo 7.

There is however a simple way to obtain the same effect, without sacrificing persistence, if we replace module `7a` by:

```
external 7b
 import N
 sort 7
 function i(N): 7
internal 7b
 variable n: N
 equation
  i(S(S(S(S(S(S(S(n)))))))): i(n)
```

Instead of changing the sort of the natural numbers we now first make a copy called `7` of this sort (with the 'identification' function `i`), and make the changes local to that copy. Module `7b` is almost equal to module `7a`, and has the advantage that it satisfies persistence. Furthermore, module `7b` can be considered to be clearer, because it has been made explicit that a sort is being modified here.

This example is simple, but the same approach can be applied in general to eliminate impersistencies that were made intentionally in algebraic specifications. If some module modifies a sort in some way, one can generally avoid the impersistence that is introduced in this way by creating a new sort in which the modifications are made. This shows that even though there may be cases in which an impersistence in a specification may seem to be advantageous, in practice that kind of impersistencies can be eliminated in a simple way, and without much disadvantage.

So, assuming that persistence of specifications is a desirable property, it would be nice to have some automated support for the creation of persistent specifications.

One kind of 'support' for persistence in algebraic specifications is to require in the definition of a specification language that imports between modules should be persistent. However, one could as well postulate that specifications should be correct (in the sense that the specified algebra is equal to the algebra that was intended by the author of the specification) and avoid the detour of the notion of persistence. Simply postulating a requirement does not help much.

Instead, one wants not only to claim the persistence of the specification, but also to have some way to verify this claim. As persistence is not a decidable property, this cannot be done in general. One cannot have a system in which one can formulate precisely the persistent algebraic specifications, and in which it is decidable whether a specification is valid according to the definition of the system.

In this thesis we solve this dilemma by not opting for the full class of persistent

algebraic specifications, but only for some decidable subclass. This is analogous to the restriction in recursion theory of the class of all total recursive functions to the primitive recursive functions, in order to get a decidable subclass.

It turns out that the way we restrict the class of persistent specifications leads to a class of specifications that can be efficiently executed as a term rewriting system. That persistence is related to executability is not really surprising. In fact, a persistent specification has a natural notion of execution. This means that persistence implies executability of the specification according to this notion.

Consider some module M_1 which introduces some sort σ. Furthermore, consider also some other module M_2 that imports M_1, and some term t of type σ, over the signature of M_2. Now, a term t′ can be defined to be *a result of an execution* of t when t′ is a term over the signature of M_1, and when t′ can be proved to be equal to t. If the import of module M_1 by module M_2 is persistent, such a term t′ can (by the definition of persistence) always be found.

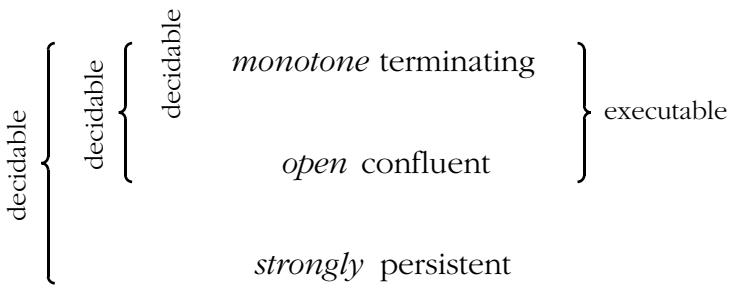
A nice property of the way we check persistence in this thesis is that in the subclass of specifications that comply with our constraints, the notion of executing a term as defined in the previous paragraph coincides with the notion of executing it by rewriting it in the specification considered as a term rewriting system.

So, we are looking for a specification system that satisfies the following three properties:

- It is decidable whether a specification is valid according to the definition of the formalism.
- All specifications that are valid according to the formalism are persistent.
- All specifications that are valid according to the formalism are executable as a term rewriting system.

We will satisfy this set of requirements as follows. We start with the class of unconditional specifications, considered as a term rewriting system. We restrict this class by only admitting specifications that satisfy a strong form of termination, a strong form of confluence, and a strong form of persistence. In order to prove decidability, we also need the requirement of left linearity (which will be needed in the theory developed in section 2.5).

The relation between the requirements of termination, confluence and persistence on the one hand and decidability and executability on the other hand is shown in the following figure:



This figure means that:

- The strong form of termination that we require is a decidable property of the specification.
- Given that the specification satisfies this strong form of termination, the strong form of confluence that we require is also decidable.
- Given that the specification satisfies both the strong forms of termination and confluence, the strong form of persistence that we require is again decidable.
- Given that a specification is terminating and confluent, i.e., given that it is a *complete* term rewriting system, that specification is executable as a term rewriting system.

We will define in chapter 3 a language called *Perspect*. It will be a concrete representation for the decidable class of specifications that we sketched in the previous paragraphs. Some arguments for the introduction of a special language for this specific purpose can be found in the next section.

One must realize that the price that we pay for decidability is high. Because of the way we verify the persistence of a specification, it is hard to write a specification that satisfies all restrictions. In this way, Perspect resembles the empty language, and in fact the empty language also satisfies the three requirements that Perspect was meant to satisfy. The empty language is clearly decidable, and if no text is valid, all texts are persistent and executable.

However, the restrictiveness of Perspect is partly illusory. Often the checks of Perspect fail, not because they are so heavy, but because it is simply hard to find an initial algebra specification of the desired algebra at all.

1.1.6. Yet another specification language. One can ask whether the proposed approach to checking persistence should be accompanied by a new language. Was it not possible to reuse an old language, and add the persistence check to it? Surely the world does not need yet another specification language!

The requirement of persistence is not unique for Perspect. Other languages also have modularisation constructions that are defined to imply persistence of the corresponding imports. This means that the fact that Perspect specifications are by definition persistent is not reason enough to introduce a new formalism.

One good reason for not using an existing formalism is a practical one. Most formalisms have a lot of constructions that were considered useful at the time the formalism was introduced. However, with Perspect we want to focus on the persistence check without being distracted by irrelevant details (at least, irrelevant to the issue of persistence). This means that Perspect is, in contrast to most other formalisms, a *minimal* formalism. The only features of Perspect that depart from the real minimal specification formalism, namely the **`echo`** and **`rec`** declarations, concern specifically the persistence check. Another way the minimality of Perspect expresses itself is in the minimality of Perspects modularisation constructions and lexical syntax. Both aspects of Perspect were specifically designed to make Perspect easily translatable to Modula-2.

Because Perspect is a dialect of ASF, we will now compare Perspect with ASF in some detail. As we said, Perspect is a minimal language as compared to ASF. Features

of ASF that are not present in Perspect are:
- conditional equations
- parametrised modules
- a built-in if function
- tuples
- renaming of imported objects
- overloading of function names
- operators
- non-alphanumerical characters in identifiers

In the design of Perspect some of the experience gained from working with ASF has been applied. This means that Perspect has a number of improvements over ASF. Some of these improvements are:
- Perspect has a less restricted form of hiding. It is possible to import a module without automatically exporting the signature of that module.
- Perspect has a free order between the various components of a module. It is not necessary to write down sorts, functions, variables and equations in that order, but these objects may be mixed freely. This makes it possible to group the declaration of a function with the equations describing it without having to introduce a separate module. The illogical order of imports and exports in ASF has been eliminated.
- Perspect is more orthogonal in its relations between the different parts of a module. For example, in Perspect variables can be exported.
- Perspect has unnamed variables, like the variables called '_' in Prolog.
- Perspect is more compact than ASF (often a Perspect specification is only half as long as the corresponding ASF specification). Perspect has a couple of abbreviations, which make it possible to reduce the number of declarations and equations drastically.
- Perspect has a simpler comment convention than ASF. In Perspect it is possible to remove a part of a specification temporarily by putting it between comment brackets.

The most striking difference between Perspect and the various other specification formalisms is that Perspect has the syntactic means to indicate a specific termination ordering in conjunction with the specification. This termination ordering is given by the order of the function declarations in the specification by using the **`echo`** and **`rec`** declarations and by marking some of the function argument positions as 'inductive' (or, in path ordering terminology, 'lexical') by prefixing them with an asterisk in the function declaration. The semantics of a Perspect specification does not depend on these syntactic elements; they are only used to verify the persistence of the specification.

1.2. Examples of erroneous specifications

In this section we will look at four examples of errors that occurred in real specifications. The errors were taken from the original version of the PICO specification (as it appeared in [Bergstra, Heering, Klint, 1985]), and from the ADT part of the examples

in [Bergstra, 1988]. Both specifications were written in the ASF formalism, although the version of the PICO specification studied here is written in an old dialect of ASF and although the ADT specifications in [Bergstra, 1988] have a somewhat deviant syntax.

We try to show that most of the errors that we will study in this section destroyed the persistence of the specification they occurred in. This implies that an attempt to prove the persistence of these specifications would have pointed out the majority of these errors. In other words, the application of a persistence check to these specifications would have been a valuable heuristic for finding some of these errors.

1.2.1. Converting strings to integers. The first error that we will look at, occurs in the PICO specification when one tries to convert a string (e.g. the string '`12345`') to an integer (which should in this case be the integer 12345).

In order to understand the context of the error, we first have to look at the aim of the PICO specification in some detail. The PICO specification intends to specify the semantics of a small programming language called PICO. This goal is attained by specifying a function called `run`. This function has the sort `STRING` as its domain, and the sort `PICO-VALUE` as its range. A `PICO-VALUE` is either an `INTEGER`, a `STRING` or an `error-value`.

As an example, consider the string `s` that is defined as:

'**begin declare** output: **integer**; output := 12345 **end**'

The term `run(s)` should then be equal to the term `pico-value(i)` where `i` is the integer 12345.

Clearly, at some point in the specification the transition of the substring '`12345`' of `s` to the integer 12345 that occurs in the value of `run(s)` has to be made. In order to do this, the PICO specification contains the module `Strings`, which exports the function `str-to-int` by means of the following export section:

```
exports
   begin
     functions
        str-to-int : STRING → INTEGER
   end
```

The meaning of the function `str-to-int` is then specified using the following two equations:

```
variables
     c        :→ CHAR
     str      :→ STRING
equations
[120] str-to-int(seq(c, str)) =
```

```
                        if(eq(str, null-string),
                          ord(c),
                          add(mul(ord(c), 10), str-to-int(str)))
[121] str-to-int(null-string) = 0
```

However, elaboration of an example will show that these equations are incorrect. This can be seen when we use this definition to convert the string '12345' to an integer, because the value that we get will be:

$$1 \cdot 10 + 2 \cdot 10 + 3 \cdot 10 + 4 \cdot 10 + 5 = 105$$

instead of:

$$(((1 \cdot 10 + 2) \cdot 10 + 3) \cdot 10 + 4) \cdot 10 + 5 = 12345$$

Now, what are precisely the consequences of this error for the PICO specification? And specifically, can the PICO specification still be persistent when it contains this error? If the error would make the persistence of the specification impossible, a persistence check on the specification should make the error visible.

Unfortunately, this error cannot be detected by verifying the persistence of the specification, because it does not disturb the persistence at all. Although the function `str-to-int` has a somewhat misleading name, it is a perfectly acceptable function as it is defined here. Also, there are no serious consequences for the rest of the specification caused by the use of this function, because the only equation in which the function is used is equation `338` in module `PICO-concrete-syntax`:

```
[338] build(integer-constant, env)
      = pico-atree(op-integer-constant,
                   integer-pico-atree(str-to-int(str)))
      when lexical-pico-atree(token("integer-constant", str))
                      = "i" ^ env
```

This is the point where the substring '12345' is evaluated as having the (erroneous) value 105. It will be clear that the error in the specification of `str-to-int` has no disastrous consequences in this equation.

This first example of an error in a specification shows that there exists a class of errors that will not be caught by checking the persistence of a specification. This class of errors strongly resembles the errors in conventional computer programs that are traditionally denoted by the word 'bugs'. This is not remarkable, because a conventional computer program is always persistent (in the algebraic sense), so all errors in a conventional program have to be of this type.

One can hope that the nature of algebraic specifications leads to a style of writing specifications that is less prone to 'bugs' than conventional programming languages. However, there is no real support for this hope.

1.2.2. Calculating the answer of a nonterminating program. The second error that we will look at, is also an error from the PICO specification. It occurs for example when one tries to evaluate the value of the term `run(s)`, where `s` is the string:

```
'begin
  declare true: integer;
  true := 1;
  while true do od
end'
```

The function `run` is specified in the final module of the specification: module `PICO-system`. The export section of this module looks like:

```
exports
     begin
        functions
             run:    STRING  → PICO-VALUE
     end
```

The function `run` is specified using two auxiliary functions `run1` and `run2`. The relevant part of module `PICO-system` is:

```
functions
     run1: PICO-ATREE        → PICO-VALUE
     run2: PICO-PROGRAM      → PICO-VALUE
variables
     s       : → STRING
     p       : → PICO-ATREE
     abs-prog: → PICO-PROGRAM
     has-output: → BOOL
     v       : → PICO-VALUE
     env     : → VALUE-ENV
equations
[369] run(s)                 = run1(parse-and-construct(s))
[370] run1(error-pico-atree) = error-value
[371] run1(p)                = if(check(pico-program(p)),
                                  run2(pico-program(p)),
                                  error-value)
[372] run2(abs-prog)         = if(has-output, v, error-value)

                    when program-state(env) =
                                 eval(program-state(abs-prog)),
                         <has-output, v> =
                                 lookup("output", env)
```

These equations create the impression that the following four cases are distinguished:
- If the string `s` is not a syntactically valid PICO program, the value of `run(s)` is `error-value`.
- If the string `s` is syntactically valid, but is not correctly typed, again the value of `run(s)` is `error-value`.
- If the string `s` is syntactically valid, and correctly typed, but execution of the program does not have output (e.g. because the program does not have a variable called `output`), again the value of `run(s)` is `error-value`.
- Finally, if `s` is correct, and terminates with output `v`, the value of `run(s)` is `v`.

In the case that the execution of the program corresponding to `s` does not terminate at all, the preceding paragraph suggests that the value of `run(s)` is equal to `error-value`. However, this turns out not to be the case. In this situation `eval(program-state(abs-prog))` is not equal to a term of the form `program-state(env)`. This implies that `run(s)` is not of the form `pico-value(i)`, `pico-value(s)` or `error-value`.

That `eval(program-state(abs-prog))` is for no environment `env` equal to `program-state(env)` can be seen by looking at module `PICO-evaluator` where the function `eval` has been specified. The relevant part of the export signature of module `PICO-evaluator` is:

```
exports
    begin
       sorts        PROGRAM-STATE
       functions
            program-state  : PICO-PROGRAM    → PROGRAM-STATE
            ...
            program-state  : VALUE-ENV       → PROGRAM-STATE
            eval           : PROGRAM-STATE   → PROGRAM-STATE
            ...
    end
```

Note that this export section contains more than one function called `program-state`. Here the overloading mechanism of ASF has been used. The equation:

```
eval(program-state(abs-prog)) = program-state(env)
```

should be interpreted as meaning that evaluation of the PICO program corresponding to the abstract syntax tree `abs-prog` terminates with the value environment `env`.

Now, the equation that is meant to give the semantics of the while statement in PICO is equation 363 in module `PICO-evaluator`:

```
[363]  eval(program-state(abs-while(x, ser), env))

= if(eq(eval-exp(x,env), pico-value(0)),
   program-state(env),
```

```
eval(program-state(append-statement(ser, abs-while(x,ser)),
                   env))
```

It will be clear that this equation cannot be used to eliminate the `eval` function, when we are considering a program for which the condition in the while statement stays true for ever.

What does this mean? Because of this problem with nonterminating programs, there are unexpected elements in the sort `PICO-VALUE`. In module `PICO-values`, an element of sort `PICO-VALUE` was either an `INTEGER`, a `STRING` or an `error-value`, as is indicated by the export signature of module `PICO-values`:

```
exports
    begin
       sorts PICO-VALUE
       functions
          error-value :                          → PICO-VALUE
          pico-value  : INTEGER                  → PICO-VALUE
          pico-value  : STRING                   → PICO-VALUE
          eq          : PICO-VALUE × PICO-VALUE → BOOL
    end
```

However, in module `PICO-system` we also have elements of the form `run(non-terminating-PICO-program)` in sort `PICO-VALUE`. This shows that the import of module `PICO-values` in `PICO-system` is not persistent.

As is usual with this kind of problem, the impersistence is not restricted to sort `PICO-VALUE`. For example, the Booleans (sort `BOOL`) are also affected. Namely, suppose that the string `strange-PICO-program` is:

```
'begin
  declare
    true:    integer,
    output:  integer;
  true := 1;
  output := 0;
  while true do
    if output then output := 0 else output := 1 fi
  od
end'
```

The value of the variable `output` flips in this program infinitely often between 0 and 1, and the program does not terminate with either value. We have now two interesting new Boolean values in sort `BOOL`:

```
eq(run(strange-PICO-program),pico-value(0))
```

and:

```
eq(run(strange-PICO-program),pico-value(1))
```

The error that we studied in this section shows an interesting class of specification errors: Nonterminating calculations that introduce unexpected elements in simple datatypes. In contrast to the error from the previous section, we have here a class of errors that can be detected by verifying the persistence of the specification.

A remedy for this kind of errors in general involves a major restructuring of the specification.

1.2.3. Forgetting one case from a list of cases. The two errors that will be studied in the next two sections both come from one module. This module is in fact a family of modules, parametrised by a natural number K:

```
data module Nat(K)
begin
     sort
          N
          B
     functions
          0:              → N
          max:            → N
          succ:  N        → N
          true:           → B
          false:          → B
          eq:     N × N → B
     variables
          x, y: → N
     equations
          max                       = succ^K(0)
          succ(max)                 = max
          eq(0, 0)                  = true
          eq(0, succ(x))            = false
          eq(succ(x), 0)            = false
          eq(succ(x), succ(y))      = eq(x, y)
end Nat(K)
```

In the original version of this module (the one that was published in [Bergstra, 1988]) the two lines that are slanted:

```
eq: N × N → B
```

and:

```
eq(0, 0) = true
```

were missing. In a revised edition ([Bergstra, 1989]), the second line was added, but the first line was still missing. The omission of the declaration is characteristic of the fact that specifications in these experimental formalisms are often not mechanically checked, because of a lack of appropriate programs. The omission of the equation is the topic of the current section.

Suppose that we want to specify equality on the natural numbers `N` inductively, but forget to specify one of the cases in the induction: in this specification the case for `eq(0,0)`. There are then three Booleans in the sort `B`, the elements `true`, `false` and `eq(0,0)`. This is clearly not what was intended by the author of this specification. Strange enough, there is no impersistence, because there is only one module. However, we can change the example slightly by splitting it in two modules, in which case there will be an impersistence.

Therefore, consider the following (invalid) Perspect specification, that gives the pure form of the 'missing `eq(0,0)`-case problem':

```
external Bool
 sort B
 function true, false: B
internal Bool
 [empty]

external Nat
 sort N
 function 0, succ(N): N
 import Bool
 function eq(N,N): B
internal Nat
 variable x, y: N
 equation
  eq(succ(N),0), eq(0,succ(N)): false
  eq(succ(x),succ(y)): eq(x,y)
```

In this specification there is an impersistence because there are three Booleans in module `Nat` (namely `true`, `false`, and `eq(0,0)`), while there were only two in module `Bool`. This impersistence is shown when we ask the opinion of the Perspect checker. When confronted with this specification it says:

```
##  Module Nat is not strongly persistent.
#   There are new normal forms of sort B of the form
#     eq(0,0)
#   added in this module (junk). [5.4]
```

And the Perspect checker is right.

(The reference '`[5.4]`' in this error message refers us to section 3.1.5.4 of this

thesis, which says: 'For each sort in the module, it is required that the set of closed normal forms (i.e., normal forms without variables) of that sort should be equal to the set of closed normal forms of the sort with respect to the total rewriting system of the module in which the sort is declared.')

This error is also of a general nature. Because in an initial algebra specification the choice between different options is often formulated using pattern matching, which often consists of a long list of 'cases', it is easy to forget one possibility. In this situation persistence checking excels at finding the omissions.

1.2.4. Induction over finite sorts. The last real specification error that we will study is another error from the module in the previous section. It shows that it is hard to specify finite sorts with equality using initial algebra semantics. This topic is further elaborated in [Bergstra, Mauw, Wiedijk, 1989].

If we take the complete form of module `Nat`(K) in which the equation:

```
eq(0, 0) = true
```

is present, and split it in a `Bool`-part and a `Nat`-part, the Perspect checker prints in response to that specification three messages. The second of those messages is:

```
##  Module Nat(K) is not open confluent.
#     eq(succ(x),succ(max)):
#       eq(x,succ^K(0))
#       eq(x,succ^K-1(0))
#   Maybe the equations
#     eq(succ(x),succ(y)): eq(x,y)
#     succ(max): max
#   should be more specific. [5.3]
```

(This is, of course, not the literal output of the Perspect checker. We have abbreviated terms like `succ(succ(succ(succ(succ(succ(succ(0))))))` to $\texttt{succ}^{\texttt{K}}\texttt{(0)}$ for clarity)

Some reflection shows that the criticism implicit in this output is appropriate. The message for example tells us that:

$$\texttt{eq(x,succ}^{\texttt{K}}\texttt{(0))} = \texttt{eq(succ(x),succ(max))} = \texttt{eq(x,succ}^{\texttt{K-1}}\texttt{(0))}$$

which indeed follows from the specification. Now, take `x` equal to:

$$\texttt{succ}^{\texttt{K-1}}\texttt{(0)}$$

The left hand side can then be evaluated to `false`, while the right hand side gives `true`, so we have shown that the equation:

```
false = true
```

follows from the specification. In the form of the specification in which the Booleans have been introduced in a separate module, there is again an impersistence.

This specification error shows that it is hard to specify a finite sort by means of initial algebra semantics, while also having the advantages of unary induction using the `succ` function. It becomes even harder to specify a finite sort with a successor-like function when one tries to find a specification that is executable as a term rewriting system (this is necessary if one wants to express it in valid Perspect). The specification error that was studied in this section is not so much an error from some general class of specification errors, as well as an inherent weakness of initial algebra specifications.

The examples in the preceding sections show that a good heuristic for finding specification errors is to try to prove the persistence of the specification. A persistence check could have pointed out three out of the four realistic specification errors that were studied here. This observation clearly motivates the study of persistence in the remaining part of this thesis.

1.3. Background and related work

We will now briefly indicate the relation of our research with that of others. There exists a vast literature about algebraic specifications, which we have not the ability nor the desire to summarize here. An overview with an annotated bibliography can be found in [Bidoit, Kreowski, Lescanne, Orejas, Sannella, 1991].

First of all, one may ask on which results from the literature our own work is built. In fact, all we use is the theory of initial algebra semantics (as described in [Ehrich, Mahr, 1985]), the theory of term rewriting systems (as described in [Dershowitz, Jouannaud, 1991] and [Klop, 1991]) and the theory of path orderings ([Huet, Oppen, 1980]). This theory is well known and may be considered 'background knowledge' for people working in the field of algebraic specifications. The notion of *persistence* is also part of this 'background knowledge'. It is treated extensively in classics like [Goguen, Thatcher, Wagner, 1976], [Ehrich, Mahr, 1985], [Padawitz, 1983] and [Broy, Wirsing, 1982].

Still, even though the part of the theory that we use is well known, in order to make the presentation of our Perspect language self-contained we give an (almost) complete account of it in chapter 2. For definitions and proofs that we copied from the literature, we give a reference to the original source. For the rest of this theory, the originators can be found in [Dershowitz, Jouannaud, 1991] and [Klop, 1991].

Note that our terminology slightly differs from that found in the literature. For instance, what we call a term algebra (in section 2.1.2.5) is only superficially related to the notion of canonical term algebra (see [Ehrich, Mahr, 1985]). Similarly, what we call weak persistence (in section 2.5.1.2) is called strong persistency in [Ehrich, Mahr, 1985]. This discrepancy in terminology was another reason to make our presentation self-contained.

Second, one may ask in what way our research is related to the work done on algebraic specifications by the Programming Research Group at the University of Amsterdam and by the Department of Software Technology at the Centrum voor Wiskunde

en Informatica, also in Amsterdam. The answer is that this work formed the motivation for our research, but is only remotely related to it. The work on Perspect has been motivated by earlier work on ASF as reported in [Bergstra, Heering, Klint, 1989]. The experience that we got from working with the ASF system motivated our desire for more reliable specification techniques by enforcing persistence. We tried to give the definition of persistence within the framework of module algebra as introduced in [Bergstra, Heering, Klint, 1989], but this did not lead to something satisfying, and we left the attempts out of this thesis.

Finally, we should give a picture of related research that is concerned with the topics that we treat here: checking the persistence of a modular specification and compiling specifications to programs in traditional programming languages. Checking persistence is closely related to checking sufficient completeness which was introduced in [Guttag, Horning, 1978] and developed in [Huet, Hullot, 1980], [Bidoit, 1981], [Dershowitz, 1982], [Padawitz, 1983], [Thiel, 1984], [Rémy, Uhrig, 1988].

In the ASF project two researchers, P.R.H. Hendriks and H.R. Walters, looked at the problem of how to make algebraic specifications executable. Their work can be found in [Hendriks, 1991] and [Walters, 1991]. In our chapter 4 we address the same problem. While our scheme for translating term rewriting systems to Prolog was motivated by an early overview of similar schemes in [Bouma, Walters, 1987], it seems that the schemes presented in chapter 4 are on a different track compared to the schemes presented in the Ph.D. theses of Hendriks and Walters. While we are mainly concerned with conceptual simplicity and clarity, they are far more preoccupied with creating a usable and efficient system.

Various other approaches to the compilation of algebraic specifications to programs have been developed. As target languages appear LISP [Kaplan, 1987], Pascal [Geser, Hussmann, Mueck, 1987], Prolog [van Emden, Yukawa, 1986], [Ganzinger, Schäfers, 1990] and abstract machine code [Klaeren, Indermark, 1989], [Wolz, Boehm, 1989]. Of course the compilation schemes presented in these references are closely related to ours; we try to stress here orthogonality for our Prolog scheme and modularity for our Modula scheme.

Chapter 2

Theory

There is no difference between theory and practice in theory,
but there is a great deal of difference between theory and practice in practice.

In this chapter we will:
- Define a decidable subclass of the class of persistent specifications
- Prove that this subclass is not *too* small
- Prove the undecidability of various alternatives to this subclass
- Give a number of algorithms that together form a decision procedure for the question whether a specification is in this subclass

Most of the theory in this chapter is well known (see for instance [Dershowitz, Jouannaud, 1991] or [Klop, 1991]). Possibly *new* is:
- The specific combination of requirements that gives the decidable subclass of persistent specifications
- The part of the decision algorithm that checks the strong persistence requirement in the definition of this subclass

In this thesis we study persistent specifications. These are modular specifications in which the 'meaning' of the objects being specified does not change when importing some module from the specification into another module. A problem with the notion of persistence is that it is not decidable. Therefore, we want to define a *decidable* subclass of the class of persistent specifications, and for practical purposes we want this subclass to be not too small.

In order to obtain such a decidable class of persistent specifications, we follow the following approach:
- We will only look at *semi-complete* unconditional term rewriting systems (i.e. term rewriting systems that are weakly terminating and closed confluent). In these specifications, the initial algebra of the specification corresponds to its term algebra, which consists of the set of closed normal forms of the specification.
- We will require the specifications to be *strongly* persistent. The notion of strong persistence is the natural equivalent for term rewriting systems to the requirement of (weak) persistence for equational specifications. It says that the set of normal forms should not change on import (while the notion of persistence says that the

initial algebra should not change).

A problem with this approach is that while it is simple, it is *too* simple: the set of specifications defined by these two requirements is still not decidable. We will have to modify this scheme by making its requirements even stronger.

If we look at the definition of semi-completeness, it consists of the requirement of weak termination in conjunction with that of closed confluence. We will strengthen these requirements to *monotone* termination with respect to a *path ordering* and to *open* confluence. This gives a combination of requirements that *is* decidable.

It is not known (to me) whether the requirement of strong persistence is decidable (although I think it is probable that it is). If we only look at term rewriting systems that are left linear, a decision procedure for strong persistence is easy. Now, if in the general case the notion of strong persistence is decidable, a decision procedure will probably be not very efficient. For this reason we will restrict ourselves in this thesis to the class of *left linear* term rewriting systems.

Put together, the set of requirements that defines the decidable class of persistent specifications that we will study here is the following. A specification should be:

> monotone terminating with respect to some given path ordering + open confluent + left linear + strongly persistent

This set of requirements consists, apart from the requirement of left linearity, of three parts. These three parts are termination, confluence and persistence and they correspond to the three main parts of this chapter, which are sections 2.2, 2.3 and 2.5.

Each of these sections has the same structure. First, some basic 'semantic' notions are given and their implications for the relation between a term algebra and the initial algebra of a term rewriting system are studied. Second, a decision procedure for one of these notions will be developed. Specifically, algorithms will be given for checking:

- whether a given term rewriting system is monotone terminating with respect to some given path ordering
- whether a given term rewriting system that is known to be strongly terminating is open confluent
- whether a modular left linear term rewriting system is strongly persistent

Third, in each of these sections the undecidability of various alternative combinations of requirements will be shown. Each of these combinations is obtained by replacing one of the requirements in the list by some weaker variant. In particular we will show the undecidability of the lists of requirements in which:

- the term rewriting system only needs to be strongly terminating instead of monotone terminating with respect to a path ordering
- the term rewriting system only needs to be closed confluent instead of open confluent
- the term rewriting system only needs to be weakly persistent instead of strongly persistent

Apart from the sections about termination, confluence and persistence, this chapter contains three other sections. The first of these sections – section 2.1 – is introductory. It introduces the relation between the term algebras and the initial algebra of a

term rewriting system. Furthermore, there are two excursions. The first of these excursions – section 2.4 – briefly recapitulates a number of reduction strategies which will be referred to in the description of compilation to Prolog in chapter 4. The second excursion – which is section 2.6 – defines the notion of a primitive recursive algebra. This notion is used to show that the class of persistent specifications defined here is not too small. In particular it will be shown that it is possible to specify all primitive recursive algebras under the restrictions that we enumerated. So, we have the inclusions (both proper):

> primitive recursive algebras $\subset$ algebras that can be specified by a specification satisfying the requirements of the decidable class of persistent specifications that we study here $\subset$ algebras that can be specified persistently

2.1. Basic notions

In this section we will:
- Fix our basic definitions and terminology
- Give the relation between the initial algebra of an equational specification and the term algebras of a term rewriting system
- Give the basis for the motivation behind the concepts of termination and confluence

An algebraic specification can basically be looked at in two ways. On the one hand it can be an *equational specification* E and on the other hand it can be a *term rewriting system* T. With each term rewriting system T an equational specification E(T) is associated, obtained by 'forgetting' the direction of the rewriting arrows in T.

The meaning of an equational specification E, its *initial algebra* I(E), is a formal, rather abstract object. The meaning of a term rewriting system T, given a *normalization function* N for T, is the *term algebra* $I_N(T)$. It consists of the normal forms of T and is far more concrete. Here, we have the notion of 'executing' the specification in order to obtain the normal form of some term.

The main object defined in this section is a homomorphism ι that maps the term algebra $I_N(T)$ of a term rewriting system to the initial algebra I(E(T)) of its associated equational specification:

$$\iota\colon I_N(T) \rightarrow I(E(T))$$

This mapping is the basis of the motivation for the notions of termination and confluence. While this section only gives the various definitions involved, in later sections we will prove that:
- 'there is at least one normalization function N' is equivalent to 'T is weakly terminating'
- 'there is at most one normalization function N' is implied by 'T is strongly closed confluent'

- 'the mapping ι is surjective' is always true
- 'the mapping ι is injective' is equivalent to 'T is strongly closed confluent'

These properties together motivate the definition of the notion of semi-completeness as the combination of weak termination and strong closed confluence.

2.1.1. Equational specifications

2.1.1.1. Definition. A *signature* Σ consists of a finite set of sort names $S(\Sigma)$, a finite set of function names $F(\Sigma)$, a function dom: $F(\Sigma) \to S(\Sigma)^*$ that gives the names of the sorts which form the domain of each function, and a function ran: $F(\Sigma) \to S(\Sigma)$ that gives the name of the range for each function.

2.1.1.2. Definition. Let Σ be a signature. An *algebra* A with signature Σ consists of, for each sort name σ in $S(\Sigma)$, a set σ_A called the *sort* with name σ in A, and for each function name f in $F(\Sigma)$, a function f_A called the *function* with name f in A. Moreover, if for a function name f we have $dom(f) = \langle\sigma_1, \sigma_2, \ldots, \sigma_n\rangle$ and $ran(f) = \sigma$, then the function f_A should have domain and range as given by f_A: $(\sigma_1)_A \times (\sigma_2)_A \times \ldots \times (\sigma_n)_A \to \sigma_A$.

2.1.1.3. Definition. Let Σ be a signature. A set of *variables* V for the signature Σ consists of a disjoint union $\cup_{\sigma \in S(\Sigma)} V_\sigma$, in which each V_σ is a countably infinite set of variables of type σ.

Given a set of variables, we define the set of *open terms* or just *terms* $T(\Sigma, V)$ and the typing function τ: $T(\Sigma, V) \to S(\Sigma)$ simultaneously. This set $T(\Sigma, V)$ is the minimal set such that:

(i) if v is a variable of type σ, then v is an open term in $T(\Sigma, V)$ and has type $\tau(v) = \sigma$.

(ii) if f is a function symbol, and $t_1, t_2, \ldots, t_n$ are open terms from $T(\Sigma, V)$ of the appropriate types (i.e., each t_i has type σ_i and $dom(f) = \langle\sigma_1, \sigma_2, \ldots, \sigma_n\rangle$), then $t \equiv f(t_1, t_2, \ldots, t_n)$ is also an open term in $T(\Sigma, V)$ and it has type $\tau(t) = ran(f)$. The terms t_i are called the *arguments* of t

An *equation* $t = t'$ is a pair of open terms t and t' that have the same type.

An *equational specification* is a set of equations.

2.1.1.4. Definition. The *subterm* relation between terms is the reflexive and transitive closure of the argument relation. A variable *occurs in* a term if it is a subterm of that term. The *number of occurrences* of a variable in a term is defined recursively:

(i) if w is a variable, the number of occurrences of v in w is one if v is equal to w and zero if v is not equal to w.

(ii) if $t = f(t_1, t_2, \ldots, t_n)$, then the number of occurrences of v in t is the sum of the number of occurrences of v in the arguments t_i.

It will be clear that a variable occurs in a term if and only if its number of occurrences in that term is strictly positive.

Now let Σ be a signature and let V be a set of variables for it. A *substitution* for the

variables in V is a function σ: V $\rightarrow$ T(Σ, V) such that the term being substituted for a variable has the appropriate type, i.e., $\tau(\sigma(v)) = \tau(v)$ for all variables v. The operation of *applying* a substitution σ to a term t will be written using square brackets as σ[t]. It is defined recursively:

(i) if v is a variable, then $\sigma[v] \equiv \sigma(v)$.

(ii) if $t = f(t_1, t_2, \ldots, t_n)$, then $\sigma[t] \equiv f(\sigma[t_1], \sigma[t_2], \ldots, \sigma[t_n])$.

A *context* s[♦] is a term s together with a variable ♦ that occurs exactly once in s. The result s[t] of placing a term t in the context s[♦] is defined as σ[s] where the substitution σ is defined by $\sigma(♦) \equiv t$ and $\sigma(v) \equiv v$ for $v \neq ♦$.

Now let be given an equational specification E. The relation of *one-step provable equality* in E exists between all pairs s[σ[t]] and s[σ[t′]] for which $t = t'$ is an equation in E, s[♦] an arbitrary context and σ an arbitrary substitution.

The relation of *provable equality* in E is the reflexive, symmetric and transitive closure of the relation of one-step provable equality, i.e., it is the minimal equivalence relation that contains the relation of one-step provable equality.

2.1.1.5. Definition. A term is called *closed* when no variables occur in it. The set of closed terms is written as $T_c(\Sigma)$. This notation reflects that the set of closed terms does not depend on the set of variables V.

Let be given an equational specification E. In order to define the *initial algebra* I(E) of this specification, we will have to define sorts $\sigma_{I(E)}$ and functions $f_{I(E)}$. Let σ be a sort name, then the set $\sigma_{I(E)}$ consists of the equivalence classes of the set of closed terms of type σ under the equivalence relation of provable equality with respect to E. If f is a function name, then the function $f_{I(E)}$ is defined by:

$$f_{I(E)}([t_1], [t_2], \ldots, [t_n]) \equiv [f(t_1, t_2, \ldots, t_n)]$$

In this definition [t] is the equivalence class of the closed term t under provable equality with respect to E. It is straightforward to verify that this definition does not depend on the representations $t_1, t_2, \ldots, t_n$ that are used.

2.1.2. Term rewriting systems

2.1.2.1. Definition. A *rewriting rule* $t \rightarrow t'$ is a pair of open terms t and t′ of the same type, such that all variables that occur in t′ also occur in t.

A *term rewriting system* T is a set of rewriting rules.

2.1.2.2. Definition. Let T be a term rewriting system. We obtain the *equational specification* E(T) *that is associated with* T by replacing each rewriting rule $t \rightarrow t'$ in T by the equation $t = t'$.

2.1.2.3. Definition. Let be given a term rewriting system T. The relation of *one-step reduction* in T exists between all pairs s[σ[t]] and s[σ[t′]] for which $t \rightarrow t'$ is a rewriting

rule in T, $s[\blacklozenge]$ an arbitrary context and σ an arbitrary substitution.

The relation of *reduction* in T is the reflexive and transitive closure of the relation of one-step reduction in T.

A term n is called a T-normal form, or just a *normal form*, if there does not exist a term n' such that n has a one-step reduction to n'.

2.1.2.4. Definition. Let T be a term rewriting system. A function $N: T_c(\Sigma) \rightarrow T_c(\Sigma)$ is called a *normalization function* for T if for all closed terms t the closed term N(t) is a normal form and t reduces to N(t) in T.

2.1.2.5. Definition. Let be given a term rewriting system T and some normalization function N for T. We define the *term algebra* $I_N(T)$ associated with N as having sorts $\sigma_{I_N(T)}$ that consist of all closed normal forms of type σ, and functions $f_{I_N(T)}$ that are defined by

$$f_{I_N(T)}(n_1, n_2, \ldots, n_n) \equiv N(f(n_1, n_2, \ldots, n_n))$$

2.1.2.6. Definition. Let A and B be algebras over the same signature. A *homomorphism* $h: A \rightarrow B$ consists of a collection that for each sort σ contains a function $h_\sigma: \sigma_A \rightarrow \sigma_B$ such that:

$$h_\sigma(f_A(x_1, x_2, \ldots, x_n)) = f_B(h_{\sigma_1}(x_1), h_{\sigma_2}(x_2), \ldots, h_{\sigma_n}(x_n))$$

where $x_1 \in (\sigma_1)_A$, $x_2 \in (\sigma_2)_A$, …, $x_n \in (\sigma_n)_A$, $dom(f) = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ and $ran(f) = \sigma$.

A homomorphism is called an *isomorphism* if all functions h_σ are bijective.

Let T be a term rewriting system, and N a normalization function for T. The homomorphism $\iota: I_N(T) \rightarrow I(E(T))$ is defined by:

$$\iota_\sigma(n) \equiv [n]$$

for all closed normal forms n of type σ. It is straightforward to verify that this definition in fact defines a homomorphism. In proposition 2.3.1.6 we will show under what circumstances ι is an isomorphism.

2.2. Termination

In this section we will:

- Show how the notion of termination affects the relation between a term algebra of a term rewriting system and the initial algebra of its associated equational specification
- Give definitions and properties of various path orderings: the recursive path ordering, the lexicographic path ordering and the path ordering with argument status
- Give an algorithm for verifying whether two terms are related by such a path ordering

- Show the undecidability of a number of notions of termination, even in the presence of some heavy restrictions

This section defines a number of variations on the theme of termination. There are two, orthogonal, dimensions to this notion. On the one hand there is a dichotomy between weak and strong termination. On the other hand there is a dichotomy between open and closed termination.

A more constructive notion of termination is that of *monotone* termination with respect to a path ordering. This notion has various variants corresponding to various types of path ordering. Here we will define three types of path ordering: the recursive path ordering, the lexicographic path ordering and the path ordering with argument status.

We will start the section by showing, as promised in section 2.1, that a term rewriting system has a normalization function if and only if it is weakly closed terminating. This proposition is trivial, but essential because it shows the desirability of the notion of termination.

Then, we will move on to the study of path orderings. A path ordering is not some fixed ordering, but is parametrized by an ordering on the set of function symbols. So, a path ordering associates an 'ordering' (actually a partial preordering) on the set of terms over some signature with an 'ordering' on the function symbols from that signature. The definition of a path ordering is chosen such that it can be used to verify the termination of a term rewriting system.

Now, how is a path ordering defined? Even the definition of the best known of path orderings – the *recursive path ordering* – is not very easy due to some complicating factors. These are:

- The ordering on the function symbols which forms the input of the path ordering construction is a *partial preordering*. This means that two function symbols may be unrelated ('partial') or may be equivalent but not equal ('preordering').
- The terms that are being compared by the path ordering are *open*: they may contain variables. While this is essential for the application of the path ordering, it distorts the elegant symmetry of its definition.
- When defining a path ordering, in fact *two* orderings are being defined simultaneously: an ordering on terms, and an ordering on tuples. The ordering on tuples is related to the one on terms by being the *multiset ordering* corresponding to the ordering on the terms. This multiset ordering causes the path ordering to be a preordering – even when the ordering on function symbols it is derived from is not. For instance the terms f(a, b) and f(b, a) are equivalent under the recursive path ordering (because {a, b} and {b, a} are the same multiset) though they are not equal.

So, even when we remove two of these complicating factors by postulating that we will start with a total ordering on the function symbols and that we will only compare closed terms, we will first have to clarify the notion of a multiset ordering. Now, in the case that the ordering on the terms is *total*, the multiset ordering on the tuples – seen as multisets of terms – is relatively simple. One just compares the maximum of both tuples. If they are different, the tuple with the largest maximum will be largest in the multiset ordering. If the maxima are equal, one removes this common maximum *once*

on both sides of the comparison and tries again.

Given that the ordering on function symbols is total, that the terms being compared do not contain variables and that the multiset ordering on tuples is understood, the definition of the recursive path ordering becomes simple. Suppose that there are given two terms $s \equiv f(s_1, s_2, \ldots, s_m)$ and $t \equiv g(t_1, t_2, \ldots, t_n)$ and that we want to show that $s < t$ with respect to the recursive path ordering (note that either m or n may be zero, in which case s or t will be a constant).

There are three cases:

- $f < g$: in this case $s < t$ will hold if and only if for *all* i in {1…m} it is true that $s_i < t$
- $f > g$: in this case $s < t$ will hold if and only if for *some* j in {1…n} it is true that $s \leq t_j$
- $f = g$: in this case $s < t$ will hold if and only if $\{s_1, s_2, \ldots, s_m\} < \{t_1, t_2, \ldots, t_n\}$ with respect to the multiset ordering

That is all: a nice symmetrical definition by cases. Actually, we have not defined the relation '≤' which is used in the second clause; it may be replaced here by 'not >'.

Note that the definition of the path ordering between two terms is recursive with respect to the way smaller terms compare: in each of the cases in the definition, on at least one side of the comparison one will select one of the arguments.

Now, apart from extending this definition to the case of a partial preordering on the function symbols and to that of the comparison of open terms, one can change the role of the multiset ordering in this definition. First of all, one can replace it by the lexicographic ordering between the argument tuples. This leads to the *lexicographic path ordering*. More general, one can make the choice between multiset and lexicographic ordering dependent on the function symbol. Still even more general, one can also allow a mixed ordering: some of the argument places may be compared by the multiset ordering and some others lexicographically. This last generalization gives the *path ordering with argument status*.

All these path orderings satisfy a number of properties that make them useful for proving the termination of a term rewriting system. These properties are:

- the path ordering is stable under substitution
- the path ordering on two terms is not disturbed by placing these terms in some common context
- the path ordering considers subterms of a term to be smaller than that term itself

An ordering that satisfies the last two properties is called a *simplification ordering*. Such an ordering always has the property that it is well founded: there is no infinitely decreasing sequence of terms. This surprising fact is called *Kruskal's theorem*.

The relation between path orderings and term rewriting systems is simple. A term rewriting system is called *monotone terminating with respect to the path ordering* when all its rules are decreasing in the path ordering. In that case Kruskal's theorem implies that the term rewriting system is indeed (strongly) terminating.

As we remarked earlier, the definition of a path ordering is recursive in terms of itself evaluated on smaller terms. This means that following the definition gives one an *algorithm* for evaluating the path ordering. This implies that it is decidable whether a term rewriting system is monotone terminating with respect to a given path ordering, as we promised.

So, the notion of monotone termination with respect to a path ordering is decid-

able. However, the notions of strong/weak open/closed termination are all undecidable, even when the following requirements are satisfied by the specification:

open confluent + left linear + strongly persistent

This will be shown by constructing, given some arbitrary recursive function F, a term rewriting system that satisfies these requirements. This term rewriting system is constructed in such a way that it is (strongly/weakly open/closed) terminating if and only if F is a total function (i.e., terminating for all natural numbers). Because this last property is undecidable, the property of being (strongly/weakly open/closed) terminating is also undecidable.

2.2.1. Weak and strong termination

2.2.1.1. Definition. A sequence of terms is called a *reduction sequence* when each element in the sequence is a one-step reduct of the previous one.

A term rewriting system is called *strongly open terminating* when for all open terms t there are no infinite reduction sequences starting with t. It is called *weakly open terminating* when for all open terms t there exist a reduction sequence starting with t that cannot be extended to an infinite reduction sequence.

If one restricts oneself in the previous paragraph to the set of closed terms instead of that of all open terms, one gets the definition of strong and weak *closed* termination.

In this definition 'strong' and 'closed' are the default, i.e., when we call a term rewriting system terminating, we mean that it is strongly closed terminating.

2.2.1.2. Proposition. *A term rewriting system* T *has a normalization function if and only if it is weakly closed terminating.*

Proof. '⇒': Let N be a normalization function. For closed terms t the image N(t) is a normal form of t. The reduction of t to N(t) gives us a reduction sequence that is not infinitely extensible (in fact, it is not extensible at all). This implies that T is weakly closed terminating.

'⇐': Let t be a closed term. Because T is weakly closed terminating, t has a reduction sequence that is not infinitely extensible. Try to extend this sequence one term at the time. After some finite number of steps this will become impossible, and then the last element of the extension will be in normal form. So, it follows that each closed term has a normal form. Now, one can choose some function that maps each closed term to one of its normal forms, which is a normalization function.

2.2.2. Path orderings

2.2.2.1. Definition. A *partial preordering* $\leq$ is a relation that is reflexive and transitive. Alternatively, a partial preordering can be given as two relations $<$ and $\approx$ such that $<$ is a strict partial ordering – i.e., it is irreflexive and transitive – and $\approx$ is an equivalence relation – i.e., it is reflexive, symmetric and transitive.

These two representations can be related to each other by defining:

$$x < y \;\equiv\; x \leq y \wedge \neg\, y \leq x$$
$$x \approx y \;\equiv\; x \leq y \wedge y \leq x$$

in order to obtain $<$ and $\approx$ from $\leq$, and by defining:

$$x \leq y \;\equiv\; x < y \vee x \approx y$$

in order to obtain $\leq$ from $<$ and $\approx$.

It is easy to verify that these definitions are inverse to each other, and that they transform the two definitions of a partial preordering into each other.

2.2.2.2. Definition. Let be given some signature Σ with variables V, and let T be the set of all open terms $T(\Sigma, V)$. Furthermore, let be given some partial preordering $\leq_T$ on T with associated partial ordering $<_T$ and equivalence $\approx_T$.

For $\mathbf{t} \equiv \langle t_1, t_2, \ldots, t_n \rangle$ a tuple from T^*, the *characteristic function* $\chi_{\mathbf{t}}$ of $\mathbf{t}$ with respect to the partial preordering $\leq_T$ is defined on the set T by $\chi_{\mathbf{t}}(t) \equiv \#\{\, i \mid t_i \approx_T t \,\}$, i.e., the value for t is the number of elements from $\mathbf{t}$ that are equivalent to t.

Now, the *multiset ordering* $\leq_{mul}$ associated with $\leq_T$ is the partial preordering on T^* defined by:

$$\mathbf{s} \leq_{mul} \mathbf{t} \;\equiv\; \forall\, s\colon (\chi_{\mathbf{s}}(s) > \chi_{\mathbf{t}}(s) \;\Rightarrow\; \exists\, t >_T s\colon \chi_{\mathbf{s}}(t) < \chi_{\mathbf{t}}(t))$$

This definition clearly does not depend on the order of the components in the tuples that are being compared (because $\chi_{\mathbf{t}}$ does not), which motivates the name 'multiset ordering'. Note that the truth of $\mathbf{s} \leq_{mul} \mathbf{t}$ only depends on the truth-values of $s <_T t$ for s a component of $\mathbf{s}$ and t a component of $\mathbf{t}$.

The *lexicographic ordering* $\leq_{lex}$ associated with $\leq_T$ is also a partial preordering on T^*. It is defined recursively by:

$$\langle s_1, s_2, \ldots, s_m \rangle \leq_{lex} \langle t_1, t_2, \ldots, t_n \rangle \;\equiv$$
$$m = 0 \vee s_1 <_T t_1 \vee (s_1 \approx_T t_1 \wedge \langle s_2, \ldots, s_m \rangle \leq_{lex} \langle t_2, \ldots, t_n \rangle)$$

A *status marker* is an element from the two element set {*mul*, *lex*}. An *argument status* ζ for the signature Σ is a function that associates with each function symbol f from $F(\Sigma)$ a tuple of status markers, such that the tuple $\zeta(f)$ has the same number of components as the domain dom(f). Now, given an argument status ζ, if $t \equiv f(t_1, t_2, \ldots,$

t_n) is a term from T and $\mathbf{t} \equiv \langle t_1, t_2, \ldots, t_n \rangle$ is the tuple of its arguments, then we can define the tuples $\mathbf{t}_{mul}$ and $\mathbf{t}_{lex}$ as containing the components from $\mathbf{t}$ for which $\zeta(f)$ is respectively *mul* and *lex*.

So, let be given an argument status ζ. The *ordering with argument status* $\leq_\zeta$ that is associated with $\leq_T$ and ζ is a partial preordering on T and is defined by:

$$s \leq_\zeta t \equiv \mathbf{s}_{lex} <_{lex} \mathbf{t}_{lex} \vee (\mathbf{s}_{lex} \approx_{lex} \mathbf{t}_{lex} \wedge \mathbf{s}_{mul} \leq_{mul} \mathbf{t}_{mul})$$

2.2.2.3. Definition [Dershowitz, 1982]**.** We will now define an ordering that is parametrized by an argument status. It has two special cases: if we take the argument status everywhere *mul*, then we obtain a partial preordering called the *recursive path ordering*; if we take the one that is everywhere *lex*, we obtain the *lexicographic path ordering*.

Let be given some partial preordering $\leq_F$ on the set of function symbols $F(\Sigma)$. Furthermore, let be given some argument status ζ. We will define the *path ordering with argument status* $\leq_{path}$ which is a partial preordering on the set of open terms and which is parametrized by $\leq_F$ and ζ. Its definition – which is highly recursive – consists of three cases, depending on whether we compare variables x and y, or non-variables $f(s_1, s_2, \ldots, s_m)$ and $g(t_1, t_2, \ldots, t_n)$.

$$\begin{aligned} s \leq_{path} t = y &\equiv s = t \\ s = x \leq_{path} t = g(t_1, t_2, \ldots, t_n) &\equiv \exists\, j\colon s \leq_{path} t_j \\ s = f(s_1, s_2, \ldots, s_m) \leq_{path} t = g(t_1, t_2, \ldots, t_n) &\equiv \\ (\forall\, i\colon s_i <_{path} t \wedge (f <_F g \vee (f \approx_F g \wedge s\ (\leq_{path})_\zeta\ t))) &\vee \exists\, j\colon s \leq_{path} t_j \end{aligned}$$

2.2.2.4. Definition. Let be given some relation $\Diamond$ on the set of open terms $T(\Sigma, V)$. This relation is called *stable under substitutions*, when for all terms t and t′ and for all substitutions σ, the relation $t \Diamond t'$ implies that the relation $\sigma[t] \Diamond \sigma[t']$ also holds.

The relation is called *stable in context*, when for all terms t and t′ and for all contexts $s[\blacklozenge]$, the relation $t \Diamond t'$ implies that the relation $s[t] \Diamond s[t']$ also holds.

2.2.2.5. Definition. A partial preordering $\leq$ on the set of open terms $T(\Sigma, V)$ is said to have the *subterm property* when for all terms t and t′, with t a proper subterm of t′ (i.e., t is a subterm of, but not equal to, t′), we have that $t < t'$.

It is called a *simplification ordering* when it is stable in context and has the subterm property. (Note that this definition says that $\leq$ should be stable in context, while we will need in the proof of proposition 2.2.2.9 that $<$ is stable in context. A path ordering is stable in both senses, as the next proposition shows.)

2.2.2.6. Proposition. *A path ordering is a partial preordering, has the subterm property and is a simplification ordering. The associated partial ordering is stable under substitutions and stable in context.*

Proof. We have to prove a number of properties, which are somewhat interdependent, so we have to be careful with the order in which they are verified. In particular,

the transitivity of $\leq_{path}$ will be proved *after* proving the subterm property.

We will start by verifying simultaneously two statements about two terms s and t, using induction on the sum of the depths of s and t. These statements are:

$s \leq_{path} t$, when s is a subterm of t
$s <_{path} t$, when s is a proper subterm of t

Note that the reflexivity of $\leq_{path}$ is a special case of the first statement, and that the second statement is just the subterm property.

Suppose that s is a subterm of t. There are three cases:

(i) The term s is equal to t, and it is a variable x. In this case, we have to show that $x \leq_{path} x$, which follows trivially from the first case in the definition of $\leq_{path}$ in 2.2.2.3.

(ii) The term s is equal to t, and it is not a variable so it has the form $f(t_1, \ldots, t_n)$. We have to show that $t \leq_{path} t$. In order to do this we need that $t_i <_{path} t$ for all i – which follows by induction – and that $t \ (\leq_{path})_\zeta \ t$. Using induction, we may assume that for each i we already know that $t_i \leq_{path} t_i$. Now (defining $\mathbf{t}$ to be $\langle t_1, \ldots, t_n \rangle$) the statements $\mathbf{t}_{mul} \ (\leq_{path})_{mul} \ \mathbf{t}_{mul}$ and $\mathbf{t}_{lex} \ (\leq_{path})_{lex} \ \mathbf{t}_{lex}$ can easily be seen to follow from the definitions of $\leq_{mul}$ and $\leq_{lex}$ in 2.2.2.2, and together they give that $t \ (\leq_{path})_\zeta \ t$.

(iii) The term s is a proper subterm of t. This means that for some j the term s is a subterm of the argument t_j of t. Then by induction we know that we have that $s \leq_{path} t_j$, which – according to the $\exists$ clause in the second and third case of definition 2.2.2.3 – gives us $s \leq_{path} t$. In order to prove $s <_{path} t$ we have to show that $t \leq_{path} s$ leads to a contradiction. So, suppose $t \leq_{path} s$, then either for some i we have $t \leq_{path} s_i$ which is impossible by induction because s_i is a subterm of t that is even deeper than s – or for all j we have $t_j <_{path} s$. But, for the index j for which s was a subterm of t_j this is also impossible.

This shows that $\leq_{path}$ is reflexive and has the subterm property.

In order to prove transitivity of $\leq_{path}$ we will need the fact that $(\leq_{path})_\zeta$ is transitive when $\leq_{path}$ is transitive on the set of arguments of the terms involved. For this, we will have to verify (switching momentarily to the notation of 2.2.2.2 in order to avoid having to write $_{path}$ all the time) that $\leq_{mul}$, $\leq_{lex}$ and $\leq_\zeta$ are transitive given that $\leq$ is a preordering on the set of terms occurring in the tuples. The case of $\leq_{mul}$ is not trivial. Suppose that $\mathbf{s} = \langle s_1, s_2, \ldots, s_m \rangle$, $\mathbf{t} = \langle t_1, t_2, \ldots, t_n \rangle$, $\mathbf{u} = \langle u_1, u_2, \ldots, u_k \rangle$, $\mathbf{s} \leq_{mul} \mathbf{t}$ and $\mathbf{t} \leq_{mul} \mathbf{u}$, then we have to prove that $\mathbf{s} \leq_{mul} \mathbf{u}$. Following the definition of $\leq_{mul}$ in 2.2.2.2, suppose we have a term s with $\chi_{\mathbf{s}}(s) > \chi_{\mathbf{u}}(s)$. Consider the set U defined by $U \equiv \{ u >_T s \mid \chi_{\mathbf{s}}(u) \neq \chi_{\mathbf{t}}(u) \vee \chi_{\mathbf{t}}(u) \neq \chi_{\mathbf{u}}(u) \}$. Because either $\chi_{\mathbf{s}}(s) > \chi_{\mathbf{t}}(s)$ or $\chi_{\mathbf{t}}(s) > \chi_{\mathbf{u}}(s)$ we know that there either is an $u >_T s$ with $\chi_{\mathbf{s}}(u) < \chi_{\mathbf{t}}(u)$ or one with $\chi_{\mathbf{t}}(u) < \chi_{\mathbf{u}}(u)$; this means that the set U is not empty. Also, because $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{u}$ all have finitely many elements, U is finite. This implies that there is some u in U such that there is no u′ in U with $u' >_T u$ – else there has to be a $<_T$ 'loop', which is impossible. Now, $\chi_{\mathbf{s}}(u) \leq \chi_{\mathbf{t}}(u)$, because if this were not the case we could use $\mathbf{s} \leq_{mul} \mathbf{t}$ to find an u′ with $u' >_T u$. Similarly, $\chi_{\mathbf{t}}(u) \leq \chi_{\mathbf{u}}(u)$. By definition of U, one of the inequalities has to be strict, so together: $\chi_{\mathbf{s}}(u) < \chi_{\mathbf{u}}(u)$. This proves $\mathbf{s} \leq_{mul} \mathbf{u}$. The transitivity of $\leq_{lex}$ and $\leq_\zeta$ are easy.

Now (switching back to the notation of 2.2.2.3), we will prove transitivity of $\leq_{path}$. We will prove this using induction on the total depth of s, t and u. So, suppose that we have s, t and u and we know that $s \leq_{path} t$ and $t \leq_{path} u$. There are four cases:

(i) If u is a variable, it is easy to see that s, t and u must all be equal.

(ii) If u is not a variable then u has to have the form $h(u_1, u_2, \ldots, u_p)$. If for some argument u_k of u we have that $t \leq_{path} u_k$, induction gives $s \leq_{path} u_k$, which shows that $s \leq_{path} u$.

(iii) If this is not the case, we know that t is also not a variable, so has the form $g(t_1, t_2, \ldots, t_n)$, and we have that $t_j <_{path} u$ for all j. Now, if there is some argument t_j of t such that $s \leq_{path} t_j$, induction gives $s \leq_{path} u$.

(iv) The case that is left is the one in which for all i we have $s_i <_{path} t$, for all j we have $t_j <_{path} u$ and furthermore we know that $f <_F g \vee (f \approx_F g \wedge s \, (\leq_{path})\zeta \, t)$ and $g <_F h \vee (g \approx_F h \wedge t \, (\leq_{path})\zeta \, u)$. By induction $s_i <_{path} t$ implies $s_i <_{path} u$ for all i, so in order to prove that $s \leq_{path} u$, all we have to show is that $f <_F h \vee (f \approx_F h \wedge s \, (\leq_{path})\zeta \, u)$. Now, if either $f <_F g$ or $g <_F h$ holds we know that $f <_F h$. If not, we may conclude that $f \approx_F h$ and application of the result from the previous paragraph then gives that $s \, (\leq_{path})\zeta \, u$.

This shows that $\leq_{path}$ is a preordering. We still have to show that $<_{path}$ is stable under substitutions and that both $\approx_{path}$ and $<_{path}$ are stable in context.

First, however, we will show the following characterization of $\approx_{path}$:

$$\begin{aligned} s \approx_{path} t = y &\equiv s = t \\ s = x \approx_{path} t = g(t_1, t_2, \ldots, t_n) &\equiv \textit{never} \\ s = f(s_1, s_2, \ldots, s_m) \approx_{path} t = g(t_1, t_2, \ldots, t_n) &\equiv f \approx_F g \wedge s \, (\approx_{path})\zeta \, t \end{aligned}$$

(The relation $(\approx_{path})\zeta$ that occurs in the last line can either be defined as the equivalence relation associated with the preordering $(\leq_{path})\zeta$, or as the ordering with argument status associated with the preordering $\approx_{path}$. Both definitions give the same relation. One can verify that $\mathbf{s} \, (\approx_{path})\zeta \, \mathbf{t}$ holds if and only if the components of **s** and **t** are $\approx_{path}$-equivalent up to some permutation.)

Only the last case is not trivial. From right to left it follows from definition 2.2.2.3 and the observation that from $s \, (\approx_{path})\zeta \, t$ it follows that for each argument s_i of s there exists some argument t_j of t such that $s_i \approx_{path} t_j$. In order to prove the last case from left to right, suppose we know that $s \approx_{path} t$. Then, we cannot have any argument t_j of t for which $s \leq_{path} t_j$, because this implies that $s \leq_{path} t_j <_{path} t$, which gives a contradiction. This implies that $f <_F g \vee (f \approx_F g \wedge s \, (\leq_{path})\zeta \, t)$ and vice versa also $g <_F f \vee (g \approx_F f \wedge t \, (\leq_{path})\zeta \, s)$. Because $f <_F g$ is not compatible with $g <_F f \vee g \approx_F f$ we find that we are in the case that $f \approx_F g \wedge s \, (\leq_{path})\zeta \, t$. Together with the converse this gives the required result.

From the characterization of $\approx_{path}$ it becomes straightforward to verify the following characterization of $<_{path}$:

$$
\begin{aligned}
s <_{path} t = y &\equiv \textit{never}\\
s = x <_{path} t = g(t_1, t_2, \ldots, t_n) &\equiv \exists\ j\colon s \leq_{path} t_j\\
s = f(s_1, s_2, \ldots, s_m) <_{path} t = g(t_1, t_2, \ldots, t_n) &\equiv\\
(\forall\ i\colon s_i <_{path} t \wedge (f <_F g \vee (f \approx_F g \wedge s\ (<_{path})_\zeta\ t))) &\vee \exists\ j\colon s \leq_{path} t_j
\end{aligned}
$$

(The relation $(<_{path})_\zeta$ that occurs in the last line is defined to be the partial ordering that is associated with the preordering $(\leq_{path})_\zeta$.)

Stability of the relation $<_{path}$ under substitutions can now be proved by applying the characterization that was just given. Let $s <_{path} t$, and suppose we have a recursive verification of this fact following the characterization that was just given. If we replace all occurrences of some variable x in both s and t by some other term, it is easy to see that this verification of $s <_{path} t$ remains valid.

We will finally indicate how to show stability in context of $\approx_{path}$ and $<_{path}$.

Stability of $\approx_{path}$ in context. Let **t** be some tuple, and **t′** a tuple obtained by replacing one of its components by an $\approx_{path}$-equivalent one. Then $\mathbf{t}\ (\approx_{path})_{mul}\ \mathbf{t'}$, $\mathbf{t}\ (\approx_{path})_{lex}\ \mathbf{t'}$ and $f(\mathbf{t})\ (\approx_{path})_\zeta\ f(\mathbf{t'})$, because all definitions in 2.2.2.2 only refer to the partial preordering and not to real equality. From this, we can derive that when we replace one argument of a term by an $\approx_{path}$-equivalent one, we get an $\approx_{path}$-equivalent term. From this observation the stability in context of $\approx_{path}$ follows with induction.

Stability of $<_{path}$ in context. Again, first show what happens when we replace some component in a tuple by some term that is larger with respect to $<_{path}$, and then apply this in the definition of the path ordering.

2.2.2.7. Definition. A partial ordering < is called *well founded* if there does not exist an infinite sequence that is decreasing in it.

2.2.2.8. Kruskal's Theorem [Kruskal, 1960], [Dershowitz, 1982]**.** *The partial ordering associated with a simplification ordering is well founded on the set of closed terms.*

Proof. We are going to show a property of a simplification ordering that is stronger: if $t_0, t_1, t_2, \ldots$ is an infinite sequence of closed terms, then there are always indices $i < j$ such that $t_i \leq t_j$.

An infinite sequence for which this property is not true we call a *counter example sequence*. We are going to show that such a sequence does not exist. Suppose, towards a contradiction, that such a counter example sequence does exist. Then we can choose a 'minimal' counter example sequence $t_0, t_1, t_2, \ldots$ in the following way. Take for t_0 some minimal term (i.e., of minimal term depth) that starts a counter example sequence. Now, suppose we have already chosen $t_0, t_1, \ldots, t_{i-1}$. Then choose for t_i a minimal term such that $t_0, t_1, \ldots, t_i$ still starts a counter example sequence. It will be clear that the sequence that one gets in this way is itself a counter example sequence.

It has the property that if we take some subsequence, and in this subsequence replace each element by one of its arguments, then we can prove that we get a sequence – say $s_0, s_1, s_2, \ldots$ – that has a subsequence that is increasing in ≤. We will

show this in two steps. First, we prove that there are $i < j$ such that $s_i \leq s_j$. To see this, suppose that s_0 was the argument of t_k. Now, consider the sequence $t_0, t_1, \ldots, t_{k-1}, s_0, s_1, s_2, \ldots$, then it cannot be a counter example sequence, because s_0 is smaller than t_k. Therefore, either for $i < j$ there are $t_i \leq t_j$ which is impossible; or for some $i < k$ and for some j we have $t_i \leq s_j$, but this is also impossible because for some $k' \geq k$ we have that s_j was the argument of $t_{k'}$ and because $\leq$ is a simplification ordering, by the subterm property, we find that $s_j \leq t_{k'}$, which means that $t_i \leq t_{k'}$ which is also impossible; or, for some $i < j$ we have $s_i \leq s_j$, which is what we wanted to prove. For the second step, we prove that there are only finitely many i such that there is no j, with $i < j$ and $s_i \leq s_j$ (i.e., for which s_i has no 'successor'). For, suppose that there are infinitely many. The subsequence of the sequence $s_0, s_1, s_2, \ldots$ given by these i itself also consists of arguments of a subsequence of $t_0, t_1, \ldots$, and this gives a contradiction with the previous paragraph.

We now, return to the 'minimal counter example sequence' $t_0, t_1, t_2, \ldots$. Because there are only finitely many function symbols, there has to be a function symbol f that occurs infinitely often as outermost function symbol in this sequence. Select the terms that have outermost function symbol f as a subsequence. We have shown that this sequence in its turn has a subsequence for which the first argument is increasing in $\leq$. Repeating this sub-selection for the other argument positions of f gives a subsequence that is increasing in all its argument positions. But $\leq$ is a simplification ordering and therefore stable in context. This proves that this last subsequence is itself increasing in $\leq$, which contradicts the choice of $t_0, t_1, t_2, \ldots$ as a counter example sequence.

2.2.2.9. Proposition. *If all rules of a term rewriting system are decreasing with respect to the partial ordering associated with a path ordering, the term rewriting system is strongly open terminating.*

Proof. Suppose that the term rewriting system is not strongly open terminating, so there exists an infinite reduction sequence. Extend (if necessary) the signature with sufficiently many constants such that for each sort a closed term exists: this will give a term rewriting system that is still decreasing with respect to the path ordering (regardless the way one orders the extra constants) and that is still not strongly open terminating by virtue of the same sequence. For each sort in the signature choose some closed term. Replace each variable in the reduction sequence by the closed term of the appropriate term: this gives an infinite reduction sequence of closed terms.

Because the rules of the term rewriting system are strictly decreasing with respect to the path ordering, and because the path ordering is stable under substitutions and in context, the reduction sequence of closed terms is also strictly decreasing. This gives a contradiction with Kruskal's theorem.

2.2.2.10. Definition. A term rewriting system is called *monotone terminating*, when there exists some path ordering with argument status (given by some partial preordering $\leq_F$ on the set of function symbols and with some argument status ζ) such that all rules of the term rewriting system are monotone decreasing in it.

2.2.3. Decidability

2.2.3.1. Algorithm. *A procedure (for a given path ordering) for deciding whether a term rewriting system is monotone terminating.*

For all rules $t \to t'$ in the term rewriting system check whether $t >_{path} t'$ by recursively evaluating the various definitions given in 2.2.2.2 and 2.2.2.3

2.2.3.2. Proposition. *It is not decidable whether a term rewriting system is strongly/weakly open/closed terminating, even when it is known that it is open confluent and left linear.*

Proof (see also [Klop, 1991])**.** Let F be some recursive function. We will construct an open confluent and left linear term rewriting system that is strongly open terminating when F is a total function and that is not even weakly closed terminating when F is not a total function.

F is a recursive function, so there exist primitive recursive functions G and G′ such that $F(x) \cong G(\mu y \cdot G'(x, y) = 0)$. Because G and G′ are primitive recursive, there exists a sequence of primitive functions $G_0, G_1, G_2, \ldots, G_n$ such that $G_{n-1} = G'$, $G_n = G$ and each G_i is zero, the successor, a projection function or is obtained from previous elements from the list by composition or primitive recursion. This list of G_i functions can easily be given by a term rewriting system in which the way each G_i is defined in terms of earlier G_i's is modelled in a straightforward way by means of rewriting rules. For instance if G_i is defined with primitive recursion from $G_{i'}$ and $G_{i''}$ then we add the following two 'defining' rules:

$$G_i(0, y_1, y_2, \ldots, y_n) \to G_{i'}(y_1, y_2, \ldots, y_n)$$
$$G_i(S(x), y_1, y_2, \ldots, y_n) \to G_{i''}(x, G_i(x, y_1, y_2, \ldots, y_n), y_1, y_2, \ldots, y_n)$$

This term rewriting system will be left linear and strongly open confluent (because there will be no non-trivial overlaps).

Now, add the following rules to this 'primitive recursive' term rewriting system:

$$F(x) \to F'(x, 0, 0)$$
$$F'(x, y_1, y_2) \to F''(x, y_1, y_2, G'(x, y_1))$$
$$F''(x, y_1, y_2, 0) \to G(y_1)$$
$$F''(x, y_1, S(y_2), S(z)) \to F'(x, S(y_1), y_2)$$
$$F''(x, y_1, 0, S(z)) \to F'(x, 0, S(y_1))$$

Clearly, everything stays left linear and non-overlapping.

Now, suppose that F is a total function. Then, every term can be reduced inside out, and such a computation will always terminate. The only terms for which this is not trivial have outermost functions F′ and F″. Because F is total, for each x there exists some y such that G′(x, y) is equal to zero. Now, while rewriting F′ and F″ a 'diagonal'

pattern is followed by the pair of arguments y_1 and y_2. So, after some finite time y_1 will become again the y for which the term $G'(x, y)$ reduces to 0, and the rewriting will terminate.

Now, let be given some closed term t. By the same 'diagonal pattern' argument, we find that we can adorn each function symbol F, F′ and F″ that occurs in t with an index that 'counts' the number of steps it takes this function symbol to 'reach' the rule $F''(x, y_1, y_2, 0) \rightarrow G(y_1)$. This gives us a new term t′. Now, each reduction sequence of t according to the original term rewriting system corresponds to a reduction sequence of t′ according to the term rewriting system in which the rules for F_i, F'_i and F''_i are:

$$F_i(x) \rightarrow F'_i(x, 0, 0)$$
$$F'_i(x, y_1, y_2) \rightarrow F''_i(x, y_1, y_2, G'(x, y_1))$$
$$F''_i(x, y_1, y_2, 0) \rightarrow G(y_1)$$
$$F''_0(x, y_1, y_2, S(z)) \rightarrow \textit{some small irrelevant term, say } 0$$
$$F''_{i+1}(x, y_1, S(y_2), S(z)) \rightarrow F'_i(x, S(y_1), y_2)$$
$$F''_{i+1}(x, y_1, 0, S(z)) \rightarrow F'_i(x, 0, S(y_1))$$

But this term rewriting system is monotone terminating according to the path ordering given by $0 < S < G_0 < G_1 < G_2 < \ldots < G_n < F''_0 < F'_0 < F''_1 < F'_1 < F''_2 < F'_2 < \ldots$ and $F'_0 < F_0$, $F'_1 < F_1$, $F'_2 < F_2$, …. This implies that the adorned term t′ has no infinite reduction sequence, and therefore that the original term t also has no infinite reduction sequence. And this means that the original term rewriting system is strongly closed terminating. To see that it is even strongly open terminating, suppose some open term has an infinite reduction. Then, if we substitute some fixed constant term (e.g. the constant 0) for all variables that occur in this reduction sequence, we obtain an infinite reduction of a closed term, which contradicts strong closed termination.

Now, if F is not a total function, by choosing for x an argument of the form $S^n(0)$ for which F diverges we find a term $F(S^n(0))$ that has no normal form, and this implies that in this case the term rewriting system is not even weakly closed terminating.

2.3. Confluence

In this section we will:

- Show how the notion of confluence affects the relation between a term algebra of a term rewriting system and the initial algebra of its associated equational specification.
- Give an algorithm for verifying whether a strongly terminating term rewriting system is *open* confluent
- Show the undecidability of *closed* confluence, even in the presence of some heavy restrictions

This section defines a number of variations on the theme of confluence. There are (again) two, orthogonal, dimensions to this notion. On the one hand there is a dichotomy between weak and strong confluence. On the other hand there is a dichotomy

between open and closed confluence.

We will start the section by showing, as promised in section 2.1, that it follows from strong closed confluence that a term rewriting system has at most one normalization function, and that strong closed confluence is equivalent to the condition that the mapping ι (defined in section 2.1 and relating a term algebra to the initial algebra) is injective. This proposition is trivial, but essential because it shows the desirability of the notion of confluence.

Now, if a term rewriting system is both weakly closed terminating and strongly closed confluent it is called *semi-complete*. Such a term rewriting system has exactly one normalization function and the mapping ι is an isomorphism, which implies that in that case the term algebra will be isomorphic to the initial algebra.

It will be clear that we should like to show a term rewriting system to be strongly confluent. This turns out not to be easy. Fortunately when a term rewriting system is strongly terminating – a property that we will want anyway – it *is* easy.

The algorithm to check whether a strongly terminating term rewriting system is strongly open confluent really only checks whether it is weakly open confluent. This is done is by looking for critical pairs: if they exist, the check has failed. These pairs are looked for by trying all overlapping pairs of redex patterns. This gives a larger term – formed by the overlap – that has two natural one step reducts. Normalizing these two reducts and comparing the normal forms tells whether the overlapping redex patterns were a critical pair. If the normal forms are equal, all is well, but if they are not equal we have found a critical pair, which implies that the check has failed.

This shows that (weak/strong) open confluence is decidable when the term rewriting system is strongly terminating. On the other hand (weak/strong) *closed* confluence is not decidable even when the following requirements are satisfied by the specification:

> monotone terminating with respect to some given path ordering + left linear + strongly persistent

To see this, we associate a term rewriting system with each pair of primitive recursive functions F and G. It is defined such that it is (weakly/strongly) closed confluent if and only if F is equal to G (i.e., their values are equal for all natural numbers). Because this last property is undecidable, the property of being (weakly/strongly) confluent is also undecidable.

2.3.1. Weak and strong confluence

2.3.1.1. Definition. A term rewriting system is called *strongly open confluent* when, if an open term t reduces both to terms t_1 and t_2 those terms have a common reduct t'. It is called *weakly open confluent* when this only needs to be true in the case that t reduces to t_1 and t_2 in one step.

If one restricts oneself in the previous paragraph to the set of closed terms instead of that of all open terms, one gets the definition of strong and weak *closed* conflu-

ence.

Just like in the definition of termination, here 'strong' and 'closed' are again the default, i.e., when we call a term rewriting system confluent, we mean that it is strongly closed confluent.

2.3.1.2. Proposition. *If a term rewriting system is strongly closed confluent it has at most one normalization function.*

Proof. Suppose this is not true, so let some strongly closed confluent term rewriting system have more than one normalization function. Then, for some closed term these strategies will give different normal forms. But these normal forms have no common reduct, and so the term rewriting system cannot be strongly closed confluent. Contradiction.

2.3.1.3. Proposition. *The homomorphism* ι *is always surjective.*

Proof. Let [t] be some element of the initial algebra I(E(T)). Then the image under the normalization function N(t) is a normal form of t, and is provably equal to t in E(T), so [t] = [N(t)]. But N(t) is a normal form, so is an element of $I_N(T)$ which is mapped to [N(t)] by ι. So, [t] is in the image of ι.

2.3.1.4. Proposition. *The homomorphism* ι *is injective if and only if the term rewriting system is strongly closed confluent.*

Proof. '$\Rightarrow$': Let a closed term t have two reducts t_1 and t_2. Now, the normal forms $N(t_1)$ and $N(t_2)$ are both provably equal to t, so $[N(t_1)] = [N(t_2)]$. In other words $\iota(N(t_1)) = \iota(N(t_2))$. But because ι is injective this means that $N(t_1) = N(t_2)$. But that means that $N(t_1) = N(t_2)$ is a common reduct of t_1 and t_2.

'$\Leftarrow$': Let n_1 and n_2 be normal forms with $\iota(n_1) = \iota(n_2)$. By definition of ι this means that $[n_1] = [n_2]$ in I(E(T)), so n_1 is provably equal to n_2. So, there exists some sequence $t_0, t_1, \ldots, t_n$ with $t_0 = n_1$, $t_n = n_2$ and each t_i provably equal in one step to t_{i-1} or vice versa. Now, strong confluence gives that each t_i has the same normal form as t_{i-1} so with induction one sees that t_0 has the same normal form as t_n. But t_0 has n_1 as normal form and t_n has n_2 as normal form, so $n_1 = n_2$.

2.3.1.5. Definition. A term rewriting system is called *semi-complete* when it is weakly closed terminating and strongly closed confluent.

2.3.1.6. Corollary. *A term rewriting system is semi-complete* $\Leftrightarrow$ *it has exactly one normalization function* $\Leftrightarrow$ *the homomorphism* ι *is an isomorphism (and therefore the term algebra is isomorphic to the initial algebra of the associated equational specification).*

Proof. First, the equivalence between semi-completeness and having exactly one normalization function. '$\Rightarrow$': This follows from 2.2.1.2 and 2.3.1.2. '$\Leftarrow$': Weak closed

termination follows from 2.2.1.2. We have to show strong closed confluence. Suppose this is false, so there is some closed term t that reduces to t_1 and t_2 without a common reduct. Then t has two different normal forms in the normal forms of t_1 and t_2. Now let N be the unique normalization function. One can create two different normalization strategies by replacing N(t) by $N(t_1)$ and by $N(t_2)$. Contradiction.

The equivalence between semi-completeness and ι being an isomorphism follows directly from 2.3.1.3 and 2.3.1.4.

2.3.1.7. Newman's lemma [Newman, 1942]**.** *If a term rewriting system is strongly open terminating and weakly open confluent it is strongly open confluent.*

Proof. We are going to show that in a strongly terminating and weakly confluent term rewriting system there are no terms having more than one different normal form. From this strong confluence directly follows.

In order to do this we show that a term with more than one normal form has at least one one-step reduct that has the same property. This leads to an infinite reduction, in contradiction with strong termination.

So, suppose that t is a term with at least two different normal forms n_1 and n_2. This means t has one-step reducts t_1 and t_2 that are the initial steps in the reductions to n_1 and n_2. Because of weak confluence, t_1 and t_2 have a common reduct and because of termination that common reduct has to have a normal form. This shows that t_1 and t_2 have a common normal form. But if t_1 or t_2 each have only one normal form it has to be n_1 respectively n_2, which are different. So either t_1 or t_2 has to have at least one other normal form. This shows that t has a one-step reduct with more than one normal form.

2.3.2. Overlapping

2.3.2.1. Definition. Two terms s and t *unify* when there are substitutions σ and τ such that $\sigma[s] = \tau[t]$. Such a pair of substitutions is called a *most general unifier* when for all substitutions σ' and τ' such that $\sigma'[s] = \tau'[t]$ there always exists some substitution υ such that $\sigma'(x) = \upsilon[\sigma(x)]$ and $\tau'(x) = \upsilon[\tau(x)]$. It can be shown that two terms that unify always have a most general unifier. (Note that the definitions of these notions are different from what is customary. For example, with these definitions terms x and F(x) unify.)

Two terms are called *overlapping* when one of the terms has a subterm which is not a variable and which unifies with the other term.

2.3.2.2. Definition. Let be given a term rewriting system T. Let further be given two rewriting rules $s \to s'$ and $t \to t'$ from T such that s and t overlap. The terms s and t are called *overlapping redex patterns*.

According to the previous definition this means that one of the terms – say s – has the form $s_1[s_2]$ for some context $s_1[\blacklozenge]$ and term s_2, with the property that s_2 unifies with the other term – t. This pair of a context $s_1[\blacklozenge]$ and a term s_2 is called the *position* of the overlap.

Let σ and τ be some most general unifier for s_2 and t, which implies that $\sigma[s_2] = \tau[t]$, then consider the term $\sigma[s] = \sigma[s_1][\sigma[s_2]] = \sigma[s_1][\tau[t]]$. Its two one-step reducts $\sigma[s']$ and $\sigma[s_1][\tau[t']]$ are called its *natural reducts*.

2.3.3. Decidability

2.3.3.1. Algorithm [Knuth, Bendix, 1970]**.** *Procedure for deciding whether a strongly open terminating term rewriting system is strongly open confluent.*

For each pair of overlapping redex patterns and for each overlap position calculate a normal form for both natural reducts and verify whether they are equal.

Proof sketch *of the correctness of the algorithm***.** Suppose that the algorithm succeeds and suppose that we have some open term t with one-step reducts t_1 and t_2. There are two possibilities. Either the redex patterns in t that lead to t_1 and t_2 overlap. Then, because the algorithm succeeded, the natural reducts of this overlap have reductions to a common normal form. By applying a substitution to these reductions, and by placing the resulting reductions in a context, we obtain reductions of t_1 and t_2 to a common reduct.

Or, the redex patterns in t do not overlap. If they are not contained in each other (i.e., they occur in different subterms of the term) it is easy to see that t_1 and t_2 have a common reduct. So, suppose that one redex pattern is contained in the other. Let t_1 be the result of reducing the 'outer' redex pattern. This might have duplicated the 'inner' redex pattern a number of times. By reducing all these copies, one arrives at the common reduct from t_1. On the other hand the instance of the 'outer' redex pattern may no longer apply in t_2, because terms that matched in t may have become different. So, first find all copies of the 'inner' redex pattern that have to be reduced to restore these matches. After that reduce the 'outer' redex pattern to get to the common reduct.

This shows that the term rewriting system is weakly open confluent. But it is given that it is strongly open terminating. Then Newman's lemma says that it is strongly open confluent.

2.3.3.2. Proposition. *It is not decidable whether a term rewriting system is strongly/weakly closed confluent, even when it is known that it is monotone terminating with respect to both a recursive and a lexicographic path ordering and is left linear.*

Proof. Let F and G be two primitive recursive functions. We are going to give a left linear term rewriting system that is monotone terminating with respect to both a recursive and a lexicographic path ordering, such that it is strongly closed confluent when F and G are the same function, but not even weakly closed confluent when they differ.

Associate two sets of function symbols F_0, F_1, …, $F_m \equiv F$ and G_0, G_1, …, $G_n \equiv G$ with F and G, and represent the way they are defined in terms of each other by means of two sets of rules like we did in the proof of proposition 2.2.3.2. Further add the two rules:

$$H(x) \rightarrow F(x)$$
$$H(x) \rightarrow G(x)$$

Left linearity is trivially satisfied. To see monotone termination, order the function symbols in the obvious way: $0 < S < F_0 < F_1 < \ldots < F$, $0 < S < G_0 < G_1 < \ldots < G$, $F < H$, $G < H$. One can verify that the representation of the definition of a primitive recursive function as a set of rewriting rules is decreasing in both the recursive and lexicographic path orderings given by this ordering on the function symbols.

Now, let F be the same primitive recursive function as G. Then we can 'calculate' the value of a term by evaluating it inside out. When reducing a term, the associated values of the terms in the reduction sequence clearly cannot change. This implies that each closed term has a unique normal form, and so the term rewriting system will be strongly closed confluent.

That the term rewriting system is not even weakly closed confluent in the case that F differs from G follows trivially by choosing some n for which $F(n) \neq G(n)$, and considering the one-step reducts $F(S^n(0))$ and $G(S^n(0))$ of the closed term $H(S^n(0))$.

Note that the term rewriting system given here will in general not be open confluent because the open terms F(x) and G(x) which are both reducts of the open term H(x) will in general not reduce, and thus will not have a common reduct.

2.4. Reduction strategies

In this section we will:

- Define a number of reduction strategies that will be mentioned in the description of compilation to Prolog in chapter 4, and list some of their properties

In section 2.1 we introduced the concept of a normalization function. It is a function that maps each closed term to one of its normal forms. Here we are interested in *reduction strategies*: they map each closed term to one of its reducts. A reduction strategy might lead to a normalization function (by iterating it until it reaches a normal form); however it need not to because this iteration might not terminate.

The four reduction strategies that will be briefly recapitulated are:

- leftmost innermost reduction
- leftmost outermost reduction
- parallel outermost reduction
- Gross-Knuth reduction

Parallel outermost reduction has the nice property that – under certain conditions – it is *normalizing*. This means that for terms having a normal form some normal form will be reached after finitely many steps.

Gross-Knuth reduction is not only normalizing, but is also a *cofinal* reduction strategy. This means that if a term does *not* have a normal form, reducing it under Gross-Knuth reduction gives an infinite reduction that will eventually be cofinal, which means that any reduct of a term in the sequence then can always be reduced back to a term in the sequence. The notion of a cofinal infinite reduction is conceptually the 'best' approximation to that of a normal form for terms that do not have a normal form.

2.4.1. Definition. A function R: $T_c(\Sigma) \to T_c(\Sigma)$ is called a *reduction strategy* if each closed term reduces to its image under R.

2.4.2. Definition. We will define a number of reduction strategies. Clearly, a reduction strategy has to map a normal form to itself, so in order to define these strategies it is sufficient to define what happens to terms that are not already in normal form.

A term rewriting system consists of a set of rewriting rules without an intrinsic ordering, but in this definition we will need such an ordering. So, suppose we have a term rewriting system T together with some total ordering on the rewriting rules.

Let $t \equiv f(t_1, t_2, \ldots, t_n)$ be a closed term that is not a normal form. Then if t is a redex – i.e., there is some rewriting rule $s \to s'$ and some substitution σ such that $t = \sigma[s]$ – and none of the t_i are reducible, then define R(t) to be the one-step reduct $\sigma[s']$. In order to remove the ambiguity which rule $s \to s'$ should be taken we use the order on the rewriting rules that we have introduced: the first rule that matches wins. Otherwise – t is not redex or one of its arguments is not in normal form – let R(t) be recursively defined as $f(t_1, t_2, \ldots, t_{i-1}, R(t_i), t_{i+1}, \ldots, t_n)$, in which $t_1, t_2, \ldots, t_{i-1}$ are all in normal form and in which t_i the first argument that is reducible. This choice of R defines *leftmost innermost reduction*.

If we reduce a non-redex in the same way, but reduce a redex regardless whether its arguments are in normal form or not, we get *leftmost outermost reduction*.

If furthermore when t is not a redex we recurse in all arguments simultaneously, i.e., use the construction $f(R(t_1), R(t_2), \ldots, R(t_n))$ instead of $f(t_1, t_2, \ldots, t_{i-1}, R(t_i), t_{i+1}, \ldots, t_n)$, we get *parallel outermost reduction*.

If we finally modify the definition of parallel outermost reduction still further by not defining $R(t) \equiv \sigma[s']$ when t is a redex but by taking $R(t) \equiv \sigma_R[s']$ in which σ_R is defined by $\sigma_R(x) = R(\sigma(x))$ we get *Gross-Knuth reduction*.

2.4.3. Definition. A term is called *linear* when a variable occurs in it at most once. A term rewriting system is called *left linear* when the left hand sides of its rules all are linear. A term rewriting system is called *orthogonal* when it is left linear and when the left hand sides of its rules only overlap trivially, i.e., each left hand side overlaps only with itself and only at the root.

2.4.4. Definition. A reduction strategy R is called *normalizing* if for each open term t that has some normal form there exists some n such that $R^n(t)$ is in normal form.

2.4.5. Proposition. *Parallel outermost reduction and Gross-Knuth reduction are both normalizing for orthogonal term rewriting systems.*

Proof. See [O'Donnell, 1977], [Klop, 1980].

2.4.6. Definition. A sequence of terms is called *cofinal* when each reduct of some element in the sequence in its turn has a reduct that is again an element in the sequence.

A reduction strategy R is called *cofinal* when for all closed terms t there exists some n such that $R^n(t)$, $R^{n+1}(t)$, $R^{n+2}(t)$, … is a cofinal sequence. It is easy to see that a cofinal reduction strategy is always normalizing.

2.4.7. Proposition. *Gross-Knuth reduction is cofinal for orthogonal term rewriting systems.*

Proof. See [O'Donnell, 1977], [Klop, 1980].

2.5. Persistence

In this section we will:
- Introduce modular specifications and define persistence
- Give the relation between strong and weak persistence
- Show how to check strong persistence in the left linear case
- Show the undecidability of weak persistence, even in the presence of some heavy restrictions

While termination and confluence are properties of *one* set of rules, persistence is a property of the *relation* between a number of such sets. Therefore, in order to define the notion of persistence, we first have to introduce the concept of a *modular specification*. Such a specification consists of a set called the modules of the specification, plus a relation between the elements of that set which is called the import relation. Furthermore, with each module M in the set is associated a module signature Σ_M and either (in the case of a modular equational specification) a module equational specification E_M or (in the case of a modular term rewriting system) a module term rewriting system T_M. Finally, these module signatures and module specifications should satisfy an inclusion property: if a module M′ imports a module M, then $\Sigma_M \subseteq \Sigma_{M'}$ and $E_M \subseteq E_{M'}$ (or, $T_M \subseteq T_{M'}$) should hold.

This definition does not represent the concept of *hiding*. While hiding is essential when writing real specifications, incorporating it would make the presentation of persistence more complicated, without any clear advantages: therefore, we did not clutter our definitions with it.

Now, what is usually called persistence, we will call *weak* persistence (in contrast to the notion of strong persistence, that we will also introduce). This is a property of a modular *equational* specification, which relates the initial algebras of the module

specifications. In a modular specification, even in one that is not persistent, there is for each import a natural homomorphism between the initial algebras of the module specifications. Precisely: If M′ imports M, then (because of the inclusion relations) we can define a homomorphism κ that maps $I(E_M)$ to $I(E_{M'})|\Sigma_M$. The definition of *weak persistence* now says that these homomorphisms should be isomorphisms, i.e., that they should be both surjective and injective. This means that the sorts in the initial algebra get no new members (because κ is a surjection; this goes under the slogan 'no junk'), and also that elements of a sort that are different in the imported module will not become equal in the importing module (because κ is a injection; 'no confusion').

In the previous sections we studied the relation between the set of normal forms of a term rewriting system and the initial algebra of its associated equational specification. The property of strong persistence is the analogon for the set of normal forms of a specification to the property of weak persistence for initial algebras. This means that the notion of strong persistence is defined as a property of modular term rewriting systems (while the notion of weak persistence was defined as a property of modular equational specifications).

A modular term rewriting system is called *strongly persistent* if, for all modules M′ importing a module M, the set of closed T_M normal forms is equal to the set of closed $T_{M'}$ normal forms whose type is a sort from Σ_M. This means that while in a weakly persistent specification the sorts in the specification should not change on import between modules, in a strongly persistent specification the objects in that sort should be given by the same normal forms (if at all) as well. Or: in a weakly persistent specification the specified objects do not change on import; in a strongly persistent specification even the normal forms representing them do not change.

One cannot simply say that strong persistence implies weak persistence. First of all, specifications that are strongly persistent are of a different type than specifications that are weakly persistent. Second, even when we consider the associated equational specification, it is not true that strong persistence implies weak persistence. But, if we add the requirement that the module term rewriting systems all have to be semi-complete, it *is* true. This is not surprising: we have the equations (the first and the third by semi-completeness, and the second because of strong persistence):

$$I(E(T_M)) = I_N(T_M) = I_N(T_{M'})|\Sigma_M = I(E(T_{M'}))|\Sigma_M$$

and all these equalities between algebras are canonical.

So, we now have introduced the notion of persistence, and we want to show that the notion of strong persistence is decidable (*if* we know that all the term rewriting systems are left linear). The major part of the rest of section will consist of the development of an algorithm to check this property.

If we look at the definition of strong persistence, we see that we have to compare sets containing closed normal forms. A problem with the approach of just trying to calculate these sets and then compare them is that in general they are infinite.

Now, in order to have a finite object that we can manipulate (which still resembles this infinite set of closed normal forms), we have to apply a 'trick': of each term we

only retain the upper part, so 'deep' in the term we replace subterms by variables. This gives us a *finite* set of *open* terms (because we replaced parts of the term by variables). These open terms we call *normal form approximations*.

Now, if we perform the approximation procedure of replacing subterms by variables in a sufficiently systematic way, we can arrange two term rewriting systems to have the same set of closed normal forms if and only if they have the same set of normal form approximations.

We also can find an algorithm for computing these sets of normal form approximations. The result of this computation is a finite set of terms, which is easily representable by virtue of it being a finite object.

Together, this shows that we have an effective way of comparing the sets of closed normal forms of two term rewriting systems: it consists of calculating the sets of normal form *approximations* (which are finite) and then comparing these sets. This gives us the somewhat surprising result that it is decidable whether two left linear term rewriting systems over the same signature have the same sets of closed normal forms. This result is surprising because generally it is *not* decidable whether two term rewriting systems have a given correspondence that is 'semantic'.

There are two details in this procedure that have to be made explicit:

- the choice of an approximation for each closed normal form, i.e., a method for systematically 'pruning' normal forms to normal form approximations
- the algorithm for calculating the set of normal form approximations (in fact we even want something that is stronger than just this algorithm: we want to be able to decide whether some given open term can be obtained by approximating a closed normal form)

The method for choosing the normal form approximations can go wrong in two ways. On the one hand, we can make the approximations too shallow (i.e., cut off too large parts of the term tree). In that case, we will lose essential information, and the equality of the sets of normal form approximations will not tell us anything. On the other hand, we can make the approximations too deep. This is not as bad, but in that case the sets of normal form approximations might become very large, and the algorithm might become very inefficient.

A first attempt to approximate normal forms is to 'cut off' all parts of the term that are 'deeper' than a given cut-off depth. In order for this to work, the cut-off depth must be greater than the maximal depth of all left hand sides in the relevant term rewriting systems. This approach is simple, but the sets of normal form approximations obtained in this way can easily become very large.

A second attempt is to let the top of the term tree – the outermost function symbol of the term – determine the cut-off depth. If a function f occurs only shallowly on the left hand side of the rewriting rules, it should not be necessary to have a deep approximation of terms with function symbol f.

The continuation of this line of thought leads to the following recursive definition. We form a set of *reference terms* REF, consisting of all left hand sides of all equations, together with all subterms of those terms.

Now, if we want to find the approximation of some term $f(t_1, t_2, \ldots, t_n)$ we select all terms with head function f from REF. Then, for all i, we (recursively) approximate the

argument t_i with respect to a *new* set of reference terms consisting of the i-th argument of those terms that have head function f.

This construction leads to two notions that will be introduced separately, before we introduce the approximation construction.

- The first notion is denoted: $R_{f,i}$. Given some set of terms R, one has to – first – select the terms in R with head function f, and – then – take the i-th arguments of those terms.
- The second notion is called the *basis* associated with a set of terms R, which we denote by R^π. It consists of the approximations of *all* closed terms with respect to the set R in the recursive manner that was sketched above. It is called a basis, because it has the property that each closed term matches exactly one of the open terms in R^π, so it gives a natural partitioning of the set of all closed terms.

  We argued before that the 'pruned' terms which are the elements of R^π should be 'deep enough'. The definition of R^π is framed in such a way that all terms in R^π are in a sense 'deeper' than the elements of R.

Now, from the basis REF^π we take those terms that are a normal form approximation. This gives a set called NFA. It is a finite set strongly resembling the set of closed normal forms of the term rewriting system. For instance, the function symbols that occur in this set are exactly the *generators* of the initial algebra corresponding to the term rewriting system, while the function symbols that do not occur in it are the *defined functions* of the specification.

We now turn to the algorithms given in this section. They are fairly straightforward. There are three algorithms which are built on top of each other. They are:

- The calculation of the set NFA of normal form approximations
- A decision procedure for deciding whether some term is a normal form approximation
- A procedure that compares the sets of closed normal forms of two term rewriting systems

The idea behind the algorithm for calculating the set NFA is not too difficult. Instead of calculating *one* set – the set NFA itself – directly, we will calculate a sequence of sets that will steadily increase to the set we are looking for. Each set in this sequence will contain the approximations of deeper closed normal forms. When the sequence becomes constant, we have found the full set NFA.

The construction of this sequence is as follows. We start with the empty set. At each iteration we form all terms that have arguments from the previous set in the sequence (initially we will of course only be able to form constants). If such a term is reducible, it is not interesting. Else, we approximate it and add it to the set.

The main application of this set NFA is in the algorithm that decides whether a given open term is a normal form approximation. It turns out that this gives a powerful technique for investigating the set of normal forms of a term rewriting system. We call this technique *normal form analysis*.

The decision procedure that decides whether some term is a normal form approximation works in the following way. Suppose that one is given some *template* term, and that one wants to decide whether there is a closed normal form that matches it. In order to do this, one looks at all elements from the set NFA that unify

with the template. If one of these unifying elements of NFA is an instance of the template, we are finished and may conclude that the template approximates some closed normal form. If this is not yet true, we go in recursion: we calculate the unification and invoke the algorithm on the arguments of this unification.

Once one has the ability to decide whether a given open term is a normal form approximation, checking strong persistence is not very hard. In order to check that there is no *junk*, one just has to verify that there are no normal forms containing a 'new' function symbol, i.e., a function symbol that was introduced after the import. This can easily be done by checking whether there exists some left hand side $f(t_1, t_2, \ldots, t_n)$ of a rule in the new, importing, module for which f is such a new function and for which all t_i are normal form approximations in the original, imported, module. In order to check that there is no *confusion*, one has to verify that all 'new' left hand sides, i.e., left hand sides of rewriting rules introduced after the import, are not normal form approximations in the original module.

The complexity of this algorithm for checking strong persistence turns out to be rather bad in the worst case: it uses at least exponential space. However, in practice the algorithm appears feasible for checking the persistence of moderately sized specifications as the examples in chapter 3 show.

We end this section with a proof of the fact that it is not decidable whether a modular specification is *weakly* persistent, even if all module specifications are:

> monotone terminating with respect to some given path ordering + open confluent + left linear

This is proved by associating a modular term rewriting system to a primitive recursive function F and choosing the construction in such a way that this modular term rewriting system is weakly persistent if and only if F is a bijection.

2.5.1. Weak and strong persistence

2.5.1.1. Definition. A *modular equational specification* consists of a set called the *set of modules* of the specification, a relation on this set called the *import relation*, and for each module M in the specification a *module signature* Σ_M and a *module equational specification* E_M, such that E_M has signature Σ_M, and such that when a module M′ imports a module M the inclusions $\Sigma_M \subseteq \Sigma_{M'}$ and $E_M \subseteq E_{M'}$ both hold.

A *modular term rewriting system* consists of a *set of modules*, an *import relation* on this set, and for each module M in the specification a *module signature* Σ_M and a *module term rewriting system* T_M, such that T_M has signature Σ_M, and such that when a module M′ imports a module M the inclusions $\Sigma_M \subseteq \Sigma_{M'}$ and $T_M \subseteq T_{M'}$ both hold.

Each modular term rewriting system has an *associated* modular equational specification that is obtained by taking the same set of modules, the same import relation, the same module signatures Σ_M, and by taking for the module equational specification E_M the equational specification $E(T_M)$ associated with T_M.

2.5.1.2. Definition. Let be given some modular equational specification. Then, if a module M′ imports a module M we can define a homomorphism κ that maps $I(E_M)$ to $I(E_{M'})|\Sigma_M$ in the following way. Each element of $I(E_M)$ is by the definition of initial algebra an equivalence class of closed terms over Σ_M. Let t be such a closed term and let [t] be its equivalence class. The term t is also a term over the larger signature $\Sigma_{M'}$ and as such is in some equivalence class [t] in the initial algebra $I(E_{M'})$. Because t is a term over the signature Σ_M, it will be clear that the [t] is an element of $I(E_{M'})|\Sigma_M$. Now, define the mapping κ by

$$\kappa([t]) \equiv [t]$$

Because $E_M \subseteq E_{M'}$, it will be clear that this definition of $\kappa([t])$ does not depend on the representative t of [t] that is used in the definition. It is also straightforward to verify that the mapping κ that is defined in this way is a homomorphism.

The modular equational specification is called *weakly persistent* when for each import the homomorphism κ is an isomorphism, i.e., when for each import the mapping κ is both surjective and injective.

2.5.1.3. Definition. A modular term rewriting system is called *strongly persistent* if, for all modules M′ importing a module M, the set of closed T_M normal forms is equal to the set of closed $T_{M'}$ normal forms whose type is a sort from Σ_M.

2.5.1.4. Proposition. *If a modular term rewriting system is strongly persistent, and all module term rewriting systems are semi-complete, the associated modular equational specification is weakly persistent.*

Proof. According to 2.3.1.6 the fact that each module term rewriting system T_M is semi-complete implies that each closed term t over Σ_M has a T_M normal form, and that for two closed terms t_1 and t_2 the normal forms are equal if and only if $[t_1] = [t_2]$ in $I(E_M)$.

Now, in order to prove weak persistence, we have to show that for each import of a module M in a module M′ the mapping κ is both surjective and injective.

Surjective: Let [t] be some element from $I(E_{M'})|\Sigma_M$. This means that t is a closed term over the signature $\Sigma_{M'}$ typed by a sort in Σ_M. The term t has a $T_{M'}$ normal form n to which it is provably equal in $E_{M'}$. Now by strong persistence n is also a closed T_M normal form. But this means that n is a closed term over the signature Σ_M, so [n] is an element of $I(E_M)$. This means that [t], which is equal to [n] in $I(E_{M'})|\Sigma_M$ is the κ-image of [n] in $I(E_M)$.

Injective: Let $[t_1]$ and $[t_2]$ be two elements of $I(E_M)$ that map to the same element of $I(E_{M'})|\Sigma_M$, which means that the closed terms t_1 and t_2 are provably equal in $E_{M'}$. They have in T_M the normal forms n_1 and n_2, which because of strong persistence are also $T_{M'}$ normal forms, and because $T_M \subseteq T_{M'}$ they are also the $T_{M'}$ normal forms of t_1 and t_2. But, because t_1 and t_2 are provably equal in $E_{M'}$ they have the same normal form, so $n_1 = n_2$. But this means that that t_1 and t_2 had the same T_M normal form, so

they were also provably equal in T_M, and so $[t_1] = [t_2]$ in $I(E_M)$.

2.5.2. Term approximations and bases

2.5.2.1. Definition. In the rest of section 2.5 we will only be interested in linear terms. Therefore we will not work here with terms in which variables have a name (which is not very relevant here because all terms are known to be linear and we do not need to able to apply a substitution) but with terms in which all variables have been replaced by the symbol ♦, which is pronounced 'hole'. One should realize that the term f(♦, ♦) does not correspond to a non-linear term like f(x, x); instead it corresponds to a term like f(x, y). So, each ♦ denotes a *different* variable.

The set $T(\Sigma, \blacklozenge)$ of *linear open terms* or (for the remainder of this section) *terms* is the minimal set such that:

(i) for each sort σ it contains the hole $\blacklozenge_\sigma$ of type σ. If the type of a hole is clear from the context we will omit the subscript σ and just write ♦.

(ii) if f is a function symbol, and $t_1, t_2, \ldots, t_n$ are terms from $T(\Sigma, \blacklozenge)$ of the appropriate types, then $t \equiv f(t_1, t_2, \ldots, t_n)$ is also a term in $T(\Sigma, \blacklozenge)$.

This definition clearly mimics 2.1.1.3. Most of the notions in 2.1 concerning the set $T(\Sigma, V)$ have counterparts for the set $T(\Sigma, \blacklozenge)$. We will not treat this distinction formally, so, we will talk about 'subterms', 'contexts', etc, and consider the specific definitions for linear open terms understood.

2.5.2.2. Definition. A term s is a *approximation* of a term t when it is obtained from t by replacing a number of subterms of t by a hole ♦ of the appropriate type. Formally:

(i) a hole $\blacklozenge_\sigma$ is an approximation of all terms t of type σ

(ii) a term $f(s_1, s_2, \ldots, s_n)$ is an approximation of a term $f(t_1, t_2, \ldots, t_n)$ when all arguments s_i are approximations of the corresponding t_i.

If s is an approximation of t, the term t is called an *instance* of s.

A term is called a *normal form approximation* when it is an approximation of some *closed* normal form. Note that an approximation of an *open* normal form need not be a normal form approximation. For instance, consider the term rewriting system given by the following Perspect specification:

```
external Addition
 import Naturals
 function add(NAT,NAT): NAT
internal Addition
 variable m, n: NAT
 equation
  add(0,n): n
  add(succ(m),n): succ(add(m,n))
```

Then the term t ≡ `add(♦,♦)` is an open normal form because there does not exist a rule that can reduce t. However, there is no closed normal form that is an instance of t

because all closed normal forms have the form $\texttt{succ}^n\texttt{(0)}$. So t is not a normal form approximation.

2.5.2.3. Definition. Let R be some set of terms. Let f be a function symbol and i some index that corresponds to an argument position of f. Then the set $R_{f,i}$ is obtained by selecting the elements from R that have the outermost function symbol f, and collecting the set of the i-th argument of each element from this subset. This operation is called *setwise argument selection*. More formally:

$$R_{f,i} \equiv \{ t \mid \exists\ s = f(s_1, s_2, \ldots, s_n) \in R\text{: } t = s_i \}$$

2.5.2.4. Definition. A set of terms B is called a *basis* when each closed term in $T_c(\Sigma)$ is an instance of exactly one element of B.

2.5.2.5. Definition. Let R be some set of terms called *reference terms*. The set R^π called the *basis associated with* R is defined as the union of sets $(R^\pi)_\sigma$ for each type σ. The definition of the sets $(R^\pi)_\sigma$ has two cases:

(i) if R does not contain non-hole terms of type σ, then $(R^\pi)_\sigma \equiv \{\blacklozenge_\sigma\}$

(ii) if R contains a non-hole term of type σ we define the set $(R^\pi)_\sigma$ recursively as:

$$(R^\pi)_\sigma \equiv \cup_{f\in F(\Sigma),\ ran(f)=\sigma} \{ t = f(t_1, t_2, \ldots, t_n) \mid \forall\ i\text{: } t_i \in (R_{f,i})^\pi \}$$

Note that the set R^π may contain terms of different types, and that the definition of R^π not only depends on the set R but also on the signature Σ.

2.5.2.6. Proposition. *The set* R^π *is a basis.*

Proof. Let $t \equiv f(t_1, t_2, \ldots, t_n)$ be some closed term of type σ. We have to prove that R^π contains exactly one term that approximates t. This is proved using induction on the depth of t. There are two cases (because the definition of $(R^\pi)_\sigma$ has two cases). Either $(R^\pi)_\sigma = \{\blacklozenge_\sigma\}$ in which case the claim is trivially true, or we are in case (ii) of the definition of R^π. Then induction gives that each t_i has exactly one approximation s_i in $(R_{f,i})^\pi$, and then $s = f(s_1, s_2, \ldots, s_n)$ is the unique approximation of t in R^π.

2.5.2.7. Proposition. *Let be given terms* $s \in R^\pi$ *and* $t \in R$. *Then the fact that* s *does unify with* t *implies that* s *is an instance of* t.

Proof. This proposition is proved using induction on the depth of s and t. The case that t is a hole is trivial – every term is an instance of a hole – so suppose t has the form $f(t_1, t_2, \ldots, t_n)$. Then s cannot be a hole – because t is not a hole we are in case (ii) in the definition of R^π – so because s and t unify we find that s also has the form $f(s_1, s_2, \ldots, s_n)$. We know that each $s_i \in (R^\pi)_{f,i} = (R_{f,i})^\pi$ and $t_i \in R_{f,i}$, so induction gives that each s_i is an instance of the corresponding t_i. But that means s is an instance of t.

2.5.2.8. Proposition. *Let be given a set* R_1 *that contains a set* R_2. *Then each term in* $(R_1)^\pi$ *is approximated by some term in* $(R_2)^\pi$.

Proof. Let t be some term of type σ in $(R_1)^\pi$. If t is a hole, R_1 does not contain non-hole elements of type σ, and nor will R_2, so the hole of type σ will also be in $(R_2)^\pi$ and R_2 contains an approximation of t. If t is not a hole, say $t = f(t_1, t_2, \ldots, t_n)$, then for each i we may conclude, with induction, from the fact that $(R_1)_{f,i}$ contains $(R_2)_{f,i}$ that the argument t_i – being in $((R_1)^\pi)_{f,i} = ((R_1)_{f,i})^\pi$ – has an approximation s_i in $((R_2)_{f,i})^\pi$. And then the term $s = f(s_1, s_2, \ldots, s_n)$ is the required approximation of t.

2.5.2.9. Proposition. *Let be given a set* R_1 *that contains a set* R_2 *and let be given terms* $s \in (R_1)^\pi$ *and* $t \in (R_2)^\pi$. *Then the fact that* s *unifies with* t *implies that* s *is an instance of* t.

Proof. This proposition is proved using induction on the depth of s and t. The case that t is a hole is trivial, so let t have the form $f(t_1, t_2, \ldots, t_n)$. Then R_2 has to contain a non-hole term of the same type as t, and therefore so has R_1. Moreover, because s and t unify, s also has the form $f(s_1, s_2, \ldots, s_n)$. Now, $((R_1)_{f,i})^\pi$ contains $((R_2)_{f,i})^\pi$, furthermore we know that $s_i \in ((R_1)^\pi)_{f,i} = ((R_1)_{f,i})^\pi$ and $t_i \in ((R_2)^\pi)_{f,i} = ((R_2)_{f,i})^\pi$ and finally s_i unifies with t_i. Therefore, induction tells us that s_i is an instance of t_i. This shows that s is an instance of t.

2.5.3. Normal form analysis

2.5.3.1. Definition. Let T be some left linear term rewriting system. Then define the set of reference terms REF associated with T as the set of all subterms of the left hand sides LHS of rules from T.

Now the set NFA of *normal form approximations associated with* T is defined as the elements from REF^π that are normal form approximations. More formally:

$$NFA \equiv \{ t \in REF^\pi \mid \exists \text{ normal form } n \in T_c(\Sigma): t \text{ is an approximation of } n \}$$

2.5.3.2. Algorithm. *Procedure for calculating the set* NFA.

Construct a sequence of sets NFA_0, NFA_1, NFA_2, … in the following way. Let $NFA_0 \equiv \varnothing$ be the empty set. Now if NFA_{i-1} is given, construct NFA_i as follows. Form all terms of the form $f(t_1, t_2, \ldots, t_n)$ in which the t_i are taken from NFA_{i-1} and for each of these terms take the approximation that is in REF^π. Only retain the approximations which are an open normal form (i.e., which are not an instance of a left hand side of a rule of T). These are the elements of the set NFA_i.

The sequence that one obtains in this way will after some index n become constant, i.e., for some index n we have $NFA_n = NFA_{n+1} = NFA_{n+2} = \ldots$ The set NFA will be equal to this 'limit set' NFA_n.

Proof *of the correctness of the algorithm.* The algorithm contains the hidden assumption that the terms of the form $f(t_1, t_2, \ldots, t_n)$ occurring in the inductive construction of the sets NFA_i all have an approximation in REF^π. That this is true follows from the fact that REF contains all subterms of its elements, because it consists of all subterms of elements from LHS. Therefore REF contains $REF_{f,i}$ for each i. Now proposition 2.5.2.8 shows that all t_i – which have been taken from NFA_{i-1} and therefore are in REF^π – have some approximation in $(REF_{f,i})^\pi$. And this means that $f(t_1, t_2, \ldots, t_n)$ has an approximation in REF^π.

We will now give a characterization of the sets NFA_i. We claim that each NFA_i consists of the approximations in REF^π of the closed normal forms that have depth less than i. We prove this using induction. For $i = 0$ this is easy: NFA_0 is empty and there are no natural numbers less than 0.

The induction step has two directions: given some closed normal form of depth less than i we have to show that its approximation is in NFA_i and vice versa. So, let be given some closed normal form $n = f(n_1, n_2, \ldots, n_n)$ of depth less than i. Then, its arguments n_i are normal forms of depth less than i-1, so *their* REF^π approximations t_i are in NFA_{i-1}. We define $t = f(t_1, t_2, \ldots, t_n)$ and let t' be the REF^π approximation of t. Now, t', being an approximation of the normal form n, will itself be an open normal form. So t' which is the approximation of n will end up in NFA_i as desired.

In the other direction, let be given some element $t = f(t_1, t_2, \ldots, t_n)$ of NFA_i. The t_i are approximations of elements from NFA_{i-1}, which by induction are approximations of normal forms n_i of depth less than i-1. But then t is an approximation of $n = f(n_1, n_2, \ldots, n_n)$. Now, suppose that n is not a normal form. Then t unifies with the left hand side s of the rule that reduces n. According to proposition 2.5.2.7, we may conclude from the fact that some element t of REF^π unifies with some element s of LHS (which is a subset of REF) that t is in fact an instance of s. This shows that t is not in normal form, which contradicts the construction of NFA_i. So n in fact is a normal form, and therefore t is an approximation of a normal form of depth less than i.

Finally we will give the proof of the correctness of the algorithm. Because the construction of the set NFA_i only depends on the set NFA_{i-1} and this dependency is monotone with respect to set inclusion, the sequence of sets is increasing. However, each set is a subset of the set REF^π, which is finite. So, after finitely many steps we reach the point that $NFA_n = NFA_{n+1}$ and then the NFA_i's cannot change anymore. The characterization that we have given of the NFA_i then shows that the set NFA_n is equal to NFA.

2.5.3.3. Algorithm. *Procedure for deciding whether some term* t *is a normal form approximation.*

There are four cases:

(i) The first case is the one that t is a hole. In this case, check whether NFA contains some term of the same type as t. If it does, t is a normal form approximation; else it is not.

In the next three cases t is not a hole so t has the form $f(t_1, t_2, \ldots, t_n)$.

(ii) NFA does not contain a term that unifies with t. In this case t is not a normal

form approximation.

(iii) NFA contains a hole of the same type as t. Check – by going into recursion – whether each argument t_i is a normal form approximation. If they all are, t is also a normal form approximation; else it is not.

(iv) In this last case, NFA contains a non-hole term that unifies with t. Such a term clearly has the form $s = f(s_1, s_2, \ldots, s_n)$. Now, for each argument position i of f, first check whether t_i is an approximation of s_i. If this is not the case, decide – by going into recursion – whether the unification of t_i and s_i is a normal form approximation. If there is an element s of NFA that unifies with t, for which either of these checks succeeds for all i, the term t is a normal form approximation; else it is not.

Proof *of the correctness of the algorithm***.** There is a hidden assumption in this algorithm as well, which is that the algorithm will not go into infinite recursion. We first show that this will not happen. If the algorithm invokes itself, it always does this by unifying the term t with an element of NFA, and then selecting one of the arguments (actually, in case (iv) of the algorithm these two steps are performed in a different order, but because all terms are linear this makes no difference). Let the sequence of elements in this recursive invocation lead to the sequence $s_0, s_1, s_2, \ldots$ in which each s_i is the term after unification and before argument selection. Also, let the sequence in which each time only the argument is being selected (omitting the unification) be called $t_0 = t, t_1, t_2, \ldots, t_n = \blacklozenge$.

We claim that each s_i is the unification of t_i with some element u_i of NFA. For $i = 0$ this is evident. In order to prove the induction step, one has to verify that the unification of some element of NFA and an argument of some element of NFA gives an element of NFA. This is a consequence of the fact that all subterms of elements of REF are also an element of REF, which means that we may take $R_1 = \mathrm{REF}$ and $R_2 = \mathrm{REF}_{f,i}$ in proposition 2.5.2.9. This proposition then shows that if some element s of NFA (which will also be an element of REF^{π}) and the argument t of some element of NFA (this argument then is an element $(\mathrm{REF}^{\pi})_{f,i} = (\mathrm{REF}_{f,i})^{\pi}$) unify, s will be an instance of t, and so their unification will be s which is an element of NFA.

To finish the proof of the fact that this algorithm will not go into infinite recursion: We have shown that each s_i is the unification of t_i with some element of NFA. This means that because $t_n = \blacklozenge$, it follows that s_n is an element of NFA, and this means that at that point the recursion will end.

And now for the correctness of the algorithm. We have to prove that the algorithm succeeds if and only if it has been given some normal form approximation.

'$\Rightarrow$': We use induction on the running time of the algorithm. This means that when the algorithm goes into recursion, we may assume that in that case the answer will be correct. Now, let t be some term of type σ for which the algorithm succeeds. It is easy to see that in the first two cases of the algorithm, it will give the correct answer. If the algorithm succeeds in the third case, we know that t has the form $f(t_1, t_2, \ldots, t_n)$ and that each t_i is an approximation of some closed normal form n_i. Now suppose that $n = f(n_1, n_2, \ldots, n_n)$ is reducible. Then n must be a redex, but because NFA only contains a hole of the type σ, there are no left hand sides of rewriting rules that have type σ. So, n

is not a redex but a closed normal form, and t, being an approximation of n, is indeed a normal form approximation.

If we are in the fourth case of the algorithm, t again has the form $f(t_1, t_2, \ldots, t_n)$, and we also have some $s = f(s_1, s_2, \ldots, s_n)$ in NFA such that for all i the unification of t_i and s_i is a normal form approximation – say the approximation of some closed normal form n_i. Now, define $n = f(n_1, n_2, \ldots, n_n)$ and suppose that n is reducible. Then n must be a redex, i.e., it must unify with some left hand side u of some rewriting rule. Now s is an approximation of n, so s also unifies with u. We further know that $u \in$ LHS and that $s \in \mathrm{REF}^{\pi}$. From this it follows (as was shown in the proof of 2.5.3.2) that u approximates s. But that means that s is reducible, i.e., s cannot be a normal form approximation, which means it cannot be in NFA. This contradiction show that in fact n was a normal form, and so t which approximates n is indeed a normal form approximation.

'$\Leftarrow$': Again we use induction on the running time of the algorithm. Suppose we are given some normal form approximation t which is the approximation of the closed normal form n. By the definition of NFA the set NFA also has to contain the approximation s of n in REF^{π}. Now, if we are in the first case of the algorithm, because s exists we get the right answer. Again, because of the existence of s we cannot be in the second case of the algorithm. In the third case we will also get the correct answer, because each argument of t approximates the corresponding argument of n, so each argument of t is a normal form approximation, and by induction the algorithm will succeed for it. In the fourth case the approximation s of n in NFA will of course work: We know that t has the form $f(t_1, t_2, \ldots, t_n)$, s has the form $f(s_1, s_2, \ldots, s_n)$ and n has the form $f(n_1, n_2, \ldots, n_n)$. If some argument t_i is not an approximation of s_i, the recursion will – by induction – still give an affirmative answer, because both t_i and s_i approximate n_i, so their unification also does.

2.5.3.4. Example. For a small example, consider the term rewriting system T given by the following Perspect specification:

```
external Naturals
 sort NAT
 function
  0, succ(NAT): NAT
internal Naturals
 [empty]

external Integers
 import Naturals
 sort INT
 function
  pos(NAT), neg(NAT): INT
internal Integers
 equation
  neg(0): pos(0)
```

The infinite set of closed normal forms of T is:

{0, succ(0), succ(succ(0)), succ(succ(succ(0))), ...,
pos(0), pos(succ(0)), pos(succ(succ(0))), ...,
neg(succ(0)), neg(succ(succ(0))), neg(succ(succ(succ(0)))), ...}

With the term rewriting system T the following sets are associated:

LHS = {neg(0)}
REF = {0, neg(0)}
REF^{π} = {0, succ($\blacklozenge_{NAT}$), pos($\blacklozenge_{NAT}$), neg(0), neg(succ($\blacklozenge_{NAT}$))}
NFA = {0, succ($\blacklozenge_{NAT}$), pos($\blacklozenge_{NAT}$), neg(succ($\blacklozenge_{NAT}$))}

Note that one can partition the set of closed terms according to the elements of the basis REF^{π} and that one can partition the set of closed normal forms according to the normal form approximations in the set NFA.

2.5.4. Decidability

2.5.4.1. Algorithm. *Procedure for deciding whether two left linear term rewriting systems* T_1 *and* T_2 *over the signatures* Σ_1 *and* Σ_2 *have the same set of closed normal forms with a type in* $S(\Sigma_1) \cap S(\Sigma_2)$.

In the first place, check that all closed normal forms with a type in $S(\Sigma_1) \cap S(\Sigma_2)$ involve in both T_1 and T_2 only functions from $F(\Sigma_1) \cap F(\Sigma_2)$. In order to do this verify (using algorithm 2.5.3.3) that all terms $f(\blacklozenge, \blacklozenge, \ldots, \blacklozenge)$ with f in $F(\Sigma_1) \setminus F(\Sigma_2)$ and ran(f) in $S(\Sigma_1) \cap S(\Sigma_2)$ are not normal form approximations of T_1 and vice versa.

In the second place, check (again, using algorithm 2.5.3.3) that all left hand sides of T_2 with a type in $S(\Sigma_1) \cap S(\Sigma_2)$ are not normal form approximations of T_1 and vice versa.

Proof *of the correctness of the algorithm***.** The set of the closed T_1 normal forms consists of two parts: those with an outermost function symbol in $F(\Sigma_1) \setminus F(\Sigma_2)$ and those with an outermost function symbol in $F(\Sigma_1) \cap F(\Sigma_2)$. The first check verifies that this first part is empty. This implies that T_1 normal forms cannot contain any function symbol from $F(\Sigma_1) \setminus F(\Sigma_2)$ (because a subterm of a normal form is itself a normal form). So, after the first check we know that the closed normal forms of T_1 and T_2 with types in $S(\Sigma_1) \cap S(\Sigma_2)$ only contain function symbols from $F(\Sigma_1) \cap F(\Sigma_2)$.

Suppose that some T_1 normal form is reducible in T_2. The subterm which reduces is a T_1 normal form and a T_2 redex. But then the corresponding T_2 rewriting rule has a left hand side that is a T_1 normal form approximation. The second check verifies whether that happens.

We have now shown that if the algorithm says that the sets of normal forms are

equal, it is right. On the other hand, if they *are* equal, it is easy to verify that the algorithm will succeed.

2.5.4.2. Algorithm. *Procedure for deciding whether a modular left linear term rewriting system is strongly persistent.*

For all modules M′ importing a module M compare the sets of closed normal forms with a type in $\Sigma_M = \Sigma_M \cap \Sigma_{M'}$ of the term rewriting systems T_M and $T_{M'}$ using algorithm 2.5.4.1.

2.5.4.3. Proposition. *The previous algorithm uses space at least exponential in the length of the specification.*

Proof. Consider the following Perspect specification scheme (which is inspired by the examples in [Bergstra, Mauw, Wiedijk, 1989]):

```
external HypercubeWithOneCornerRemoved
 sort B, C   ["Bit" and "Cube"]
 function
  0, 1: B
  f(B,B,...,B): C
internal HypercubeWithOneCornerRemoved
 equation
  f(1,1,...,1): f(0,0,...,0)
```

The ellipses should be expanded such that the function `f` becomes an n-ary function. Clearly this specification has length of order n. The set NFA consists of all terms of the form $f(b_1,b_2,\ldots,b_N)$ with all b_i in {0, 1} and at least one $b_i \neq 0$ and so clearly has size exponential in n. Now algorithm 2.5.4.2 uses this set, and so uses exponential space to store it.

2.5.4.5. Proposition. *It is not decidable whether a modular equational specification is weakly persistent, even when it is known that it corresponds to a modular term rewriting system in which each module term rewriting system is monotone terminating with respect to both a recursive and a lexicographic path ordering, is open confluent and is left linear.*

Proof. For an arbitrary primitive recursive function F we are going to define a modular term rewriting system for which each module term rewriting system is monotone terminating with respect to both recursive and lexicographic path orderings, is open confluent and is left linear. Furthermore, it will be chosen in such a way that if F is a bijection, its associated modular equational specification is weakly persistent, while if F is not a bijection, the specification is not weakly persistent. Because it is not decidable whether some arbitrary primitive recursive function is a bijection, it is not decidable whether the specification is weakly persistent.

The first module of this specification has two sorts: the natural numbers and a second sort called X. It has three functions: the generators 0, S of the natural numbers and a function G that maps the natural numbers on X. In this module there are no rewriting rules.

In the second module there are no new sorts. There are a number of new functions: the sequence of functions F_0, F_1, …, $F_n \equiv F$ defined on the natural numbers and leading to the primitive recursive function F as in 2.2.3.2 and 2.3.3.2, and a new function H mapping the natural numbers on X. There are the set of rules 'defining' F in terms of the sequence, and one additional rewriting rule:

$$G(x) \rightarrow H(F(x))$$

Now, the ordering on the function symbols for which this is a monotone terminating term rewriting system is of course $0 < S < F_0 < F_1 < \ldots < F$, $F < G$, $H < G$. (Note that it is impossible to do this in Perspect, because there functions in 'later' modules always have to be 'larger'. In fact, in Perspect weak persistence is equivalent to strong persistence.) Open confluence and left linearity are also trivial.

Now, before the import, in the initial algebra the sort X is the set {[G(0)], [G(S(0))], [G(S(S(0)))], …}, while after the import it becomes {[H(0)], [H(S(0))], [H(S(S(0)))], …}, with the mapping κ from definition 2.5.1.2 given by $\kappa([G(x)]) = [H(F(x))]$. So in a sense κ is given by F, and so κ is a bijection if and only if F is. This shows that the import is weakly persistent if and only if F is a bijection.

2.6. Primitive recursive algebras

In this section we will:

- Introduce the notion of a primitive recursive algebra
- Give the construction of a 'diagonal algebra' that is not primitive recursive but can be specified in the presence of some heavy restrictions

The class of primitive recursive algebras is for algebras what the class of primitive recursive functions is for functions. Just like each primitive recursive function is a total recursive function whereas a lot of everyday functions are primitive recursive, each primitive recursive algebra is a decidable semi-computable algebra while a lot of everyday algebras are primitive recursive algebras.

The definition of primitive recursive algebra that is given here is somewhat *ad hoc*. It does not have the property that the minimal subalgebra of a primitive recursive algebra is necessarily primitive recursive, which is not very satisfying. One can ask the analogous question here for 'Perspect algebras': is the minimal subalgebra of an algebra that is specifiable in Perspect again specifiable in Perspect? The answer to this question is unknown to me, though the analogy with the class of primitive recursive algebras suggests that this is not the case.

We use the notion of a primitive recursive algebra to give a lower bound to the expressiveness of specifications satisfying all the restrictions that gave the decidable

class of persistent specifications. We find that the following inclusions hold:

> primitive recursive algebras $\subset$ algebras specifiable under the restrictions of the decidable class of persistent specifications $\subset$ algebras in which equality between terms is decidable $\subset$ semi-computable algebras

In order to prove that the first inclusion is proper, we need an algebra that can be specified under the restrictions of the decidable class of persistent specifications, while not being primitive recursive. This algebra is the subject of the sole proposition of this section. It is constructed from a diagonal function δ that has to satisfy a certain diagonal property, which can easily be satisfied by some diagonal construction.

2.6.1. Definition. A *primitive recursive algebra* is an algebra that is isomorphic to an algebra in which all sorts are either the set of all natural numbers or a set of the form {0, 1, …, n-1}, and in which all functions are primitive recursive.

2.6.2. Proposition. *There is a term rewriting system that is monotone terminating, open confluent and left linear, but whose initial algebra is not primitive recursive.*

Proof. A term rewriting that satisfies this proposition is given by the Perspect specification in 3.3.3. Because this is a Perspect specification it is trivially monotone terminating, open confluent and left linear.

It contains (among other things) a sort containing the natural numbers – given by generators 0 and S – and a function δ from the natural numbers to the natural numbers that has the property that for all primitive recursive functions F and G, with F a bijection on the natural numbers, the function δ is not equal to the composition $F^{-1}\cdot G$. This equivalent to the property that for each primitive recursive F and G, with F a bijection, there exists some natural number n with $F(\delta(n)) \neq G(n)$. For the explanation why this is true see 3.3.3.

Suppose that the initial algebra of this specification is primitive recursive. Then it has to be isomorphic to an algebra in which all sorts are either the set of all natural numbers or a set of the form {0, 1, …, n-1}, and in which all functions are primitive recursive – its *primitive recursive representation*. Call the primitive recursive functions that correspond to 0, S and δ in this representation $0'$, S' and δ'.

The component of the isomorphism for the natural numbers is a bijection i from the natural numbers to the natural numbers that satisfies $i(S(n)) = S'(i(n))$. From this it follows that $i(n) = i(S^n(0)) = S'^n(0')$, which implies that i is itself also a primitive recursive function because it can be defined by primitive recursion in terms of S' and $0'$, which are primitive recursive.

Also, we know that $i(\delta(n)) = \delta'(i(n))$ or, equivalently, $\delta(n) = i^{-1}(\delta'(i(n)))$. This means that $\delta = i^{-1}\cdot(\delta'\cdot i)$. Now define, $F \equiv i$ and $G \equiv \delta'\cdot i$. Both functions are primitive recursive, and F is a bijection. This gives a contradiction with the property that δ cannot have the form $F^{-1}\cdot G$.

Chapter 3

Perspect

If it has syntax, it isn't user-friendly.

In this chapter we will define an algebraic specification language called Perspect. This language will be shown to have a number of desirable properties. A couple of interesting example specifications written in Perspect will be given. Finally, an implementation of a checker for Perspect written in Modula-2 will be described.

3.1. Language definition

Section 3.1 of this thesis is the revised report on the specification language Perspect. In this report both the language Perspect itself is defined, as well as a range of five dialects of Perspect. The main language is defined in sections 3.1.1 to 3.1.5. The dialects are defined in section 3.1.6.

3.1.1. Syntax

3.1.1.1. Lexical syntax. Perspect has nine keywords which are written using lower case letters. The keywords are reserved, and cannot be used as an identifier.

⟨keyword⟩ ::=
'**echo**' | '**equation**' | '**external**' | '**function**' | '**import**' |
'**internal**' | '**rec**' | '**sort**' | '**variable**'.

Perspect identifiers are strings of the form:

⟨ident⟩ ::= (⟨lower case letter⟩ | ⟨upper case letter⟩ | ⟨digit⟩)$^+$

different from the nine reserved keywords. Upper case letters and lower case letters are considered to be different.

Perspect has five symbols for punctuation: the comma, the round brackets, the colon and the asterisk.

⟨punctuation mark⟩ ::= ‘`,`’ | ‘`(`’ | ‘`)`’ | ‘`:`’ | ‘`*`’.

The following lexical elements are used for layout and have no further meaning:

⟨layout element⟩ ::= ⟨comment⟩ |
⟨space⟩ | ⟨horizontal tab⟩ | ⟨vertical tab⟩ | ⟨newline⟩ | ⟨carriage return⟩ | ⟨form feed⟩.

A layout element should not occur inside another lexical element (like an identifier), but is meant to separate lexical elements.

Comments are delimited by square brackets ‘`[`’ and ‘`]`’. Comments may be nested. This makes it possible to temporarily remove some part of a specification (even if it already contains other comments) by placing the text of this part between comment brackets.

⟨comment⟩ ::= ‘`[`’ (⟨printable character ≠ square bracket⟩ | ⟨comment⟩)* ‘`]`’.

Comments are only permitted to contain printable ASCII characters, i.e., characters with an ASCII value in the set:

{9, 10, 11, 12, 13, 32, …, 90, 92, 94, …, 126}

(The ASCII values of the characters ‘`[`’ and ‘`]`’ are respectively 91 and 93). Characters from an extended ASCII character set, i.e., with a value larger than 127, are not allowed in a comment.

Comments that start with the character ‘-’ (hyphen) are reserved as ‘pragma’. These comments should be used when it is necessary to communicate information to an implementation of Perspect. For example, a comment like:

```
[-implementation Integers.MOD]
```

could be used to tell a Perspect compiler not to compile a module, but to use the manually written implementation in the file `Integers.MOD` instead. Note that this kind of comment is not allowed to alter the correctness or the meaning of a specification.

3.1.1.2. Concrete syntax. A Perspect specification consists of a number of texts that are represented in a computer by a number of different files. A Perspect file consists of the external part of a module, the internal part of a module, or the concatenation of one or more parts of modules.

⟨specification⟩ ::= ⟨file⟩*.
⟨file⟩ ::= ⟨external⟩ | ⟨internal⟩ | (⟨external⟩ | ⟨internal⟩)+.

When a file only contains the external part of a module, the name of that file is ⟨mod-

ule ident⟩ followed by the suffix '`.ext`' (which consists of lower case letters). When a file contains the internal part of a module, the name of the file is ⟨module ident⟩ followed by '`.int`'. When a file contains several parts of a module, the name of the file should end in '`.per`'.

The first two kinds of files are intended to enable a Perspect implementation to easily locate all relevant files on import. The last kind of file makes it possible to represent a Perspect specification (or some part of it) as one big file.

The context free syntax of Perspect is given by the following EBNF grammar. Note that *braces* '{' and '}' have a special status here: we use the notation {⟨notion⟩ ⟨separator⟩}* for zero or more ⟨notion⟩s *separated* by ⟨separator⟩s, and similarly {⟨notion⟩ ⟨separator⟩}+ for one or more ⟨notion⟩s *separated* by ⟨separator⟩s. This means that {⟨notion⟩ ⟨separator⟩}* differs from (⟨notion⟩ ⟨separator⟩)* in which the brackets are used for grouping. A set of syntax diagrams corresponding to this grammar appears as appendix I of this thesis.

⟨external⟩ ::=
    '**`external`**' ⟨module ident⟩ (⟨import⟩ | ⟨declaration⟩)*.
⟨internal⟩ ::=
    '**`internal`**' ⟨module ident⟩ (⟨import⟩ | ⟨declaration⟩ | ⟨equation⟩)*.
⟨import⟩ ::= '**`import`**' {⟨module ident⟩ '`,`'}+.
⟨declaration⟩ ::=
    '**`sort`**' {⟨sort ident⟩ '`,`'}+ |
    '**`function`**' ⟨function⟩+ |
    '**`variable`**' ⟨variable⟩+.
⟨function⟩ ::= {⟨function type⟩ '`,`'}+ '`:`' ⟨sort ident⟩.
⟨function type⟩ ::=
    ['**`rec`**'] ['**`echo`**'] ⟨function ident⟩ ['`(`' {(['`*`'] ⟨sort ident⟩) '`,`'}+ '`)`'].
⟨variable⟩ ::= {⟨variable ident⟩ '`,`'}+ '`:`' ⟨sort ident⟩.
⟨equation⟩ ::= '**`equation`**' ({⟨term⟩ '`,`'}+ '`:`' ⟨term⟩)+.
⟨term⟩ ::=
    ⟨sort ident⟩ | ⟨function ident⟩ ['`(`' {⟨term⟩ '`,`'}+ '`)`'] | ⟨variable ident⟩.

(Most syntactic constructions in Perspect will be self-explanatory for someone who already knows an algebraic specification formalism, except for the use of sort identifiers for unnamed variables (see section 3.1.4.4), the keywords **`echo`** and **`rec`** (see sections 3.1.3.2 and 3.1.5.2) and the use of the asterisk character '`*`' (see section 3.1.5.2).)

Note that, when a function identifier is preceded both by the **`rec`** and **`echo`** keywords, the **`rec`** keyword should come first.

Also, note that it is possible to write more than one function declaration after the keyword **`function`**. However, one is not allowed to write **`functions`** (as in ASF). The keyword **`function`** should be interpreted as 'function section'.

3.1.2. Elimination of abbreviations

A number of Perspect constructions are used as an abbreviation. Typing, semantics and restrictions of Perspect specifications will only be described for a kernel formalism, from which these abbreviations have been eliminated. This elimination procedure is called simplification.

Simplification eliminates the various lists in the Perspect specification, after which each imported module has its own import statement, each object has its own declaration (including keyword **`sort`**, **`function`** or **`variable`**), and every rewrite rule has its own equation (again, including the keyword **`equation`**).

3.1.2.1. The kernel formalism. The kernel formalism of which the simplified Perspect specification will be an instance, is a subset of Perspect and is given by the following syntax:

⟨external⟩ ::=
    '**`external`**' ⟨module ident⟩ (⟨import⟩ | ⟨declaration⟩)*.
⟨internal⟩ ::=
    '**`internal`**' ⟨module ident⟩ (⟨import⟩ | ⟨declaration⟩ | ⟨equation⟩)*.
⟨import⟩ ::= '**`import`**' ⟨module ident⟩.
⟨declaration⟩ ::=
    '**`sort`**' ⟨sort ident⟩ |
    '**`function`**' ⟨function⟩ |
    '**`variable`**' ⟨variable⟩.
⟨function⟩ ::= ⟨function type⟩ '`:`' ⟨sort ident⟩.
⟨function type⟩ ::=
    ['**`rec`**'] ['**`echo`**'] ⟨function ident⟩ ['`(`' {(['`*`'] ⟨sort ident⟩) '`,`'}+ '`)`'].
⟨variable⟩ ::= ⟨variable ident⟩ '`:`' ⟨sort ident⟩.
⟨equation⟩ ::= '**`equation`**' ⟨term⟩ '`:`' ⟨term⟩.
⟨term⟩ ::=
    ⟨sort ident⟩ | ⟨function ident⟩ ['`(`' {⟨term⟩ '`,`'}+ '`)`'] | ⟨variable ident⟩.

Note that the nonterminals in this context free grammar have the same names as those in the full grammar (given in the previous section). However, this is a different grammar, and therefore here a different interpretation is given to those nonterminals. Yet, all instances of a nonterminal in the grammar of the kernel formalism will always be an instance of the nonterminal with the same name in the grammar of the full formalism.

3.1.2.2. List elimination. Simplification of a Perspect specification eliminates eight kinds of lists from the specification. These kinds of lists correspond to the plus signs in the following fragment of the Perspect grammar:

⟨import⟩ ::= '**`import`**' {⟨module ident⟩ '`,`'}+.
⟨declaration⟩ ::=
    '**`sort`**' {⟨sort ident⟩ '`,`'}+ |

‘**function**’ ⟨function⟩$^+$ |
‘**variable**’ ⟨variable⟩$^+$.
⟨function⟩ ::= {⟨function type⟩ ‘,’}$^+$ ‘:’ ⟨sort ident⟩.
⟨variable⟩ ::= {⟨variable ident⟩ ‘,’}$^+$ ‘:’ ⟨sort ident⟩.
⟨equation⟩ ::= ‘**equation**’ ({⟨term⟩ ‘,’}$^+$ ‘:’ ⟨term⟩)$^+$.

Each list is eliminated by replacing the import, declaration or equation by as many imports, declarations or equations as there are elements in the list. The order in which these lists are eliminated from the specification is irrelevant for the result of the simplification.

3.1.2.3. Example. Consider the following simple Perspect specification that defines the Booleans (For a description of the use of the sort identifier `BOOL` as an unnamed variable, see section 3.1.4.4.):

```
external Booleans
 sort BOOL
 variable bool: BOOL
 function
  true, false, not(BOOL), or(BOOL,BOOL), and(BOOL,BOOL): BOOL
internal Booleans
 equation
  not(false), or(true,BOOL), or(BOOL,true), and(true,true):
   true
  not(true), or(false,false), and(false,BOOL), and(BOOL,false):
   false
  or(false,bool), or(bool,false), and(true,bool), and(bool,true):
   bool
```

After simplification, this specification becomes:

```
external Booleans
 sort BOOL
 variable bool: BOOL
 function true: BOOL
 function false: BOOL
 function not(BOOL): BOOL
 function or(BOOL,BOOL): BOOL
 function and(BOOL,BOOL): BOOL
internal Booleans
 equation not(false): true
 equation or(true,BOOL): true
 equation or(BOOL,true): true
 equation and(true,true): true
 equation not(true): false
```

```
equation or(false,false): false
equation and(false,BOOL): false
equation and(BOOL,false): false
equation or(false,bool): bool
equation or(bool,false): bool
equation and(true,bool): bool
equation and(bool,true): bool
```

Note that the Perspect kernel formalism syntactically resembles the RAP formalism (see for example [Hussmann, 1987]).

3.1.3. Declarations and typing

In the rest of the Perspect language definition, only specifications that are written using the kernel formalism will be considered. All other Perspect specifications have to be seen as abbreviations of specifications in the kernel formalism.

3.1.3.1. Modules. A Perspect specification consists of a number of files, that together describe a number of modules. A module is identified by a module identifier that must be unique for the module in the specification (so a module identifier should not be used for more than one module; however, it *may* also be used as a sort, function or variable identifier). Therefore a module identifier occurs exactly once in the specification after the keyword **`external`**, and once after the keyword **`internal`**.

It must be possible to order the files of a Perspect specification in such a way that the text that one obtains by concatenating the files consists of a succession of all modules in the specification. Here, a module consists of its external part directly followed by its internal part. (When the internal part of a module directly follows the external part of a module in a file – the name of the file should in this case end in '`.per`', as described in section 3.1.1.2 – then it follows that these parts have to be the two parts of the same module.)

The ordering on the modules of a specification that is chosen has to satisfy another restriction. It also is used to disallow cyclical imports in the specification. See for this extra restriction the end of section 3.1.3.3.

3.1.3.2. Echo declarations. In a Perspect specification, a function declaration may contain the keyword **`echo`**. In the language definition these declarations are not considered to be real declarations. The only use of these echo declarations is to give the place of a function in a preordering that will be defined in 3.1.5.2. For the origin rule (3.1.3.4), typing (3.1.3.5) and semantics (3.1.4), these echo declarations should be ignored.

Echo declarations should satisfy the following requirements:

(i) Echo declarations are only allowed in the internal part of a module.

(ii) Each echo declaration should correspond to a declaration in the external part of the same module, so the external part should contain a function declaration with

the same function name, the same domain, and the same range. This external counterpart to an echo declaration is not allowed to contain the keyword **rec**.

(iii) Each declaration in the external part of a module corresponds to at most one echo declaration in the internal part of the same module.

3.1.3.3. Visibility. In a Perspect text there are three kinds of objects: sorts, functions and variables. These objects are declared (in the kernel formalism) by means of a declaration that starts with one of the keywords **sort**, **function** or **variable**. A declaration introduces an identifier for the declared object. Furthermore, the declaration indicates the domain and range for functions, and the type for variables.

In a specification, an object is not identified by an identifier (possibly with extra typing information), but by the place in the text where the object is declared. This is necessary to distinguish between objects with the same name without having to rename objects, an operation that is not present in Perspect. A Perspect specification has always the property however, that at a specific place in the specification text an object is always uniquely determined by its identifier (compare sections 3.1.3.4 and 3.1.3.5).

Not all declarations are visible at all places in the text; the visibility of declarations is determined by the module structure and by the import statement. This visibility is defined in the following way:

At the start of a module, no declaration is visible. After a sort, function or variable declaration, that declaration is visible in the rest of the module (so a declaration in the external part of a module is also visible in the internal part). After an import statement all declarations that are visible at the end of the external part of the imported module (which is the module whose identifier occurs in the import statement) are also visible in the rest of the (importing) module (again an import statement in the external part of a module is also effective in the internal part of that module).

The following restriction should hold for the identifier that occurs in an import statement: it should be the module identifier of a module that occurs in the Perspect specification. Furthermore, this module should occur earlier in the specification (using the ordering on the Perspect files chosen in section 3.1.3.1) than the module in which the import statement occurs. This implies that Perspect forbids cyclical imports.

3.1.3.4. The origin rule. There exists a restriction on the use of declarations, that is called the *origin rule*. Perspect specifications should satisfy the origin rule, which can be formulated in the following way:

> *When in two different declarations (of sorts, functions or variables) the two declared objects are named with the same identifier, then these two declarations should not both be visible at the same place anywhere in the specification text.*

An object can be visible at some point in the text only once. If an object is imported in a module through more than one route, it still stays *one* object.

For a more complicated version of the origin rule in the context of ASF, see section 1.1.7 in [Bergstra, Heering, Klint, 1989].

3.1.3.5. Typing. When an identifier is used, the declaration that introduces that identifier should be visible at the place it is used. There are two kinds of identifier that refer to an object that was defined earlier:

(i) The sort identifiers that occur in a function or variable declaration.
(ii) The sort, function or variable identifiers in terms.

These identifiers should have been introduced in a corresponding declaration (so, a sort declaration for a sort identifier, a function declaration for a function identifier and a variable declaration for a variable identifier) that is visible where the identifier is used. By the origin rule, at most one such declaration is visible. This declaration, then, is called *the* declaration of the object that is referred to by the identifier.

According to the declarations of the identifiers that occur in the terms in the specification, those terms should be well-typed in an inside-out fashion. The definition of well-typedness is recursive:

(i) A sort or variable identifier is always well-typed. A sort identifier is used as an unnamed variable, and has as type the sort the identifier refers to. A variable identifier is used as a named variable. This variable has as type the sort that is given in the declaration of the variable.
(ii) A function identifier that is not followed by arguments is used as a constant. Such a term is well-typed if and only if no domain is present in the declaration of this function.
(iii) A function identifier that is followed by one or more arguments is well-typed when the arguments are well-typed, and when in the declaration of the function the domain is given as the sequence of sorts that are the types of the arguments, in the same order.

Finally, a Perspect specification should satisfy the restriction that for all equations in the specification, the left hand and right hand sides of the equation should be well-typed, and have the same type.

3.1.4. Semantics

In the rest of the definition of Perspect only specifications that are syntactically correct (section 3.1.1), lie within the kernel formalism (section 3.1.2) and that satisfy the origin rule and are well-typed (section 3.1.3) will be considered. Only for those specifications a semantics will be defined.

A module in a Perspect specification has two different types of semantics: a semantics in the form of a term rewriting system, and a semantics in the form of an algebra. When the extra requirements that will be stated in section 3.1.5 are not satisfied, the two kinds of semantics do not necessarily correspond to each other. However, a Perspect specification is only correct when those requirements are satisfied. In that case the two kinds of semantics correspond.

A Perspect specification is a concrete object: a text in a computer memory or on paper. However, the meaning of a module from a Perspect specification is an abstract object: a term rewriting system or algebra. The definitions of the various notions that are used in the definition of these meanings are given in chapter 2.

With a Perspect module a number of signatures will be associated. The sorts and functions in such a signature correspond each with a declaration in the specification text. This means that a sort or function can uniquely be determined by pointing at a certain declaration in the text of the specification. It is well possible that in different modules the same identifier is used for different sorts (or functions). However, ambiguity is excluded by the origin rule.

Two signatures are associated with a module in a Perspect specification: its external and its total signature.

3.1.4.1. External signature. The external signature of a module consists of those sorts and functions whose declarations are visible at the end of the external part of the module (these declarations can occur within the external part itself, or can have been made visible by an import statement).

3.1.4.2. Relevance. To define the total signature of a module, the relation of relevance between modules must be defined. A module is relevant for some other module, if it has been imported by that other module (either in the external, or in the internal part), or if it is relevant to a relevant module. Furthermore, a module is relevant to itself. So relevance is the reflexive and transitive closure of the import relation, in which the distinction external versus internal is ignored.

3.1.4.3. Total signature. The total signature of a module consists of all sorts and functions that are visible at the end of the internal part of the modules that are relevant to that module.

The total signature of a module does *not* only consist of those sorts and functions whose declarations are visible at the end of the internal part of that module (the hidden objects from the imported modules would be missing). The concept of *internal signature* could be defined in this way, but the notion of the internal signature of a module has no further application.

It is possible that the total signature contains objects that are referred to by the same identifier, and it is even possible for those objects to have the same kind. We apply here the statement that an object is not identified with its name, but with its declaration in the specification text (conform the remark in section 3.1.3.3).

3.1.4.4. Total rewriting system. With a Perspect module a term rewriting system is associated that is called the total rewriting system of that module. The total rewriting system of a module consists of all equations (instances of the syntactic notion ⟨equation⟩) in all modules that are relevant to the module. The '**:**' in the equations is to be interpreted as a rewriting arrow '→', and the equations are to be interpreted as rewriting rules.

The terms that are a sort identifier must be seen as unnamed variables over the given sort. When in a rewriting rule a sort identifier is used more than once, these identifiers must be interpreted as different unnamed variables. (The way Perspect has unnamed variables is a typed variation on the way Prolog uses the '_' symbol (see, for example, [Clocksin, Mellish, 1981]).)

3.1.4.5. Semantics. A Perspect module has two different kinds of semantics: one as a term rewriting system, and one as an algebra. The meaning of a Perspect module as a term rewriting system is the total rewriting system of the module as described in section 3.1.4.4. The meaning of a Perspect module as an algebra is the initial algebra of the total rewriting system considered as a set of equations (that is, after forgetting the direction of the rewriting arrows), restricted to the external signature of the module.

3.1.5. Restrictions on the specification

A Perspect specification, in order to be correct, must satisfy a number of requirements:

(i) The text of the specification must be syntactically correct according to the grammar in section 3.1.1.

(ii) The specification should satisfy the origin rule everywhere, and be well-typed according to the description in section 3.1.3.

(iii) A number of requirements must be satisfied by the rewriting systems that are associated with the different modules, as defined in section 3.1.4. These requirements on the rewriting systems are syntactic in the sense that they are (taken together) decidable. In the rest of this section these 'semantic' requirements imposed on a Perspect specification will be enumerated.

So, suppose we have a Perspect specification expressed in the kernel formalism, syntactically correct, satisfying the origin rule and correctly typed, and consider some module in the specification, with as its meaning the total rewriting system associated with it.

3.1.5.1. Variables. There are a number of restrictions on the use of variables in Perspect rewriting rules:

(i) Each variable should only occur once in the left hand side of a rewriting rule. This is the requirement of *left linearity* of the term rewriting system.

(ii) All variables that occur in the right hand side of a rewriting rule should also occur in the left hand side of the same rewriting rule, and all named variables that occur in the left hand side of the rewriting rule should also occur in the right hand side of the same rewriting rule. When a variable is needed in the left hand side of a rewriting rule that does not occur in the right hand side of that rule, an unnamed variable should be used, i.e., the sort identifier of the type of that variable.

(iii) Unnamed variables in a term, i.e., variable given as a sort identifier, should only occur in the left hand side of a rewriting rule.

3.1.5.2. Monotone termination under a path ordering. Given the text of a Perspect specification, it is possible to derive from this text a partial preordering on all functions in the total signature of a module in the specification:

(i) When a module is relevant for another module (see 3.1.4.2), then all functions of the first module are smaller in the preordering than those from the second one.

(ii) Functions that are declared within different modules which are not relevant for each other are not related by the preordering.

(iii) Functions that are declared within one module, are ordered by the textual ordering of the declarations. However, the order of the external and internal part of the module has been interchanged, i.e., although the textual ordering determines the preordering within the external and internal parts, all functions in the internal part come before the external part. Also, the next two clauses overrule this clause for the case of declarations containing the keyword **`echo`** or **`rec`**.

(iv) For functions that also have a declaration containing the keyword **`echo`** (see also 3.1.3.2), the place of the echo declaration determines the place in the preordering, instead of the real declaration. This makes it possible to have an arbitrary ordering between the functions in a module, whatever the status as external or internal functions may be.

(v) If a function declaration contains the keyword **`rec`**, the function is equivalent to the previous function in the preordering that is defined here, instead of 'next greater'. This makes it possible to have different functions in the specification that are equivalent with regard to the preordering which can be useful when two functions are defined recursively in terms of each other. Obviously, the first function declaration of the internal part of a module is not allowed to contain the **`rec`** keyword. Likewise, the first function declaration of the external part of a module should not contain the **`rec`** keyword.

(There are some restrictions on the occurrences of the **`echo`**, **`rec`** and '`*`' markers that should be satisfied by the specification: if a function has an **`echo`** declaration, only that **`echo`** may participate in a **`rec`** relation with another function. Also, only that **`echo`** declaration may contain the '`*`' marker.)

Now, for each argument position of a function in the total signature of the module a Boolean called the *status* of that argument position is defined in the following way. A function argument position has *lexical* status if the sort identifier in the declaration of that function is preceded by an asterisk character '`*`'. If it is not preceded by an asterisk, it has *multiset* status.

The preordering on the function symbols and the status of the argument positions of the functions together define an associated *path ordering with argument status* as is described in section 2.2.2.

A module in a Perspect specification should satisfy the requirement that the right hand sides of all rewriting rules have to be smaller in this path ordering than the corresponding left hand sides. In other words, the rewriting rules should be monotone decreasing in this path ordering.

This requirement implies strong termination of the term rewriting system.

3.1.5.3. Open confluence. The total rewriting system associated with a module should be strongly open confluent as defined in section 2.3.1.1. This means that the rewriting system should also be confluent on terms that contain free variables. In the case of a terminating rewriting system this requirement can be verified using a weak form of the Knuth-Bendix algorithm.

This requirement of course implies confluence of the term rewriting system. The requirements from sections 3.1.5.2 and 3.1.5.3 together imply the completeness of the

term rewriting system.

3.1.5.4. Strong persistence. For each sort in the module, it is required that the set of closed normal forms (i.e., normal forms without variables) of that sort should be equal to the set of closed normal forms of the sort with respect to the total rewriting system of the module in which the sort is declared.

This requirement implies, given the requirements from 3.1.5.2 and 3.1.5.3, persistence of the specification. For the definition of persistence see section 2.5.1.

This last requirement transcends the modular structure of the specification, and says effectively that the semantics of the specification should respect the modular structure of the specification. It is not a requirement on the total rewriting system of one module of the specification, but on the relation between the total rewriting systems of two related modules.

3.1.6. Five levels of Perspect

It is hard to write a correct Perspect specification. It might be nice to have a way to say that a specification is halfway between an ordinary algebraic specification and a Perspect specification. Therefore, we will describe in this section five *dialects* of Perspect that are less restrictive than the full language as defined above. These dialects are called Perspect Level 1, Perspect Level 2, Perspect Level 3, Perspect Level 4 and Perspect Level 5. This sequence is monotone: a text that is in a certain dialect is also in all earlier dialects (at least if it does not contain the identifiers **`if`** and **`then`**, which are keywords in Perspect Levels 1, 2 and 3).

We will define the dialects in reverse order, because each dialect is obtained from the next by weakening the restrictions that are imposed on the specification texts. The following diagram shows some of the properties of the five dialects that we will define:

Perspect Level	1	2	3	4	5
		'ASF'			Full Perspect
Decidable	+	+	−	−	+
Persistent	−	−	+	+	+
Executable	−	−	−	+	+

Decidable means here that it is decidable whether a text is correct according to the definition of the language, *not* that it is decidable whether two terms are equal according to a specification in the language. This second property has the same signature as the row labeled *executable*.

Perspect Level 5 or *Full Perspect* is the Perspect language as defined in sections

3.1.1 until 3.1.5. This means level 5 is the highest level there is, because a Perspect Level 5 specification meets all Perspect's requirements.

Perspect Level 4 is the dialect in which all restrictions from section 3.1.5 that were slightly too heavy are replaced by their more 'natural' equivalents. This means that Perspect Level 4 is not a decidable language (these restrictions were heavier than was desirable in order to make the language decidable). We will now list the restrictions that a Perspect Level 4 specification has to satisfy instead of the restrictions from section 3.1.5:

- The requirements of section 3.1.5.1 should still be satisfied. In particular the term rewriting system of the specification should still be left-linear.
- The specification should be strongly closed terminating as defined in section 2.2.1.1 (it need not be strongly open terminating). The keywords **`echo`** and **`rec`**, and the marker '`*`' are still allowed in the specification (as well as in all dialects that we still have to define), and if they occur they should satisfy the requirements in sections 3.1.1 up to 3.1.5. However, the path ordering with argument status that is implicit in the specification is now irrelevant. So, the equations need not be monotone in the path ordering anymore.
- The specification should be strongly closed confluent as defined in section 2.3.1.1 (it need not be strongly open confluent as in Full Perspect).
- The requirements of section 3.1.5.4 should still be satisfied: not only is persistence required, but also the stronger form that says that normal forms should stay normal forms.

The motivation for retaining the requirement of left-linearity in the definition of Perspect Level 4 is that it makes it possible to execute a specification in a more efficient way. Perspect Level 4 is the language for writing *executable* specifications. This also motivates the emphasis on the normal forms in the way persistence is required.

While Perspect Level 4 is the language that focuses on executability, Perspect Level 3 is a pure *specification* language. This means that all constraints from section 3.1.5 are removed, apart from the requirement of persistence. This means that all texts that satisfy the requirements in sections 3.1.1 to 3.1.4 are a correct Perspect Level 3 specification provided that:

- If the keywords **`echo`** and **`rec`**, and the marker '`*`' are present they still should behave as in Full Perspect.
- Conditional equations are added to the language. The definition of ⟨equation⟩:

  ⟨equation⟩ ::= '**`equation`**' ({⟨term⟩ '`,`'}$^+$ '`:`' ⟨term⟩)$^+$.

  in the grammar in section 3.1.1.2 is replaced by:

  ⟨equation⟩ ::=
  '**`equation`**' (['**`if`**' ⟨simple equation⟩$^+$ '**`then`**'] ⟨simple equation⟩)$^+$.
  ⟨simple equation⟩ ::= {⟨term⟩ '`,`'}$^+$ '`:`' ⟨term⟩.

  Clearly this makes it necessary to add the keywords **`if`** and **`then`** to the language. The interpretation of a conditional equation is straightforward.

- The meaning of a module as a term rewriting system does not make sense anymore, so we cannot require the specification to be strongly persistent. It still makes sense as an equational specification, however. We require the specification to be weakly persistent as defined in section 2.5.1.2.

There are no longer any restrictions on the use of variables in Perspect Level 3. This means for instance that unnamed variables can be used anywhere. Note however that the equation:

```
equation S: S
```

in which `S` is a sort identifier collapses sort `S` to one point, because the two unnamed variables `S` are considered to be different variables.

Perspect Level 3 is equivalent to ASF with Perspect syntax and the requirement of persistence (to deserve the name *Per*spect) added.

Perspect Level 2 is just Perspect Level 3 without the requirement of persistence. It is ASF in a Perspect syntax. One could write ASF-to-Perspect-Level-2 and Perspect-Level-2-to-ASF translators that leave the specified algebra invariant (at most they will have to rename some of the identifiers because of Perspect's lexical conventions).

While Perspect Level 2, 3, 4 and 5 specifications all have the property that a module in a specification has *one* algebra as its meaning, we lose this property in Perspect Level 1.

When specifying, one often works with *incomplete* specifications, in which detail still has to be added. Perspect Level 1 is an attempt to give these kind of *protospecifications* a formal status.

Let be given a correct Perspect Level 2 specification with an associated syntax tree. Now, omit any number of instances of the notions ⟨external⟩, ⟨internal⟩, ⟨import⟩, ⟨declaration⟩ and ⟨equation⟩. This will give a text (not necessarily correct according to the syntax of Perspect, as a number of internal parts of modules may be missing). This text is, by definition, a correct Perspect Level 1 specification, and the algebra that was the meaning of a module in the Level 2 specification is now *a* meaning of the corresponding module in the Level 1 specification. Clearly, a specification gets a large number of meanings in this way. For example, the empty text:

has all semi computable algebras as a meaning for any module which has a name that can be named by a Perspect identifier.

This ends the description of the five dialects of Perspect. A possible application of these dialects might be the development cycle of a specification: one can try to go from a Level 1 specification, via a Level 2, 3 and 4 specification to a Full Perspect specification. Another application might be financial: a software house might require a specification of the program you order from it, and the price you have to pay for the program might depend on the language level on which you submit your specification.

3.2. Properties

Perspect was designed to have a number of desirable properties: decidability of the language, executability of the specifications, persistence of the specifications, a modest amount of expressiveness and compositionality of the definition. In the following sections we will show why these properties hold. We show how to apply the theory from chapter 2 to the language defined in the previous section.

While a number of languages have *some* of these properties, Perspect has *all* of these properties. We think that it is desirable that a specification language does not compromise on any of these aspects. In this sense Perspect is a powerful language.

3.2.1. Decidability

It is decidable whether a string of ASCII characters is a correct Perspect specification according to the language definition of section 3.1. This makes it possible to write a computer program, a *checker*, that checks whether a given string is a correct Perspect specification. Such a program will be described in section 3.4. We will show that Perspect is a decidable language by showing how a checker could be implemented.

The language definition consists of two parts. The first part consists of sections 3.1.1 to 3.1.4. In these sections we define an ordinary algebraic specification language and give it a meaning. The second part is section 3.1.5. Here we put a number of extra restrictions on the texts that are allowed in the language.

In order to check the first part of the language definition, one has to perform a syntax and type check. These checks are completely standard, and we will not describe how to perform them. The existence of these checks implies that it is decidable whether a specification satisfies the first four sections of the Perspect definition. So, let be given a text that satisfies the requirements from this first part. We have to show how to verify the requirements from 3.1.5. We will see that each of these requirements is decidable when the earlier requirements from this section have already been shown to be true.

First, the specification has to satisfy the restrictions on the usage of variables from section 3.1.5.1. These requirements are simple to verify by looking at each equation in the specification in turn. If these requirements are not satisfied we clearly do not have a Perspect specification. If they *are* satisfied we know that the term rewriting system associated with the specification is left linear.

Second, the specification has to satisfy the restriction that all equations are decreasing in a given path ordering with argument status: it should be monotone terminating. Algorithm 2.2.3.1 shows how to decide whether this is the case.

Third, the specification has to satisfy the restriction that the term rewriting system is strongly open confluent. Now, we may assume that we already have tested the specification for monotone termination. So, we may assume that we know that the term rewriting system associated with the specification is strongly open terminating. This means that algorithm 2.3.3.1 gives us a way to check strong open confluence.

Last, the term rewriting system should be strongly persistent. In the case that it is left linear, algorithm 2.5.4.2 shows us how to decide whether it is strongly persistent. But may assume we already know that it is left linear, so we can check strong persistence.

One can wonder if it is possible to replace one of the requirements of section 3.1.5 by a weaker restriction. For example, each of the requirements is somewhat 'too strong'. Instead of requiring strong termination we require monotone termination under some path ordering. Instead of requiring strong closed confluence (which would be enough for the persistence check) we require strong open confluence. And, instead of requiring just persistence we require strong persistence: we are not allowed to change the normal form that is associated with an object, even if the algebra is not modified by this change.

It turns out that these 'more semantic versions' of the requirements in 3.1.5 destroy the decidability of the language. This is shown by propositions 2.2.3.2, 2.3.3.2 and 2.5.4.5. So, the easy way to make the requirements in 3.1.5 weaker does not work.

3.2.2. Executability

In the previous section we saw that the term rewriting system associated with a Perspect specification is left linear, strongly terminating and strongly confluent. Also, the language Perspect does not allow conditions in equations. Together this means that for each Perspect specification we have a term rewriting system that is very easily executable (because no conditions or equality of arguments has to be verified) and this term rewriting system (according to propositions 2.3.1.6) corresponds directly to the initial algebra associated with the specification.

Executability is the property of Perspect that is the least desirable (why should a specification be executable, after all). However, as we indicated in section 1.1.5, executability is a property that is closely linked to persistence, and the easiest way to verify persistence is by just requiring executability as a term rewriting system first.

As shown in section 1.1.4, a specification can behave reasonably when executing it as a term rewriting system, while it is completely incorrect. Of course, this cannot happen with a Perspect specification, because we know (because of confluence) that the behavior of the term rewriting system corresponds to the behavior of the algebra. So, the example from section 1.1.4 cannot be translated to the Perspect language and both still be incorrect and behave plausibly when debugging. Even though this example *is* strongly persistent (because it consists of only one module).

3.2.3. Persistence

In section 3.1.5 one of the requirements that a Perspect specification has to satisfy is that it has to be strongly persistent. In section 3.2.1 we saw that a Perspect specification also is always strongly terminating and strongly confluent. From this it follows that a Perspect specification is always persistent.

The persistence check of a Perspect checker can be used to check the persistence of a specification when it is known that the specification is a left-linear, strongly terminating and strongly confluent term rewriting system, even though the Perspect system is not able to verify this. If the Perspect checker only complains about the termination or confluence of the specification, while the specification *is* strongly terminating and strongly confluent, the specification is strongly persistent.

3.2.4. Expressiveness

As we said before: The three properties from the last three sections, decidability, executability and persistence are not very hard to satisfy. In particular the *empty* language (i.e., the language that admits no texts at all) has those same three properties. Clearly, we want to show that the language is in some sense 'big enough'.

Not all semi-computable algebras are expressible in Perspect (while they are in ASF). This follows from the fact that in a Perspect specification it is decidable whether two closed terms represent the same object in the algebra. To decide this, one just has to evaluate the normal form of those terms. Because the term rewriting system associated with the algebra is confluent, the two terms represent the same object if and only if the two normal forms are equal. There are semi-computable algebras in which it is *not* decidable whether two closed terms are equal.

On the other hand, all primitive recursive algebras, as defined in section 2.6, are specifiable in Perspect. This follows from the example that we will give in section 3.3.3. In this specification we have two sorts `N` and `PRIM`, representing the natural numbers and the primitive recursive functions, plus a function `appl` that makes it possible to apply primitive recursive functions from `PRIM` to the elements of `N`. Using these sorts and this function as 'building blocks' it is clear that each primitive recursive algebra can be modelled in Perspect.

On the other hand, the same example from section 3.3.3 shows that it is possible to specify an algebra in Perspect that is not primitive recursive. The function `xdiag` from the last module of the specification satisfies a condition that will be used in proposition 2.6.2, to prove that the algebra specified by the last module of the specification is not primitive recursive.

3.2.5. Compositionality

According to [Janssen, 1983], the semantics of a language should be *compositional*. This property means that whenever an object is built from a number of parts and each part has a meaning of its own, the meaning of that object should be a function of the *meanings* of the parts, without reference to the parts themselves. Compositionality of the semantics of a language can be a heuristic for finding errors in the definition of the language: often, when a semantics is not compositional, it is not the intended semantics.

The semantics of an algebraic specification language is in general not compositional.

Consider for example the following ASF specification:

```
module M
 begin
  exports
   begin
    sorts S
    functions
     c: → S
   end
  variables
   v: → S
  equations
   [1] v = c
 end M

module M′
 begin
  exports
   begin
    functions
     c′: → S
   end
  imports M
 end M′
```

In this specification the equation `c′ = c` is true because of equation `1`. However, if this equation is removed from the specification, this equation is no longer true, while module `M` still has the same meaning (a one point algebra). The meaning of `M′` depends on the form of module `M`, and not only on the meaning of module `M`. This shows that the semantics of ASF is not compositional.

Now, one of the two specifications in this examples (the one without equation `[1]`) is not persistent. So, maybe, if it is known that the specification is persistent, we *do* have a compositional semantics, i.e., in that case it might be possible to derive the meaning of a module from the meanings of the modules it imports.

Surprisingly, this is not the case. If we have a specification formalism that supports *hiding* of some proper part of the signature of an imported module, even if the specification is persistent, we still cannot derive the meaning of a module from the meanings of its imported modules.

For example, take a module that specifies one sort `TrafficLight`, and two constructor functions `stop` and `go`. Consider a second module that imports this module, hides the two constructors, and introduces two new functions `red` and `green` that (in order to satisfy persistence) are made equal to `stop` and `go` using two equations. Also, consider a third module that performs the same actions but with two functions `top` and `bottom` instead of `red` and `green`. We finally consider a fourth module that imports both the second and the third module without any further modifications or

additions. The algebras that are the meaning of the second and third modules both have the sort `TrafficLight`. In order to give the fourth module a persistent meaning, somehow the pair `red` & `green` has to be equated to `top` & `bottom`. However, by symmetry it will be clear that it cannot be determined how the 'matching' between these two pairs of functions should be performed. This shows that in this case the semantics cannot be compositional, while the specification *is* perfectly persistent.

We see that the semantics of a language in which the specifications are not *a priori* persistent is not compositional. We also see that a language that admits hiding of part of an imported signature does not have a compositional semantics. However, Perspect does not have these properties: it allows only persistent specifications, and it does not allow partial hiding of an imported signature. It turns out to be the case that Perspect *has* a compositional semantics. We will now give a precise description of the sense in which this is true.

Consider a secret Perspect specification. Suppose that one has to find the meaning of the final module, in the form of the algebra associated with that module. One is given the complete text of that module, but only the *external* part of all other modules in the specification. Furthermore, one is given the algebras associated with all modules imported by the final module. Finally, one is told that the specification is a correct Perspect specification (this is not decidable from the other information). The statement that Perspect has a compositional semantics is taken to mean that this task can be executed, i.e., one can determine the algebra associated with the final module from this information.

The fact that this is true follows from the observation that when it is ambiguous how to put the various copies of a sort in the algebras that have been imported together, this leads to an automorphism of the algebra in which that sort originated. And this automorphism extends to an automorphism of the imported algebras that contain that sort. This means that it does not matter in what way the identification of that sort in the imported algebras is chosen, because it will lead to an equivalent result.

Note that the fact that Perspect has a compositional semantics does not lead to a 'plus' operation on algebras (in the sense of module algebra). Because of the origin rule, it is in general not possible to obtain two given algebras as the meaning of two modules in a Perspect specification that can then can be 'added' by importing them both in a third module. This is even true if those algebras have to be 'mutually compatible', i.e., when they have to be isomorphic when restricted to the intersection of their signatures. As the `TrafficLight` example shows, it is too much to hope for a 'natural' way to combine all pairs of mutually compatible algebras.

It is true that Perspect gives a 'partial plus'. For all pairs of algebras that can simultaneously be present in a correct Perspect specification, the result of the 'addition' of these algebras is unambiguously determined.

3.3. Examples

When considering a language like Perspect, one can ask oneself the question how difficult it is to write a correct specification. To give an indication of the amount of effort

that will be needed for this, we first describe some of the problems one encounters when writing Perspect specifications, followed by a number of example specifications.

When writing Perspect specifications, a number of difficulties occur. It is amazing that the least of these difficulties lies in the requirement that the specification should be persistent. If one wants a specification to be persistent, all one has to do is find the proper modularization of the problem at hand.

Writing a specification that is monotone terminating under a path ordering turns out to be one of the main problems. This requirement makes it hard to use most forms of induction in the specification. It is, for example, almost unavoidable to use a unary representation for the natural numbers in a specification, if induction over the natural numbers is present. The examples in section 3.3.1 (which give a representation for the natural numbers and the integers that is not unary), while interesting, are not useful as a foundation for a larger specification.

It is also hard to get an open confluent specification. When a specification is written without confluence in mind, there are usually a number of open terms that can be rewritten to two different open normal forms. To avoid these *critical pairs*, there exist two conflicting strategies.

The first strategy consists of adding equations to the specification. This process is called Knuth-Bendix completion. It turns out that this is generally *not* a good solution. There are three problems that can be caused by the addition of an equation:

(i) The equation that has to be added has no direction for which it will be decreasing in the path ordering implied by the specification, and the ordering cannot be modified in order to make the system monotone terminating again.
(ii) The added equation is not left linear.
(iii) The added equation leads to new critical pairs. In this case the same problems can be caused by the equations that must be added for these new critical pairs.

The second strategy for removing critical pairs from a specification consists of making some equations more specific than is really necessary when considering the algebra that is being specified. This method in its most extreme form consists of only admitting an *orthogonal* term rewriting system. If one makes an equation more specific than is necessary, one can *eliminate* some overlap between redex patterns, and in this way the number of critical pairs in the specification decreases. However, one must be aware that if one makes the equations in the specification too specific, one might miss some 'cases' in a specification that contains an enumeration of cases. The persistence check of Perspect is the perfect protection in this situation, so this is not a real problem.

It is our experience that the second approach for eliminating critical pairs should be preferred over the first one.

The rest of this chapter consists of a number of Perspect specifications. There is a large amount of *overlap* between these specifications, and with the other specifications that are scattered throughout this thesis. For example almost every specification present either contains the Booleans or the natural numbers, and most contain both. We have not integrated these specifications in one long specification (which would have been esthetically more pleasing). There is a great difference in approach and

style between some of these specifications, and it seemed to us that in this way we would give a better impression of the variety of ways in which Perspect can be used. Also, while the examples in section 3.3.1 (the binary natural numbers and the binary integers) are interesting in their own right, they cannot be naturally integrated with the more involved specifications in the later sections.

3.3.1. Natural numbers and integers

The first example shows a number of properties of Perspect. It specifies the natural numbers, using two sorts called `uN` and `bN` which represent the natural numbers in unary and in binary form. Both sorts are mapped to each other by the two bijections `bu` and `ub`.

The representation of the unary numbers `uN` is the usual one. There are two generators. The zero is called `u0` (the prefix `u` is used to distinguish `u0` from the binary zero `b0` which will be introduced below) and the successor function is called `S`. The number 6, which is unary 111111, becomes the sixfold successor of zero: `S(S(S(S(S(u0))))))`. Note that each 1 in the unary representation corresponds to an `S` in the associated term.

The representation of the binary numbers `bN` is less standard. There are three generators: the zero, the function that adds a 0 at the back of the binary representation of a number, and the function that adds a 1 at the back. These generators are called respectively `b0`, `2` ('the double': adding a 0 at the back is the same as multiplying by 2, so `2`(i) corresponds to 2·i) and `S2` ('the successor of the double', `S2`(i) corresponds to 2·i+1). The number 6, binary 110, is represented by the term `2(S2(S2(b0)))`. Note that each 0 in the binary representation corresponds to a `2` in the term, and each 1 to an `S2`, and that the order of the digits is reversed under this correspondence.

Each generator, both unary as well as binary, has a counterpart that operates on the other sort. These functions are necessary to define the bijections between these two types of natural numbers. The binary successor is called `bS`, the unary counterparts of `2` and `S2` are called `u2` and `uS2`.

The bijections that are defined using these auxiliary functions are called `bu` (the binary representation of a unary number) and `ub` (the unary representation of a binary number).

A specification formalism that has a more powerful notation than Perspect would be able to indicate, using overloading of function names and invisible functions, that some of the sorts and functions are in some sense 'the same'. The same effect is obtained by removing all `u`'s and `b`'s from the identifiers in this specification.

This specification has a number of properties that are typical for a well written Perspect specification.

First, the sorts are, with their generators, isolated in a separate module. In this way, one has the guarantee that in a series of equations that define a function by enumerating all 'cases', none of these cases will be forgotten, because that would cause an impersistence in the specification. This is the reason for separating module `NATU1` from

module NATU2, and similarly for separating module NATB1 from module NATB2.

Second, there is a separate module, REL, that states that the two bijections bu and ub are inverse to each other. In most other specification formalisms this module would be useless, because the equations in this module do not add to the meaning of the rest of the specification. The statement that each bijection is the inverse of the other one already was true in module BIJ. However, in Perspect the presence of module REL makes sense. The system will check whether the statements from the module were already true. Note that the last three equations in REL are necessary to prove the total term rewriting system of REL open confluent. If these equations would not be present, a term like:

```
ub(bu(S(u)))
```

would be reducible to:

```
ub(bS(bu(u)))
```

and to:

```
S(u)
```

without these terms having a common (open) normal form.

Third, there are a number of sorts, in this case uN and bN, that represent the 'same' sort. In a normal algebraic specification, one sort would have been sufficient here. We see here a consequence of the fact that Perspect considers the specification to be a term rewriting system. This leads to this kind of 'tricks'.

```
external NATU1   [unary natural numbers: "6" is S(S(S(S(S(u0)))))]
 sort uN
  variable u: uN
 function u0, S(uN): uN
internal NATU1
 [empty]

external NATB1   [binary natural numbers: "6" is 2(S2(S2(b0)))]
 sort bN
  variable b: bN
 function b0, 2(bN), S2(bN): bN
internal NATB1
 equation 2(b0): b0

external NATU2   [binary constructor functions on unary numbers]
 import NATU1
 function
  u2(uN), uS2(uN): uN
internal NATU2
```

```
 equation
  u2(u0): u0
  u2(S(u)): S(S(u2(u)))
  uS2(u): S(u2(u))

external NATB2  [unary constructor functions on binary numbers]
 import NATB1
 function
  bS(bN): bN
internal NATB2
 equation
  bS(b0): S2(b0)
  bS(2(b)): S2(b)
  bS(S2(b)): 2(bS(b))

external BIJ  [bijections between uN and bN]
 import NATU2, NATB2
 function
  bu(uN): bN  [converts unary to binary]
  ub(bN): uN  [converts binary to unary]
internal BIJ
 equation
  bu(u0): b0
  bu(S(u)): bS(bu(u))
  ub(b0): u0
  ub(2(b)): u2(ub(b))
  ub(S2(b)): uS2(ub(b))

external REL  [verify that bu and ub are inverses to each other]
 import BIJ
internal REL
 equation
  ub(bu(u)): u
  bu(ub(b)): b
 [in order to satisfy the confluence check]
  bu(u2(u)): 2(bu(u))
  bu(uS2(u)): S2(bu(u))
  ub(bS(b)): S(ub(b))
```

Next, we will give a specification of the integers, in which the representation of an integer is a term which has a size that is proportional to the logarithm of the absolute value of the integer being represented. This is realized by representing the integers in a binary way.

There are various ways to represent integers in a term rewriting system:

- The most common (but not most useful) representation generates the integers in a

unary fashion, with a successor and a predecessor function. In this representation it turns out to be surprisingly hard to test whether an integer is equal to zero.

- The most useful representation constructs the set of the integers by gluing together two copies of the natural numbers, a positive and a negative copy (see for an example the integers in section 3.3.2). There is some variation possible in the way the zero is treated.
- Another method is to describe the integers as a pair of natural numbers. The integer being represented is the difference of those two numbers. This method, as well as the previous one, has the esthetical disadvantage of needing another layer in the specification to describe the natural numbers. The representation of the natural numbers in this layer can of course be unary as well as binary.
- The method we will show here describes the integers in a binary way. We use an infinite row of binary zeroes and ones to represent an integer. We use the two complement representation: an infinite row of ones stands for -1. In the specification this will be denoted by the constant `11`. In the same way the constant `00` stands for an infinite row of zeroes, which represents 0. As in the previous specification, there is a function for adding a zero at the end of a number, and one to add a one at the end of a number. Here, these functions (which in the previous specification were called `2` and `S2`) are called `0` and `1`.

The specification of the integers that we give here does not have many applications. If one wants to be able to do induction in a monotone terminating way, it is almost unavoidable to have a unary representation. This specification is only an example.

This specification cannot be easily extended obeying Perspect's restrictions. It is hard to add multiplication. Also, it is difficult to add equations stating properties of the integers. In both cases, open confluence tends to get lost.

```
external BooleanGenerators
 sort BOOL
 variable
  bool: BOOL
 function
  true, false: BOOL
internal BooleanGenerators
 [empty]

external BooleanOperations
 import BooleanGenerators
 function or(BOOL,BOOL), and(BOOL,BOOL), not(BOOL): BOOL
internal BooleanOperations
 equation
  not(false), or(true,BOOL), or(BOOL,true), and(true,true): true
  not(true), or(false,false), and(false,BOOL), and(BOOL,false):
   false
  or(false,bool), or(bool,false), and(true,bool), and(bool,true):
   bool
```

```
[properties]
 variable x, y, z: BOOL
 equation
 [double negation law:]
  not(not(x)): x
 [de morgan:]
  not(or(x,y)): and(not(x),not(y))
  not(and(x,y)): or(not(x),not(y))
[[distributivity:]
  and(x,or(y,z)): or(and(x,y),and(x,z))
  and(or(y,z),x): or(and(y,x),and(z,x))]

external Booleans
 import BooleanGenerators, BooleanOperations
internal Booleans
 [empty]

external IntegerGenerators
 sort INT
 variable
  i, j, k, l, m, n: INT
 [generators: 00 is 0, 11 is -1, 0(i) is 2 . i, 1(i) is 2 . i+1]
 function
  00, 11, 0(INT), 1(INT): INT
internal IntegerGenerators
 equation
  0(00): 00
  1(11): 11

external IntegerOperations
 import IntegerGenerators
 function
  inc(INT), dec(INT), add(INT,INT), min(INT), sub(INT,INT): INT
internal IntegerOperations
 equation
  inc(00): 1(00)
  inc(11): 00
  inc(0(i)): 1(i)
  inc(1(i)): 0(inc(i))
  dec(00): 11
  dec(11): 0(11)
  dec(0(i)): 1(dec(i))
  dec(1(i)): 0(i)
  add(00,i), add(i,00): i
  add(11,i), add(i,11): dec(i)
```

```
    add(0(i),0(j)): 0(add(i,j))
    add(0(i),1(j)), add(1(i),0(j)): 1(add(i,j))
    add(1(i),1(j)): 0(inc(add(i,j)))
    min(00): 00
    min(11): 1(00)
    min(0(i)): 0(min(i))
    min(1(i)): 1(dec(min(i)))
    sub(i,j): add(i,min(j))
   [properties]
    inc(dec(i)), dec(inc(i)): i
    min(inc(i)): dec(min(i))
    min(dec(i)): inc(min(i))
    min(min(i)): i

  external IntegerPredicates
   import Booleans, IntegerOperations
   function
    zero(INT), neg(INT), pos(INT),
    eq(INT,INT), ne(INT,INT),
    gt(INT,INT), ge(INT,INT), lt(INT,INT), le(INT,INT):
     BOOL
  internal IntegerPredicates
   equation
   [signs]
    zero(00): true
    zero(11), zero(1(INT)): false
    zero(0(i)): zero(i)
    neg(00): false
    neg(11): true
    neg(0(i)), neg(1(i)): neg(i)
    pos(i): neg(min(i))
   [comparison]
    eq(i,00), eq(00,i): zero(i)
    eq(i,11), eq(11,i): zero(inc(i))
    eq(0(INT),1(INT)), eq(1(INT),0(INT)): false
    eq(0(i),0(j)), eq(1(i),1(j)): eq(i,j)
    ne(i,j): not(eq(i,j))
    gt(i,j): pos(sub(i,j))
    ge(i,j): not(gt(j,i))
    lt(i,j): gt(j,i)
    le(i,j): not(gt(i,j))
   [properties]
    gt(i,00), lt(00,i): pos(i)
    lt(i,00), gt(00,i): neg(i)

  external Integers
```

```
 import IntegerGenerators, IntegerOperations, IntegerPredicates
internal Integers
 [empty]
```

3.3.2. Rational numbers

The specification in this section has been written in response to a specification by Piet Rodenburg [Hoekzema, Rodenburg, 1987]. It is a specification of 'numbers'. It is an attempt to describe *one* sort containing numbers and a number of functions on it, that are as generally applicable as possible.

When specifying algebraically, one sometimes feels the need for the real numbers. Obviously, there is no specification that has the *real* real numbers as its initial algebra. This is immediately clear from the consideration that an initial algebra is the quotient of a term algebra, so is countable, while the real numbers are uncountable.

One can try to approximate the real numbers in a number of ways. First, one can try to describe the 'normal' floating-point numbers, for instance the floating-point numbers as defined by IEEE standard 754 [IEEE, 1985]. While this is possible – there are only a finite number of IEEE numbers after all – this is a highly arbitrary algebra. It would be analogous to specifying the integers by specifying the integers modulo some high power of two. The second possibility is to specify a sort containing some kind of constructive reals, like the implementation of the reals as described in [Böhm, Cartwright, O'Donnell, Riggle, 1986] and [Böhm, 1987]. This would be interesting, but would result in a rather complicated specification. The third option, the one chosen by [Hoekzema, Rodenburg, 1987], and the one that we also will follow, is just to specify the rational numbers.

Now, there are a number of problems when specifying the rational numbers. First, there is the problem of division by zero. Second, one would like two rational numbers with the same numerical value to have the same normal form. Both problems are addressed by this specification.

The fact that division by zero is undefined is often a real problem. For example the Macintosh computer will give a system error with ID = 4 when encountering such a division (leaving the user no other option than rebooting the computer). In an algebraic specification in the Perspect formalism, one has only total functions at ones disposal. In order to define division as a total function, one has to resort to one of two tricks. In this specification both tricks are present simultaneously.

The solution that is used by the IEEE floating-point numbers is to extend the set of fractions with two values: 1/0 which is called *infinity*, and 0/0 which is called '*not a number*' (or *NaN*, for short). These two values behave like a kind of *error values*.

In fact, in IEEE standard 754 there are a whole variety of infinities and NaNs, but that is not relevant to this specification, which will only define one infinity called `inf`, and one NaN called `nan`. Also, the IEEE infinities are *affine* infinities which means that they are signed, in contrast to the infinity from this specification which is unsigned and closely resembles the *projective* infinity from draft 8.0 for standard 754 in [IEEE, 1981].

Alternatively, one can require the second argument of the division function to come from a sort that only contains rational numbers that differ from zero. One gets a specification that has a variety of sorts, which can be seen as subsort of each other. (In a specification formalism that supports order-sorted algebras (see, e.g., [Goguen, Jouannaud, Meseguer, 1985]), these subsort relations could have been handled by the formalism. Here, we have to specify embedding functions between these sorts explicitly if we want to indicate the subsort relations.)

This specification has a lot of sorts that all contain numbers. These sorts are:

`N`	the natural numbers
`Zp`	the positive integers
`Z`	all integers
`Qp`	the positive rational numbers
`Qf`	the rational numbers from the interval (0,1)
`Q`	all rational numbers
`Qe`	the rational numbers extended with the 'numbers' `inf` and `nan`

A number of functions have a suffix to indicate the sort they are associated with. These suffixes are `n`, `zp`, `z`, `qp`, `qf`, `q` and the empty string respectively. So, addition on the sort `Qe` (the most general sort that contains numbers) is called `add`, while addition on the sort `N` is called `addn`.

The problem of division by zero is solved in both ways in this specification. On the one hand, the function:

```
divq(Q,Qp): Q
```

avoids the problem by only allowing positive rational numbers for a divisor. On the other hand, the function:

```
div(Qe,Qe): Qe
```

has no problem with division by zero. For example the fraction 1/0 which can be expressed in the specification by the term:

```
div(num(whole(plusz(nat(succ(0n))))),num(whole(0z)))
```

turns out to have normal form `inf`.

The other problem with a specification of the rational numbers was that the simplest representation for the rational numbers, in which a number is represented as a pair of two integers, does not have the property that two terms representing the same number have necessarily the same normal form. This means that the simplest specification of the rational numbers has been ruled out if we require that the specification should have this property.

In [Hoekzema, Rodenburg, 1987] this problem is solved by representing numbers as a list of exponents in its prime factorization. Clearly, this is a representation in

which it is easy to define multiplication. However, in order to define addition, one has to recover the numerator and denominator of a fraction. Therefore, the specification has to contain a specification of an enumeration of the prime numbers.

In the specification from this section, a different representation for the rational numbers has been chosen. Here, a rational number is given as a *continued fraction*.

The two sorts that are used to represent a continued fraction are the sorts `Qp` and `Qf`, which are defined in terms of each other. The sort called `Qp` contains the positive rational numbers, and the sort `Qf` the rational numbers between 0 and 1. The two sorts are generated by the functions `wholep`, `brokenp` and `invadd1`. Expressions of the form `wholep(Zp)` and `brokenp(N,Qf)` represent an element of `Qp`. In the first case, we have an element of `Zp` that has been embedded in `Qp`. In the second case we have a representation of the sum of its first argument (which is the integral part of the number) and its second argument (which is the fractional part). An expression of the form `invadd1(Qp)` is an element of `Qf`. The expression `invadd1(x)` represents the value 1/(1+x). If x is a positive number, 1/(1+x) will be a number between 0 and 1. Together, the three generators `wholep`, `brokenp` and `invadd1` give a unique continued fraction representation for each positive rational number.

For example, consider the fraction 355/113 (the well known approximation of the number π) as an element of `Qp`. This fraction is expressed in the form of a continued fraction as 3+1/(7+1/16), which in turn can be written in the form 3+1/(1+6+1/(1+15)). It is represented in this specification by:

```
brokenp(
 nat(succ(nat(succ(nat(succ(0n))))))),
invadd1(brokenp(
 nat(succ(nat(succ(nat(succ(nat(succ(nat(succ(nat(succ(
 0n))))))))))))),
invadd1(wholep(
 nat(succ(nat(succ(nat(succ(nat(succ(nat(succ(nat(succ(
 nat(succ(nat(succ(nat(succ(nat(succ(nat(succ(nat(succ(
 nat(succ(nat(succ(nat(succ(0n)))))))))))))))))))))))))))))))
)))))
```

which should be read as:

```
brokenp('3',invadd1(brokenp('6',invadd1(wholep('15')))))
```

While these continued fraction representations are nice and simple, they are clearly not well suited for performing arithmetic. Therefore, the specification contains functions `nume`, `deno` and `fraction` to convert them to and from a representation as 'normal' fractions.

In order to be able to specify the function `fraction` that converts a normal fraction to a continued fraction, we will have to define the division and remainder functions on positive integers. Now, these functions (that might have been called `div` and `mod`) do not occur separately in the specification. Instead, they are only present in a 'bundled' way, together in the `divmod` function. The `divmod` function takes two posi-

tive integers and returns a bundle of three integers in an object of the sort `QR`. This bundle contains both the quotient and the remainder of the division together. It also contains, for convenience in the definition, the co-remainder which is the difference of the divisor and the remainder. Together, these three numbers are 'bundled' by the generator function `qr` of the sort `QR`. For an example, suppose that we evaluate:

```
divmod('42','5').
```

This expression will normalize to the normal form:

```
qr('8','2','3')
```

There is a slight complication with the definition of sort `QR`. The first argument of `qr` *might* be zero. However, if we know that the division is even, i.e., that the second argument is equal to zero, the first argument can *not* be zero, because we are dividing two *positive* integers. Now, it turns out that we will need this knowledge in order to be able to apply the `divmod` function. Therefore, the `qr` function has typing `qr(N,Zp,Zp)`, and for the case in which the second argument of the `qr` function would have been zero we have a second generator of sort `QR`, called `q0` with typing `q0(Zp,Zp)`. So, if we ignore the type errors, the expression `q0(i,j)` is just another way of saying `qr(i,'0',j)`.

There is one other interesting problem left in this specification. The function `fraction` (or rather the auxiliary function `qrtoqp` that is used by it) recursively converts a fraction to a continued fraction by repeatedly applying `divmod`. Now *we* know that this process terminates after finitely many steps. However, *Perspect* also has to be convinced of this fact. And a path ordering does not have the power to show the termination of this process.

The way out that we take is an ugly one; its only merit is that it works. We add a parameter to each function in the recursion that *counts* the number of steps that the recursion has taken. After this counter reaches zero (it is counting backwards), the recursion is broken off with some default answer. Furthermore, we tell Perspect that this parameter is the induction variable by declaring the function with a star in front of the new parameter. Now, *we* know (if we made no mistakes) that this extra parameter is harmless and does not change the meaning of the function. And Perspect will be satisfied that the induction will terminate.

The disadvantage of this method is that we will loose the protection that Perspect normally gives us. If the counter is started at too low a value, the meaning of the specification *is* changed.

This extra parameter has a nice 'application'. In the equation:

```
fractionqp(i,j): qrtoqp(nat(j),divmod(i,j))
```

the 'counter' parameter is started at `nat(j)`. This means that we know that the continued fraction of i/j has depth ≤ j. Now, if we start the parameter at some constant value instead, for example at 3, as in:

```
fractionqp(i,j):
 qrtoqp(nat(succ(nat(succ(nat(succ(0n))))))),divmod(i,j))
```

we will get a specification of *finite precision* arithmetic. In that case, the results of all calculations will be 'rounded' to a close approximation from a finite set of 'simple' fractions.

```
external B
 sort B
 function
  true, false: B
internal B
 [empty]

external Logic
 import B
 function
  not(B), or(B,B), and(B,B): B
internal Logic
 variable
  b: B
 equation
  not(false), or(true,B), or(B,true): true
  not(true), and(false,B), and(B,false): false
  or(false,b), or(b,false), and(true,b), and(b,true): b

external N
 sort N, Zp
 function
  0n, nat(Zp): N
  succ(N): Zp
internal N
 [empty]

external ArithmeticZp
 import N
 function
  addzp(N,Zp), mulzp(Zp,Zp): Zp
internal ArithmeticZp
 variable
  n: N
  i, j: Zp
 equation
  addzp(0n,i): i
```

```
   addzp(n,succ(0n)): succ(n)
   addzp(nat(succ(n)),i), addzp(n,succ(nat(i))):
    succ(nat(addzp(n,i)))
   mulzp(succ(0n),i), mulzp(i,succ(0n)): i
   mulzp(succ(nat(i)),j): addzp(nat(mulzp(i,j)),j)

external ArithmeticN
 import N
 function
  addn(N,N), muln(N,N): N
internal ArithmeticN
 import ArithmeticZp
 variable
  m, n: N
  i, j: Zp
 equation
  addn(0n,n), addn(n,0n): n
  addn(m,nat(i)): nat(addzp(m,i))
  muln(0n,N), muln(N,0n): 0n
  muln(nat(i),nat(j)): nat(mulzp(i,j))

external Z
 sort Z
 import N
 function
  0z, plusz(N), minusz(N): Z
internal Z
 equation
  plusz(0n), minusz(0n): 0z

external ArithmeticZ
 import Z
 function
  addz(Z,Z), subz(Z,Z), mulz(Z,Z): Z
internal ArithmeticZ
 import ArithmeticN
 variable
  m, n: N
  i, j: Z
 function
  minus(Z): Z
 equation
  minus(0z): 0z
  minus(plusz(n)): minusz(n)
  minus(minusz(n)): plusz(n)
```

```
 function
  subn(N,N): Z
 equation
  subn(n,0n): plusz(n)
  subn(0n,n): minusz(n)
  subn(nat(succ(m)),nat(succ(n))): subn(m,n)
 equation
  addz(0z,i), addz(i,0z): i
  addz(plusz(m),plusz(n)): plusz(addn(m,n))
  addz(plusz(m),minusz(n)): subn(m,n)
  addz(minusz(m),plusz(n)): subn(n,m)
  addz(minusz(m),minusz(n)): minusz(addn(m,n))
  subz(i,j): addz(i,minus(j))
  mulz(0z,Z), mulz(Z,0z): 0z
  mulz(plusz(m),plusz(n)): plusz(muln(m,n))
  mulz(plusz(m),minusz(n)): minusz(muln(m,n))
  mulz(minusz(m),plusz(n)): minusz(muln(m,n))
  mulz(minusz(m),minusz(n)): plusz(muln(m,n))

external ComparisonZ
 import Z, B
 function
  posz(Z), negz(Z), zeroz(Z): B
internal ComparisonZ
 equation
  posz(plusz(nat(Zp))), negz(minusz(nat(Zp))), zeroz(0z): true
  posz(minusz(N)), posz(0z), negz(plusz(N)), negz(0z),
   zeroz(plusz(nat(Zp))), zeroz(minusz(nat(Zp))): false

external QR
 sort QR
 import N
 function
  q0(Zp,Zp), qr(N,Zp,Zp): QR
internal QR
 [equation
  q0(i,j) = qr(nat(i),"0",j)]

external DivMod
 import QR, Z
 function
  divmod(Zp,Zp): QR
internal DivMod
 variable
  n: N
  i, j: Zp
```

```
 function
  succqr(QR): QR
 equation
  succqr(q0(i,succ(0n))): q0(succ(nat(i)),succ(0n))
  succqr(q0(i,succ(nat(j)))): qr(nat(i),succ(0n),j)
  succqr(qr(n,i,succ(0n))): q0(succ(n),succ(nat(i)))
  succqr(qr(n,i,succ(nat(j)))): qr(n,succ(nat(i)),j)
  divmod(succ(0n),succ(0n)): q0(succ(0n),succ(0n))
  divmod(succ(0n),succ(nat(i))): qr(0n,succ(0n),i)
  divmod(succ(nat(i)),j): succqr(divmod(i,j))

external Qp
 sort Qp, Qf
 import N
 function
  wholep(Zp), brokenp(N,Qf): Qp
  invadd1(Qp): Qf
internal Qp
 [empty]

external Q
 sort Q
 import Z, Qp
 function
  whole(Z), broken(Z,Qf): Q
internal Q
 [empty]

external Inv
 import Qp
 function
  inv(Qp): Qp
internal Inv
 variable
  n: N
  i: Zp
  f: Qf
  x: Qp
 equation
  inv(wholep(succ(0n))): wholep(succ(0n))
  inv(wholep(succ(nat(i)))): brokenp(0n,invadd1(wholep(i)))
  inv(brokenp(0n,invadd1(wholep(i)))): wholep(succ(nat(i)))
  inv(brokenp(0n,invadd1(brokenp(n,f)))): brokenp(nat(succ(n)),f)
  inv(brokenp(nat(succ(n)),f)): brokenp(0n,invadd1(brokenp(n,f)))
```

```
external PlusMinus
 import Q
 function
  plus(Qp), minus(Qp): Q
internal PlusMinus
 import Inv
 variable
  n: N
  i: Zp
  f: Qf
  x: Qp
 equation
  plus(wholep(i)): whole(plusz(nat(i)))
  plus(brokenp(n,f)): broken(plusz(n),f)
 function
  1sub(Qf): Qf
 equation
  1sub(invadd1(x)): invadd1(inv(x))
 equation
  minus(wholep(i)): whole(minusz(nat(i)))
  minus(brokenp(n,f)): broken(minusz(nat(succ(n))),1sub(f))

external TruncationQ
 import Q
 function
  truncq(Q): Z
  fractq(Q): Q
internal TruncationQ
 variable
  i: Z
  f: Qf
 equation
  truncq(whole(i)), truncq(broken(i,Qf)): i
  fractq(whole(Z)): whole(0z)
  fractq(broken(Z,f)): broken(0z,f)

external FractionsQ
 import Q
 function
  fractionq(Z,Zp): Q
  numeqp(Qp), denoqp(Qp): Zp
  numeq(Q): Z
  denoq(Q): Zp
internal FractionsQ
 import ArithmeticZp, ArithmeticN, ArithmeticZ, DivMod, PlusMinus
```

```
 variable
  m, n: N
  i, j: Zp
  k: Z
  f: Qf
  x: Qp
 function
  qrtoqp(*N,QR), fractionqp(Zp,Zp): Qp
 equation
  qrtoqp(N,q0(i,Zp)): wholep(i)
  qrtoqp(0n,qr(n,Zp,Zp)): wholep(succ(n))
  qrtoqp(nat(succ(m)),qr(n,i,j)):
   brokenp(n,invadd1(qrtoqp(m,divmod(j,i))))
  fractionqp(i,j): qrtoqp(nat(j),divmod(i,j))
 equation
  fractionq(0z,Zp): whole(0z)
  fractionq(plusz(nat(i)),j): plus(fractionqp(i,j))
  fractionq(minusz(nat(i)),j): minus(fractionqp(i,j))
 function
  numeqf(Qf), rec denoqf(Qf),
  rec echo numeqp(Qp), rec echo denoqp(Qp): Zp
 equation
  numeqf(invadd1(x)): denoqp(x)
  denoqf(invadd1(x)): addzp(nat(numeqp(x)),denoqp(x))
 equation
  numeqp(wholep(i)): i
  denoqp(wholep(Zp)): succ(0n)
  numeqp(brokenp(n,f)): addzp(muln(n,nat(denoqf(f))),numeqf(f))
  denoqp(brokenp(N,f)): denoqf(f)
 equation
  numeq(whole(k)): k
  denoq(whole(Z)): succ(0n)
  numeq(broken(k,f)):
   addz(mulz(k,plusz(nat(denoqf(f)))),plusz(nat(numeqf(f))))
  denoq(broken(Z,f)): denoqf(f)

external ArithmeticQ
 import Q
 function
  addq(Q,Q), subq(Q,Q), mulq(Q,Q), divq(Q,Qp): Q
internal ArithmeticQ
 import ArithmeticZp, ArithmeticZ, FractionsQ
 variable
  x, y: Q
  z: Qp
```

```
 equation
  addq(x,y):
   fractionq(addz(mulz(numeq(x),plusz(nat(denoq(y)))),
     mulz(plusz(nat(denoq(x))),numeq(y))),
    mulzp(denoq(x),denoq(y)))
  subq(x,y):
   fractionq(subz(mulz(numeq(x),plusz(nat(denoq(y)))),
     mulz(plusz(nat(denoq(x))),numeq(y))),
    mulzp(denoq(x),denoq(y)))
  mulq(x,y):
   fractionq(mulz(numeq(x),numeq(y)),mulzp(denoq(x),denoq(y)))
  divq(x,z):
   fractionq(mulz(numeq(x),plusz(nat(denoqp(z)))),
    mulzp(denoq(x),numeqp(z)))

external Qe
 sort Qe
 import Q
 function
  num(Q), inf, nan: Qe
internal Qe
 [empty]

external Truncation
 import Qe
 function
  trunc(Qe), fract(Qe): Qe
internal Truncation
 import TruncationQ
 variable x: Q
 equation
  trunc(num(x)): num(whole(truncq(x)))
  fract(num(x)): num(fractq(x))
  trunc(inf): inf
  fract(inf), trunc(nan), fract(nan): nan

external Fractions
 import Qe
 function
  fraction(Z,Z): Qe
  nume(Qe), deno(Qe): Z
internal Fractions
 import FractionsQ, ArithmeticQ
 variable
  i, j: Z
  k: Zp
```

```
    x: Q
   equation
    fraction(i,plusz(nat(k))): num(fractionq(i,k))
    fraction(i,minusz(nat(k))): num(subq(whole(0z),fractionq(i,k)))
    fraction(plusz(nat(Zp)),0z), fraction(minusz(nat(Zp)),0z): inf
    fraction(0z,0z): nan
    nume(num(x)): numeq(x)
    deno(num(x)): plusz(nat(denoq(x)))
    nume(inf): plusz(nat(succ(0n)))
    deno(inf), nume(nan), deno(nan): 0z

  external Arithmetic
   import Qe
   function
    add(Qe,Qe), sub(Qe,Qe), mul(Qe,Qe), div(Qe,Qe): Qe
  internal Arithmetic
   import ArithmeticZ, ArithmeticQ, Fractions
   variable
    x, y: Q
    s, t: Qe
   equation
    add(num(x),num(y)): num(addq(x,y))
    add(inf,num(Q)), add(num(Q),inf): inf
    add(inf,inf), add(nan,Qe), add(Qe,nan): nan
    sub(num(x),num(y)): num(subq(x,y))
    sub(inf,num(Q)), sub(num(Q),inf): inf
    sub(inf,inf), sub(nan,Qe), sub(Qe,nan): nan
    add(s,t):
     fraction(addz(mulz(nume(s),deno(t)),mulz(deno(s),nume(t))),
      mulz(deno(s),deno(t)))
    sub(s,t):
     fraction(subz(mulz(nume(s),deno(t)),mulz(deno(s),nume(t))),
      mulz(deno(s),deno(t)))
    mul(num(x),num(y)): num(mulq(x,y))
    mul(inf,inf): inf
    mul(nan,Qe), mul(Qe,nan): nan
    mul(s,t): fraction(mulz(nume(s),nume(t)),mulz(deno(s),deno(t)))
    div(s,t): fraction(mulz(nume(s),deno(t)),mulz(deno(s),nume(t)))

  external Comparison
   import Qe, B
   function
    pos(Qe), neg(Qe),
    eq(Qe,Qe), ne(Qe,Qe),
    lt(Qe,Qe), le(Qe,Qe), ge(Qe,Qe), gt(Qe,Qe),
```

```
   un(Qe,Qe): B
internal Comparison
 import Logic, ComparisonZ, Fractions, Arithmetic
 variable
  x, y: Q
  s, t: Qe
 equation
  pos(num(x)): posz(nume(num(x)))
  neg(s): negz(nume(s))
  pos(inf), pos(nan), neg(inf), neg(nan): false
  eq(num(x),num(y)): zeroz(nume(sub(num(x),num(y))))
  eq(inf,inf): true
  eq(inf,num(Q)), eq(num(Q),inf), eq(nan,Qe), eq(Qe,nan): false
  ne(s,t): not(eq(s,t))
  lt(s,t): pos(sub(t,s))
  le(s,t): or(lt(s,t),eq(s,t))
  ge(s,t): le(t,s)
  gt(s,t): lt(t,s)
  un(s,t): not(or(eq(s,t),or(lt(s,t),gt(s,t))))
  un(inf,num(Q)), un(num(Q),inf), un(nan,Qe), un(Qe,nan): true

external Choice
 import B, Qe
 function
  if(B,Qe,Qe): Qe
internal Choice
 variable
  s: Qe
 equation
  if(true,s,Qe), if(false,Qe,s): s
```

3.3.3. Primitive recursive functions

In this section we will specify the primitive recursive functions. The sorts that will be specified are `N`, the sort of the natural numbers, `LIST`, the sort of the lists of natural numbers and `PRIM`, the sort of the primitive recursive functions.

It is customary to define a primitive recursive function to have an *arity*, the number of arguments the function takes, which can be any natural number. However, in order to specify this kind of functions we must introduce a separate sort to contain the functions of a given arity, and an application function for each arity as well. Clearly, in this way we would need an infinite signature which cannot be specified in Perspect. One can imagine specifying only the subalgebra restricted to some finite sub-signature, but such a specification is then not as general as possible.

In this specification, the primitive recursive functions do not act on several argu-

ments. Instead, they operate on one object that consists of a list of natural numbers: an object from sort LIST. Furthermore, we do not want to introduce a sort consisting of lists of primitive functions. For this reason it turns out to be convenient that the result of a primitive recursive function also is a member from LIST.

So, we have a sort of primitive recursive functions PRIM, and an application function:

```
appl(PRIM,LIST): LIST
```

that takes a primitive recursive function together with a list of natural numbers and produces a list of natural numbers.

This function appl is total. This means that it also has to give some answer when the number of elements in the LIST is inappropriate. We will use the convention here that if the number of arguments in the LIST is too low, the 'missing' arguments will be taken to be zero. So, the equation:

```
appl(F,append(x,list(0))): appl(F,x)
```

is supposed to hold. Conversely, if the number of arguments given to appl is too high the 'extra' arguments will conveniently be ignored.

The specification of the sort PRIM is straightforward. The algebra is freely generated by the functions zero, succ, proj, concat, compose and primrec. The meaning of these generators can be seen from the various equations in module appl.

The specification of the application function appl also looks simple. Its only subtlety lies in the star in front of the first argument in the declaration of appl. The meaning of this is that the 'function' argument of appl is the argument that appl is inductively defined over. In fact, the main reason for including the path ordering with argument status in Perspect was that it made it possible to write this specification. If the star in front of the first argument of appl had been missing, the equation:

```
appl(compose(F,G),x): appl(F,appl(G,x))
```

would not have been decreasing in the path ordering, and so the termination of the evaluation of the appl function could not have been verified by the Perspect system.

The specification also contains a function enum, that enumerates all primitive recursive functions as a function of the natural numbers. This function can be used for the construction of diagonal functions.

In order to be able to define enum, we must first define functions that code the product N × N in the sort N itself. These functions, called left and right, are defined in module split. These functions are defined with the aid of a function next that acts as a 'successor' on pairs of natural numbers which have been defined as the sort PAIR.

Now, the function enum is defined recursively in a way that is not straightforward. The reason for this is the termination check that has to be satisfied in Perspect. If this check had not been present, the equations defining enum would have been:

```
enum(n): enum1(left(n),right(n))
enum1(0,N): zero
enum1(S(0),N): succ
enum1(S(S(0)),n): proj(n)
enum1(S(S(S(0))),n): concat(enum(left(n)),enum(right(n)))
enum1(S(S(S(S(0)))),n): compose(enum(left(n)),enum(right(n)))
enum1(S(S(S(S(S(N))))),n): primrec(enum(left(n)),enum(right(n)))
```

In these equations the 'left' part of the coding number is a code for the kind of function – `zero`, `succ`, `concat`, `compose` or `primrec` – while the 'right' part codes the parameters, if any.

In order to be able to verify that the recursion eventually peters out (which it does), we have added another parameter (declared with a '`*`' to indicate that it is the induction parameter) that gives an upper bound for the number of recursions that are allowed; the function `enum` becomes `enum2` with the extra parameter and the function `enum1` becomes `enum3`. If this parameter is exceeded the function that is coded is `zero`, regardless of the code. The same trick was also present in the specification of the rational numbers in section 3.3.2.

With the enumeration function `enum` in hand we are able to define a diagonal. In fact we define two diagonals, `diag` and `xdiag`. The function `diag` is the standard diagonal. It is constructed by using the fact that:

```
n → car(appl(F,list(n)))
```

is a function from `N` to `N`, for any element `F` from `PRIM`. If we replace `F` by `enum(n)` and subsequently take the successor we have the desired diagonal.

However we do not know whether the algebra specified by `diag` is a primitive recursive algebra. So, in order to specify an algebra of which we *do* know this property we have also specified the function called `xdiag`. This is the function referred to in section 3.2.4.

The function `xdiag` has the property that if F and G are primitive recursive functions on the natural numbers, and if F is a bijection on the natural numbers as well, then `xdiag` is a function different from $F^{-1} \cdot G$. This is equivalent to saying that for each such F and G there exists some natural number n for which:

$$F(\texttt{xdiag}(n)) \neq G(n)$$

Now, `xdiag` will have this property even for functions F and G for which the function F is not a bijection on the natural numbers, but only has the property that $F(0) \neq F(1)$. This is a consequence of the definition of `xdiag`. For each primitive recursive F and G with $F(0) \neq F(1)$ there is a natural number n with:

$$F(0) = G(n) \Rightarrow \texttt{xdiag}(n) \equiv 1$$
$$F(0) \neq G(n) \Rightarrow \texttt{xdiag}(n) \equiv 0$$

In both cases it is easy to verify that then $F(\texttt{xdiag}(n)) \neq G(n)$. This definition can be re-

formulated by saying that xdiag(n) 'codes' the equality between F(0) and G(n).

So, using this method to construct a diagonal function, we see that the definition of xdiag is:

```
xdiag(n):
 eq(car(appl(enum(left(n)),list(0))),
    car(appl(enum(right(n)),list(n))))
```

It should be noted that the signature of module xdiag is simple. It only has one sort, i.e., N, and only three functions: 0, S and xdiag. As we already said in section 3.2.4 the algebra specified by this module is not primitive recursive. So, this specification shows explicitly that the class of algebras that can be specified in Perspect is strictly greater then the class of primitive recursive algebras.

```
external N
 sort N
 function 0, S(N): N
 variable n, m: N
internal N
 [empty]

external LIST
 import N
 sort LIST
 function
  nil, cons(N,LIST), list(N), append(LIST,LIST): LIST
  car(LIST): N
 variable x, y: LIST
internal LIST
 equation
  list(n): cons(n,nil)
  append(nil,y): y
  append(cons(n,x),y): cons(n,append(x,y))
  car(nil): 0
  car(cons(n,LIST)): n

external PRIM
 import N
 sort PRIM
 function
  zero, succ, proj(N), concat(PRIM,PRIM), compose(PRIM,PRIM),
  primrec(PRIM,PRIM):
   PRIM
 variable F, G: PRIM
internal PRIM
```

```
  [empty]

external appl
 import LIST, PRIM
 function appl(*PRIM,LIST): LIST
internal appl
 equation
  appl(zero,LIST): list(0)
  appl(succ,nil): list(S(0))
  appl(succ,cons(n,LIST)): list(S(n))
  appl(proj(N),nil): list(0)
  appl(proj(0),cons(n,LIST)): list(n)
  appl(proj(S(n)),cons(N,x)): appl(proj(n),x)
  appl(concat(F,G),x): append(appl(F,x),appl(G,x))
  appl(compose(F,G),x): appl(F,appl(G,x))
  appl(primrec(F,PRIM),nil): appl(F,nil)
  appl(primrec(F,PRIM),cons(0,x)): appl(F,x)
  appl(primrec(F,G),cons(S(n),x)):
   appl(G,cons(n,append(appl(primrec(F,G),cons(n,x)),x)))

external split
 import N
 function left(N), right(N): N
internal split
 sort PAIR
 function pair(N,N): PAIR
 function first(PAIR), second(PAIR): N
 equation
  first(pair(n,N)), second(pair(N,n)): n
 function next(PAIR), enump(N): PAIR
 equation
  next(pair(0,n)): pair(S(n),0)
  next(pair(S(m),n)): pair(m,S(n))
  enump(0): pair(0,0)
  enump(S(n)): next(enump(n))
  left(n): first(enump(n))
  right(n): second(enump(n))

external enum
 import PRIM
 function enum(N): PRIM
internal enum
 import split
 function enum2(*N,N), rec enum3(*N,N,N): PRIM
 equation
```

```
   enum(n): enum2(S(n),n)
   enum2(0,N), enum3(0,N,N): zero
   enum2(S(m),n): enum3(m,left(n),right(n))
   enum3(S(N),0,N): zero
   enum3(S(N),S(0),N): succ
   enum3(S(N),S(S(0)),n): proj(n)
   enum3(S(m),S(S(S(0))),n):
    concat(enum2(m,left(n)),enum2(m,right(n)))
   enum3(S(m),S(S(S(S(0)))),n):
    compose(enum2(m,left(n)),enum2(m,right(n)))
   enum3(S(m),S(S(S(S(S(N))))),n):
    primrec(enum2(m,left(n)),enum2(m,right(n)))

external diag
 import N
 function diag(N): N
internal diag
 import appl, enum
 equation
  diag(n): S(car(appl(enum(n),list(n))))

external xdiag
 import N
 function xdiag(N): N
internal xdiag
 import appl, split, enum
 function eq(N,N): N
 equation
  eq(0,0): S(0)
  eq(S(N),0), eq(0,S(N)): 0
  eq(S(m),S(n)): eq(m,n)
 equation
  xdiag(n):
   eq(car(appl(enum(left(n)),list(0))),
      car(appl(enum(right(n)),list(n))))
```

3.3.4. Stacks

It has once been remarked that the *stack* appears to be an inescapable example in the realm of algebraic specifications [Bergstra, Tucker, 1988]. In this section we will therefore give two specifications of the notion of a stack. The first is a nice variant of the conventional solution to this problem. The second is the correct way to do it.

A stack is an object on which three operations are defined called: `push`, `pop` and `top`. The operation `push` adds an element to the stack, the operation `pop` removes the top element of the stack, and the operation called `top` tells what that top element is.

The problem of specifying a stack lies in the conflict that seems to exist between the requirement that we use total functions in our specification, and the fact that on the empty stack the operations `pop` and `top` are not well defined. If we denote the empty stack by the identifier `empty`, the expressions `pop(empty)` and `top(empty)` pose a problem.

The customary solution for this problem is to *extend* the sort, `D`, of the data elements that can be pushed on the stack by a new element called `error`. This gives us a new sort, `eD` (extended data), in which the sort `D` is embedded by the embedding function `i`. Now the problem of the expression `top(empty)` can be solved by equating it to the `error` object.

However, the expression `pop(empty)` is still a problem. At this point there are a number of choices, as is described in [Bergstra, Tucker, 1988]. The solution that we will show here, which corresponds to specification `SAE4` in [Bergstra, Tucker, 1988], consists of considering the empty stack `empty` to be an infinite stack of `error`s. This way, the specification has two nice properties. First, if:

```
top(s1) = top(s2) & pop(s1) = pop(s2)
```

then stack `s1` has to be equal to stack `s2`. Second, the nice equation:

```
push(top(s),pop(s)): s
```

holds. Note that this equation cannot be expressed in Perspect because it is not left linear. This means we cannot add it to the specification in order to make the Perspect system verify it.

The first specification from this section differs slightly from the specification called `SAE4` in [Bergstra, Tucker, 1988]. First, there is no sort consisting of the stacks of objects of type `D`. In [Bergstra, Tucker, 1988] that sort was hidden, and it is clear that it was not needed in this specification. Second, we have added a *projection* function `j` that projects sort `eD` back on sort `D`. In a sense it is the inverse of the embedding function `i`. It has two arguments: if one tries to project the `error` object on sort `D`, the second argument tells what the result will be. This function is not strictly needed, because it can be defined by any module that might need it. However, when the function `j` is present, one does not need to rely on the specific term rewriting system that is given here when using this specification.

```
external DATA
 sort D
 function d0, d1: D
internal DATA
 [empty]

external STACKS
 import DATA
 sort eD, S
```

```
 function
  i(D), error: eD
  j(eD,[if error:]D): D
  empty, push(eD,S), pop(S): S
  top(S): eD
internal STACKS
 variable
  d: D
  e: eD
  s: S
 equation
  j(i(d),D), j(error,d): d
  push(error,empty): empty
  pop(empty): empty
  pop(push(eD,s)): s
  top(empty): error
  top(push(e,S)): e
```

There is a better solution for solving the problem of the undefinedness of the expressions `pop(empty)` and `top(empty)`. It has the same two pleasant properties as the first solution, but it avoids the need for error objects.

The solution is simple. It consists of the observation that the expressions `pop(empty)` and `top(empty)` are not well typed, because the operations `pop` and `top` cannot be defined on a sort that contains the object `empty`.

Instead of extending the sort `D` in order to accommodate the problem of the expression `top(empty)` we *restrict* the sort `S`. This restricted sort we call `neS` (which consists of the non-empty stacks). We define the operations `top` and `pop` only on this new sort, which we embed by an embedding function `i` in the sort `S` of all stacks.

Again we have a projection function `j` that is kind of an inverse to `i`. This makes it possible to write for example:

```
pop(j(pop(j(i(push(d1,i(push(d0,empty)))),neS)),neS))
```

when we want to describe the stack that we get when we push two elements, `d0` and `d1`, on the empty stack and then pop them both again. Clearly, this open expression which contains two anonymous variables `neS`, reduces to the expression `empty`.

This second specification could be made more clear by using a specification system in which the embedding of sorts is built in the language (the best known of these languages is OBJ2, see for example [Futatsugi, Goguen, Jouannaud, Meseguer, 1985]). One can consider this specification to be a translation into Perspect of a specification in such a language.

```
external DATA
 sort D
 function d0, d1: D
```

```
internal DATA
 [empty]

external STACKS
 import DATA
 sort S, neS
 function
  empty, i(neS), pop(neS): S
  push(D,S), j(S,[if empty:]neS): neS
  top(neS): D
internal STACKS
 variable
  d: D
  s: S
  t: neS
 equation
  j(i(t),neS), j(empty,t): t
  pop(push(D,s)): s
  top(push(d,S)): d
```

The solution for avoiding the problem of the partiality of some functions by introducing new sorts (which was also used in the previous specification) has a wide range of applicability. We would like to argue for the adoption of the following analogon for pointers in current (Pascal like) languages. This analogon for pointers was brought to our attention by [Pemberton, 1989].

Consider a Pascal like language (entirely fictional) with the following constructs. If `T` is a datatype, there is the datatype **`ptr`**`(T)` in the language that consists of the pointers to objects of type `T`. If `E` is an expression of type **`ptr`**`(T)`, then the dereferencing operator denoted by `*` gives the object `*E` which is an object of type `T`. The expression **`new`**`(T)` gives a 'new' pointer of type **`ptr`**`(T)`. Also, there is an expression **`nil`**`(T)` which gives the 'null' pointer of type **`ptr`**`(T)`. Now, conventionally the expression **`*nil`**`(T)` is not well defined. A program that encounters such an expression can do anything at all (perhaps it crashes or maybe it prints '`(null)`'). This problem is the source of a large class of programming errors in conventional computer programs.

There exists a solution that solves this problem by relying on the type system of the language. It consists of replacing the type **`ptr`**`(T)` by *two* types, **`ref`**`(T)` and **`ptr`**`(T)` which are respectively the type of the non-null pointers and that of the pointers that might be null. The `*` operator is then only defined on the first kind of pointers and the expression **`new`**`(T)` has this type as well, while the expression **`nil`**`(T)` is a pointer of the second kind.

Now, an expression of type **`ref`**`(T)` can be used in any context in which a **`ptr`**`(T)` is required, because it is a subtype. However, the inverse operation (the one that we called `j` in the stack specification) cannot be implicitly present. For this we have to add a statement to the language. Let `r` be a variable of type **`ref`**`(T)` and `p` an expression of type **`ptr`**`(T)` (possibly also a variable) and let `S` be some statement. A state-

ment of the following structure can then be added to the language as a way to convert a **ptr**(T) to a **ref**(T) (the exact syntax is immaterial):

```
let
 r := p
if null
 S
tel;
```

If the pointer that is the value of p is non null this value is assigned to r. If it is null, the statement S is executed. This statement could allocate a new block, abort the program with a diagnostic or take some other appropriate action.

A language that adopts this type structure will never abort because a null pointer is being dereferenced. This shows that strong typing is powerful enough to 'solve' the problem of dereferencing null pointers.

3.3.5. Arrays

In section 3.3.6 we will give an example with a practical flavor: text and pictures. When we wrote it, it turned out that the main problem was finding a form that was open confluent. This led to the following specification, which specifies unbounded arrays, in which this problem has been isolated. Therefore, there is some overlap here (modules NaturalNumbers, Integers and IntegerOperations) with the specification in the next section.

The specification in this section describes arrays without explicit bounds, that are indexed using integers. The arrays can be considered to be a doubly infinite list that contains in almost all positions, apart from a finite number of places, the value undefined. An auxiliary sort for constructing these arrays is the sort of the Pre-Arrays. These are arrays that at the left hand side continue forever, but at the right hand side end at some finite index.

The representation of an array in this specification might become clear in the following example. In the procedural fragment:

```
var
 a: array [-2..9] of color;
begin
 a[-1] := red;
 a[1] := white;
 a[2] := blue
end;
```

the variable a will in the end contain a value, that would be represented in the specification by the term:

```
array(
 conc(conc(conc(conc(conc(conc(conc(conc(conc(conc(conc(
  index(minus(S(S(0)))),
   undefined),datum(red)),undefined),datum(white)),datum(blue)),
   undefined),undefined),undefined),undefined),undefined),
   undefined),undefined))
```

So, the generators of sort `Array` are:
- the function `index` to start an array,
- the function `conc` to extend an array with an element, and
- the function `array` to close an array (or more exactly to convert a `PreArray` to an `Array`).

Because the arrays in the specification do not contain information about their bounds, this term for array `a` is equal to the normal form:

```
array(conc(conc(conc(conc(
 index(minus(S(0))),
  datum(red)),undefined),datum(white)),datum(blue)))
```

Note that the array bounds -2 and 9 have been lost here. Because of the first two equations in module `Arrays` a normal form of sort `Array` will always start at the first filled element (here: at index -1), and will always end at the last filled element.

The functions that are used to give an element of an array a certain value, respectively recover that value, are called `set` and `get`. The term:

```
set(x,i,a)
```

corresponds to the state of the array `a` after the statement:

```
a[i] := x
```

and the term:

```
get(i,a)
```

corresponds to the value:

```
a[i]
```

In order to be able to define these functions, one needs a number of auxiliary functions:
- the function `end` gives the upper limit of a `PreArray`,
- the function `extend` adds an element to a `PreArray` at the right hand side,
- the function `insert` changes an existing element of a `PreArray`,
- the functions `setrel` and `getrel` are to `PreArrays` what `set` and `get` are to

`Arrays`. The index that is the argument is in this case not absolute, but relative to the end of the `PreArray` (which explains the suffix `rel`). The difference between `setrel` and `insert` is subtle: `setrel` can modify elements outside the `PreArray` (and in that case extends it); `insert` is not able to do this. This difference can also be seen in the difference in domain of these functions: `setrel` takes an `Int` for an index, `insert` takes a `Nat`.

In this specification an equation occurs that is on purpose more specific than appears to be necessary. The equation is:

```
set(x,i,array(conc(a,datum(d)))):
 setrel(x,dec(sub(i,end(a))),conc(a,datum(d)))
```

(note that the subterm `dec(sub(i,end(a)))` of the right hand side of this equation is a reduct of the term `sub(i,end(conc(a,datum(d))))`)

This rule is also true in the more general case:

```
set(x,i,array(a)):
 setrel(x,sub(i,end(a)),a)
```

If the specification had contained this more general equation, it would not have been open confluent, and it would have been hard to find extra equations to restore open confluence.

```
external NaturalNumbers
 sort Nat
 variable
  m, n: Nat
 function
  0, S(Nat): Nat
internal NaturalNumbers
 [empty]

external Integers
 sort Int
 variable
  i, j, k: Int [a la Fortran]
 import NaturalNumbers
 function
  zero, plus(Nat), minus(Nat): Int
internal Integers
 equation
  plus(0), minus(0): zero

external IntegerOperations
 import Integers
```

```
 function
  inc(Int), dec(Int), sub(Int,Int): Int
internal IntegerOperations
 equation
  inc(zero): plus(S(0))
  inc(plus(n)): plus(S(n))
  inc(minus(S(n))): minus(n)
  dec(zero): minus(S(0))
  dec(plus(S(n))): plus(n)
  dec(minus(n)): minus(S(n))
  sub(i,zero): i
  sub(i,plus(S(n))): dec(sub(i,plus(n)))
  sub(i,minus(S(n))): inc(sub(i,minus(n)))
  sub(plus(S(n)),i): inc(sub(plus(n),i))
  sub(minus(S(n)),i): dec(sub(minus(n),i))
 [properties]
  inc(dec(i)), dec(inc(i)): i
  sub(inc(i),j), sub(i,dec(j)): inc(sub(i,j))
  sub(dec(i),j), sub(i,inc(j)): dec(sub(i,j))

external Data
 sort Datum
 variable
  d: Datum
internal Data
 function someDatum, another(Datum): Datum

external ArrayElements
 import Data
 sort Element
 variable
  x, y: Element
 function
  datum(Datum), undefined: Element
internal ArrayElements
 [empty]

external Arrays
 sort Array
[constructors]
 import Integers, ArrayElements
 sort PreArray
 variable
  a: PreArray
 function
```

```
    index(Int), conc(PreArray,Element): PreArray
    empty, array(PreArray): Array
  internal Arrays
   import IntegerOperations
   equation
    conc(index(i),undefined): index(inc(i))
    array(conc(a,undefined)): array(a)
    array(index(Int)): empty

  external ArrayAccess
   import Arrays
   function
    set(Element,Int,Array): Array
    get(Int,Array): Element
  internal ArrayAccess
   import IntegerOperations
  [set]
   function
    end(PreArray): Int
   equation
    end(index(i)): i
    end(conc(a,Element)): inc(end(a))
   function
    extend(Nat,PreArray): PreArray
   equation
    extend(S(n),a): conc(extend(n,a),undefined)
    extend(0,a): a
   function
    insert(Element,Nat,PreArray): PreArray
   equation
    insert(x,S(n),conc(a,y)): conc(insert(x,n,a),y)
    insert(x,0,conc(a,Element)): conc(a,x)
    insert(x,n,index(i)):
      extend(n,conc(index(dec(sub(i,plus(n)))),x))
   function
    setrel(Element,Int,PreArray): Array
   equation
    setrel(x,plus(n),a): array(conc(extend(n,a),x))
    setrel(x,zero,a): array(conc(a,x))
    setrel(x,minus(n),a): array(insert(x,n,conc(a,undefined)))
   equation
    set(x,i,array(conc(a,datum(d)))):
     setrel(x,dec(sub(i,end(a))),conc(a,datum(d)))
    set(x,i,empty): array(conc(index(i),x))
  [get]
   function
```

```
 getrel(Int,PreArray): Element
equation
 getrel(Int,index(Int)), getrel(plus(Nat),PreArray),
 getrel(zero,PreArray):
  undefined
 getrel(minus(S(S(n))),conc(a,Element)): getrel(minus(S(n)),a)
 getrel(minus(S(0)),conc(PreArray,x)): x
equation
 get(i,array(conc(a,datum(d)))):
  getrel(dec(sub(i,end(a))),conc(a,datum(d)))
 get(Int,empty): undefined
```

3.3.6. Text and pictures

The specification in this section describes 'real' objects: text and pictures. It was inspired by the Macintosh computer in which the two standard datatypes (for files, resources and scrap) have types `TEXT` and `PICT`.

Macintosh pictures have an internal structure. They are hierarchically constructed from a number of primitive graphical objects like text, lines, rectangles, ovals, etc. They behave nicely under scaling: an enlarged circle is still a circle. The pictures that we will specify here, however, only consist of black and white pixels.

A similarity between the pictures that are specified here and Mac pictures is their locality. If one draws such a picture on a bitmap, the color of one of the pixels after the draw operation only depends on the color of that specific pixel before drawing. For example, unlike a drop of water, a picture cannot take the average of a number of neighboring pixels.

We will now give some explanation of the specification. For the major part the specification is self explanatory. An exception is the representation of pictures.

There are two possible interpretations for the term *picture*. There is the action of drawing the picture, and there is the result of that action, the final drawing itself. The second notion is called a `BitMap` in the specification, the first is called `Pict`.

A `BitMap` consists of a grid of square cells that are either black or white, and in which only finitely many squares are black. We have here a two-dimensional array, which explains the relation to the example in the previous section: the specification here is a double version of the specification of the arrays. The one-dimensional arrays built from `Bits` are called `Rows`, and the associated functions have an identifier ending in an `r`: `concr`, `endr`, `extendr`, etc. The two-dimensional arrays built from `Rows` are called `BitMaps`, and the associated functions have an identifier ending in a `b`: `concb`, `endb`, `extendb`, etc.

The function `set` (with associated auxiliary functions) was used in the previous example to give an element in an array a certain value. Here, the function `set` modifies a pixel in a `BitMap` according to a certain `Mode`. This mode can take three values:

- if the mode is `bic` the pixel is set to `white`,
- if the mode is `or` the pixel is set to `black`, and

- if the mode is `xor` the pixel is inverted: if it was `white` it becomes `black` and if it was `black` it becomes `white`.

To give `set` this functionality, all auxiliary functions associated with `set` have a first argument that is a `Mode`, and that is recursively passed on until the function called `change` makes the required modification.

With the aid of `set` three Boolean operations on `BitMaps` are inductively built. These operations are given by the function `op`, which again has as first argument a `Mode`.

In the specification the *active* pictures, or `Picts`, are built from two `BitMaps`, the image and the mask. Drawing the picture is performed by first clearing the pixels that are black in the mask, and then by inverting the pixels that are black in the image.

Drawing a `Pict` on a `BitMap` can now be expressed in terms of the operations on the `BitMaps`. This is described by the penultimate equation of the specification:

```
draw(pict(image,mask),bits): op(xor,image,op(bic,mask,bits))
```

The last equation describes in a similar way composition of pictures, which is called `seq` in the specification.

```
external Bits
 sort Bit
 variable
  b: Bit
 function
  white, black: Bit
internal Bits
 [empty]

external Bytes
 import Bits
 sort Byte
 function
  byte(Bit,Bit,Bit,Bit,Bit,Bit,Bit,Bit): Byte
internal Bytes
 [empty]

external NaturalNumbers
 sort Nat
 variable
  m, n: Nat
 function
  0, S(Nat): Nat
internal NaturalNumbers
 [empty]
```

```
external Integers
 sort Int
 variable
  i, j, k: Int [a la Fortran]
 import NaturalNumbers
 function
  zero, plus(Nat), minus(Nat): Int
internal Integers
 equation
  plus(0), minus(0): zero

external IntegerOperations
 import Integers
 function
  inc(Int), dec(Int), sub(Int,Int): Int
internal IntegerOperations
 equation
  inc(zero): plus(S(0))
  inc(plus(n)): plus(S(n))
  inc(minus(S(n))): minus(n)
  dec(zero): minus(S(0))
  dec(plus(S(n))): plus(n)
  dec(minus(n)): minus(S(n))
  sub(i,zero): i
  sub(i,plus(S(n))): dec(sub(i,plus(n)))
  sub(i,minus(S(n))): inc(sub(i,minus(n)))
  sub(plus(S(n)),i): inc(sub(plus(n),i))
  sub(minus(S(n)),i): dec(sub(minus(n),i))
 [properties]
  inc(dec(i)), dec(inc(i)): i
  sub(inc(i),j), sub(i,dec(j)): inc(sub(i,j))
  sub(dec(i),j), sub(i,inc(j)): dec(sub(i,j))

external Characters
 import Bytes
 sort Char
 variable
  c: Char
 function
  char(Byte): Char
internal Characters
 [empty]

external Texts
 import Characters
```

```
 sort Text
 variable
  t, t1, t2: Text
 function
  emptyText, conc(Char,Text): Text
internal Texts
 [empty]

external TextOperations
 import Texts
 function
  append(Text,Text): Text
internal TextOperations
 equation
  append(emptyText,t): t
  append(conc(c,t1),t2): conc(c,append(t1,t2))

external BitMaps
 import Bits, Integers, IntegerOperations
 sort PreRow, Row, PreBitMap, BitMap
 variable
  pr: PreRow
  ro: Row
  pb: PreBitMap
  bits: BitMap
 function
  x(Int), concr(PreRow,Bit): PreRow
  emptyRow, row(PreRow): Row
  y(Int), concb(PreBitMap,Row): PreBitMap
  emptyBitMap, bitMap(PreBitMap): BitMap
internal BitMaps
 equation
  concr(x(i),white): x(inc(i))
  row(concr(pr,white)): row(pr)
  row(x(Int)): emptyRow
  concb(y(j),emptyRow): y(inc(j))
  bitMap(concb(pb,emptyRow)): bitMap(pb)
  bitMap(y(Int)): emptyBitMap

external Modes
 sort Mode
 variable
  mode: Mode
 function
  or, xor, bic: Mode
internal Modes
```

```
 [empty]

external PixelOperations
 import BitMaps, Modes
 function
  endr(PreRow), endb(PreBitMap): Int
  set(Mode,[x:]Int,[y:]Int,BitMap): BitMap
internal PixelOperations
 function
  echo endr(PreRow), echo endb(PreBitMap): Int
 equation
  endr(x(i)): i
  endr(concr(pr,Bit)): inc(endr(pr))
  endb(y(j)): j
  endb(concb(pb,Row)): inc(endb(pb))
[change a pixel in a row...]
 function
  change(Mode,Bit), pixelr(Mode): Bit
 equation
  change(or,Bit), change(xor,white): black
  change(xor,black), change(bic,Bit): white
  pixelr(mode): change(mode,white)
 function
  extendr(Nat,PreRow): PreRow
 equation
  extendr(S(n),pr): concr(extendr(n,pr),white)
  extendr(0,pr): pr
 function
  insertr(Mode,Nat,PreRow): PreRow
 equation
  insertr(mode,S(n),concr(pr,b)): concr(insertr(mode,n,pr),b)
  insertr(mode,0,concr(pr,b)): concr(pr,change(mode,b))
  insertr(mode,n,x(i)):
   extendr(n,concr(x(dec(sub(i,plus(n)))),pixelr(mode)))
 function
  setrelr(Mode,Int,PreRow): Row
 equation
  setrelr(mode,plus(n),pr): row(concr(extendr(n,pr),pixelr(mode)))
  setrelr(mode,zero,pr): row(concr(pr,pixelr(mode)))
  setrelr(mode,minus(n),pr): row(insertr(mode,n,concr(pr,white)))
 function
  setr(Mode,Int,Row): Row
 equation
  setr(mode,i,row(concr(pr,black))):
   setrelr(mode,dec(sub(i,endr(pr))),concr(pr,black))
```

```
    setr(mode,i,emptyRow): row(concr(x(i),pixelr(mode)))
```
[... and in a bitmap]
```
  function
   pixelb(Mode,Int): Row
  equation
   pixelb(mode,i): row(concr(x(i),pixelr(mode)))
  function
   extendb(Nat,PreBitMap): PreBitMap
  equation
   extendb(S(n),pb): concb(extendb(n,pb),emptyRow)
   extendb(0,pb): pb
  function
   insertb(Mode,Int,Nat,PreBitMap): PreBitMap
  equation
   insertb(mode,i,S(n),concb(pb,ro)):
    concb(insertb(mode,i,n,pb),ro)
   insertb(mode,i,0,concb(pb,ro)): concb(pb,setr(mode,i,ro))
   insertb(mode,i,n,y(j)):
    extendb(n,concb(y(dec(sub(j,plus(n)))),pixelb(mode,i)))
  function
   setrelb(Mode,Int,Int,PreBitMap): BitMap
  equation
   setrelb(mode,i,plus(n),pb):
    bitMap(concb(extendb(n,pb),pixelb(mode,i)))
   setrelb(mode,i,zero,pb): bitMap(concb(pb,pixelb(mode,i)))
   setrelb(mode,i,minus(n),pb):
    bitMap(insertb(mode,i,n,concb(pb,emptyRow)))
  equation
   set(mode,i,j,bitMap(concb(pb,row(concr(pr,black))))):
    setrelb(mode,i,dec(sub(j,endb(pb))),
     concb(pb,row(concr(pr,black))))
   set(mode,i,j,emptyBitMap): bitMap(concb(y(j),pixelb(mode,i)))

external BitMapOperations
 import BitMaps, PixelOperations
 function
  op(Mode,BitMap,BitMap): BitMap
internal BitMapOperations
 equation
  op(Mode,emptyBitMap,bits): bits
  op(mode,bitMap(concb(pb,row(concr(pr,black)))),bits):
   set(mode,endr(pr),endb(pb),
  op(mode,bitMap(concb(pb,row(pr))),bits))

external Pictures
```

```
 import BitMaps
 sort Pict
 function
  pict([image:]BitMap,[mask:]BitMap): Pict
internal Pictures
 [empty]

external PictureOperations
 import Pictures, BitMapOperations
 function
  draw(Pict,[on:]BitMap): BitMap
  seq([first:]Pict,[second:]Pict): Pict
internal PictureOperations
 variable image, image1, image2, mask, mask1, mask2: BitMap
 equation
  draw(pict(image,mask),bits):
   op(xor,image,op(bic,mask,bits))
  seq(pict(image1,mask1),pict(image2,mask2)):
   pict(op(xor,image2,op(bic,mask2,image1)),op(or,mask1,mask2))
```

3.3.7. Small text editor

We will now specify a simple text editor. It will be used as an example for the scheme for translating term rewriting systems to Prolog that will be described in section 4.1.3.

The program that will be specified here is the sample program from [Apple, 1985]. It is a miniature Macintosh text editor. The style of editing that is used here originated at Xerox PARC in the seventies (and can therefore already be found in the Smalltalk system, as described in [Goldberg, 1984]). The way this style should behave in a Macintosh application is described extensively in [Apple, 1987]. The program is simple, but is representative for this style of editing in that it uses a mouse for selecting text and that it supports the operations 'cut', 'copy' and 'paste'.

The program that is being specified here is simple. It has one fixed window in which text can be entered by typing it. The mouse can be used for making selections. There are three menus: the apple menu, the file menu and the edit menu. The apple menu does not contain commands for the program, but only provides access to the desk accessories in the system. In the specification these are not present. The file menu contains the 'quit' command. The edit menu contains the commands 'cut', 'copy', 'paste' and 'clear'.

The three functions in the specification that give the representation of the program are:

```
welcomeToMacintosh, transition(State,Event): State
show(State): Screen
```

Clearly, this program is *event driven*. We will now highlight some noticeable details of the specification:

- The natural numbers in the specification can be given with the aid of the function called `addDigit`. This function 'adds' its second argument, if it is a digit, at the end of the number given by its first argument. This means that the value of `addDigit(Na1,Na2)` is equal to 10 * Na1 + Na2. The function `addDigit` will not be used in the specification. It is only present for convenience when operating the Prolog translation from section 4.1.3.
- Objects of type `List` are constructed from the generators `emptyList` and `dot`. The function `emptyList` is called `nil` in Lisp and `[]` in Prolog. The expression `dot(It1,Li1)` corresponds to the Lisp expression `(cons It1 Li1)` and to the (equivalent) Prolog expressions `[It1|Li1]`, `It1.Li1` and `'.'(It1,Li1)`. The last two forms explain the use of the identifier '`dot`'.
- A text containing a, possibly empty, selection (which is shown on the screen as inverted text), is called a `Document` in the specification. It consists of three `Strings`, the part in front of the selection, the selection itself and the part after the selection. If the selection is empty (there will then be a blinking cursor called *caret* in the appropriate location on the screen) it will be represented by the empty string. As an aid for the construction of objects of type `Document`, there are three functions in the specification called `front`, `middle` and `back`, which partition a `String` according to two `Naturals` that are taken to be indices in the `String`.
- The specified system has two kinds of `States`, on the one hand `States` in which the application is running and on the other hand one `State` called `finder` outside the application. A `State` that describes the running application consists of a `Document` being edited, and a `String` representing the clipboard. The `State` outside the application does not retain the clipboard. The real program of which this specification is an abstraction also has this property of not retaining the clipboard.
- It is remarkable that after a `paste` operation the selection in the text is empty instead of consisting of the `pasted` material. This means that the equation specifying the paste operation is:

```
transition(state(document(Str1,String,Str3),Str4),paste):
 state(document(append(Str1,Str4),empty,Str3),Str4)
```

  instead of the more straightforward:

```
transition(state(document(Str1,String,Str3),Str4),paste):
 state(document(Str1,Str4,Str3),Str4)
```

- The way `Events` are specified is too simple. The selection of a command in a pull down menu is represented by an object from the sort `Event`. This means that it is possible to give commands from menus in contexts in which they are not present in the menus at all. For example, it is possible to `quit` the application when it is not running, or to `open` it when it *is* already running. In the specification these `Events` are inert, in reality they are impossible.

This specification is a nice example of the fact that it is possible to sweep a lot of detail under the rug when specifying a system algebraically. It is remarkable that, after omitting so much (for instance, the windowing system that is necessary to operate the real program cannot be found in the specification at all), the Prolog translation from section 4.1.3 shows that the specification captures the 'look and feel' of the specified program neatly.

```
external NATURAL
 sort Natural
 function
  0, succ(Natural),
  1, 2, 3, 4, 5, 6, 7, 8, 9,
  addDigit(*Natural,Natural): Natural
 variable Na1, Na2: Natural
internal NATURAL
 equation
  1: succ(0)  2: succ(1)  3: succ(2)
  4: succ(3)  5: succ(4)  6: succ(5)
  7: succ(6)  8: succ(7)  9: succ(8)
  addDigit(0,Na1): Na1
  addDigit(succ(Na1),Na2):
   addDigit(Na1,
    succ(succ(succ(succ(succ(
    succ(succ(succ(succ(succ(Na2))))))))))

external CHAR
 sort Char
 function
  a, b, c, d, e, f, g, h, i, j, k, l, m,
  n, o, p, q, r, s, t, u, v, w, x, y, z,
  space, newLine: Char
 variable Chr1, Chr2: Char
internal CHAR
 [empty]

external STRING
 import CHAR
 sort String
 function
  empty, dot(Char,String),
  append(String,String), deleteLast(String): String
 variable Str1, Str2, Str3, Str4: String
internal STRING
 equation
  append(empty,Str1): Str1
```

```
  append(dot(Chr1,Str1),Str2): dot(Chr1,append(Str1,Str2))
  deleteLast(empty), deleteLast(dot(Char,empty)): empty
  deleteLast(dot(Chr1,dot(Chr2,Str1))):
   dot(Chr1,deleteLast(dot(Chr2,Str1)))

external EVENT
 import NATURAL, CHAR
 sort Event
 function
  openApplication, quit,
  cut, copy, paste, clear,
  enter(Char), return, backSpace,
  select(Natural,Natural), click(Natural): Event
internal EVENT
 equation
  return: enter(newLine)
  click(Na1): select(Na1,Na1)

external DOCUMENT
 import NATURAL, STRING
 sort Document
 function
  document(String,String,String): Document
  fromTo(String,Natural,Natural): Document
 variable Doc1: Document
internal DOCUMENT
 function
  front(String,Natural,Natural), middle(String,Natural,Natural),
   back(String,Natural,Natural): String
 equation
  fromTo(Str1,Na1,Na2):
   document(front(Str1,Na1,Na2),middle(Str1,Na1,Na2),
    back(Str1,Na1,Na2))
  front(empty,Natural,Natural),
  front(String,0,Natural), front(String,Natural,0):
   empty
  front(dot(Chr1,Str1),succ(Na1),succ(Na2)):
   dot(Chr1,front(Str1,Na1,Na2))
  middle(empty,Natural,Natural), middle(String,0,0): empty
  middle(dot(Chr1,Str1),0,succ(Na1)): dot(Chr1,middle(Str1,0,Na1))
  middle(dot(Chr1,Str1),succ(Na1),0): dot(Chr1,middle(Str1,Na1,0))
  middle(dot(Char,Str1),succ(Na1),succ(Na2)): middle(Str1,Na1,Na2)
  back(empty,Natural,Natural): empty
  back(dot(Chr1,Str1),0,0): dot(Chr1,Str1)
  back(dot(Char,Str1),0,succ(Na1)): back(Str1,0,Na1)
```

```
    back(dot(Char,Str1),succ(Na1),0): back(Str1,Na1,0)
    back(dot(Char,Str1),succ(Na1),succ(Na2)): back(Str1,Na1,Na2)

  external STATE
   import DOCUMENT
   sort State
   function
    finder, state(Document,String): State
  internal STATE
   [empty]

  external SCREEN
   import STRING
   sort Selection, Screen
   function
    caret, selection(String): Selection
    deskTop, window(String,Selection,String): Screen
  internal SCREEN
   equation
    selection(empty): caret

  external EDITOR
   import EVENT, STATE, SCREEN
   function
    welcomeToMacintosh, transition(State,Event): State
    show(State): Screen
  internal EDITOR
   equation
    welcomeToMacintosh: finder
    transition(finder,openApplication):
     state(document(empty,empty,empty),empty)
    transition(finder,quit) [should not be possible],
    transition(finder,cut), transition(finder,copy),
    transition(finder,paste), transition(finder,clear),
    transition(finder,enter(Char)), transition(finder,backSpace),
    transition(finder,select(Natural,Natural)):
     finder
    transition(state(Doc1,Str1),openApplication): state(Doc1,Str1)
     [should not be possible]
    transition(state(Document,String),quit): finder
    transition(state(document(Str1,Str2,Str3),String),cut):
     state(document(Str1,empty,Str3),Str2)
    transition(state(document(Str1,Str2,Str3),String),copy):
     state(document(Str1,Str2,Str3),Str2)
    transition(state(document(Str1,String,Str3),Str4),paste):
     state(document(append(Str1,Str4),empty,Str3),Str4)
```

```
transition(state(document(Str1,String,Str3),Str4),clear):
 state(document(Str1,empty,Str3),Str4)
transition(state(document(Str1,String,Str3),Str4),enter(Chr1)):
 state(document(append(Str1,dot(Chr1,empty)),empty,Str3),Str4)
transition(state(document(Str1,dot(Char,String),Str3),Str4),
  backSpace):
 state(document(Str1,empty,Str3),Str4)
transition(state(document(Str1,empty,Str3),Str4),backSpace):
 state(document(deleteLast(Str1),empty,Str3),Str4)
transition(state(document(Str1,Str2,Str3),Str4),
  select(Na1,Na2)):
 state(fromTo(append(append(Str1,Str2),Str3),Na1,Na2),Str4)
show(finder): deskTop
show(state(document(Str1,Str2,Str3),String)):
 window(Str1,selection(Str2),Str3)
```

3.4. Checker

In this section we will describe the Perspect checker that has been developed. We will first describe the operation of the checker in section 3.4.1. After that, an outline of the implementation of the checker will be given in section 3.4.2.

The checker that was implemented does not check Perspect as it has been defined in the Perspect report as included in this thesis as section 3.1. Instead it checks a language that is slightly different. The difference between the 'official' language and the implemented language will be described and motivated in section 3.4.3.

3.4.1. Operation

The Perspect checker is called '`Perspect`'. In its current form it only runs on the Macintosh microcomputer, as a 'tool' under the Macintosh Programmer's Workshop (MPW) environment.

The Perspect checker has a simple interface. While we have not ported the program to any other environment, it should be easy to modify it in order to make it run under any other environment that considers the external world as a TTY, like MS-DOS or UNIX. Care has been taken to isolate the system-dependent parts of the program in two small modules, in order to facilitate a port.

The Perspect checker as it is implemented only forms a partial checker, compared to what it should be according to the Perspect definition. In particular, no checking on filenames and partitioning into files is done. One should think of the Perspect checker as a back-end for a program that performs a complete check of the Perspect definition. In its present form, the Perspect checker considers the specification to be a stream of bytes, and does not check or use the filenames of the constituent files of that stream.

The interface of the Perspect checker is simple. It is invoked by the name of the checker, '`Perspect`', followed by zero or more filenames. Those files are concatenated to a specification which is checked. If no filename is present, the specification is taken from the standard input stream of the program. The checker subsequently checks the specification, and writes any errors it encounters on the standard output stream of the program.

For purpose of keeping the interface of the checker simple, it does not have some of the customary properties of an MPW (or, for that matter, UNIX) tool. For example, the program will not write to the standard error stream. Also, after the program terminates, the exit status of the program (as printed by the MPW command '`Echo {Status}`') will be 0, regardless of the correctness of the specification.

The error messages that the Perspect checker prints all have a uniform format. The general structure of such a message is:

```
##  Message
#   Message [Reference]
    File "filename"; Line linenumber
```

Here 'Message' is some string describing the problem found in the specification. 'Reference' is a reference to the Perspect report. It is a part of the section number of the relevant section in this thesis. In order to get the full section number, it has to be prefixed by the string '`3.1.`', because section 3.1 is the section that contains the Perspect report.

A list of all error messages that the Perspect system can print appears as appendix II of this thesis. There are four classes of these messages. The first class is the class of messages related to the operating system. This class has only one member, which is the complaint:

```
##  Can't open: File filename.
```

The second class contains the messages that say that the text being checked is not a correct Perspect specification as defined in the Perspect report. For example, if a syntax error has been made in the specification, the message:

```
##  Syntax error: Unexpected token. [1.2]
    File "filename"; Line linenumber
```

will be printed, and checking will terminate. The third class of messages are prefixed by the string '`Implementation restriction`'. These errors are caused by the way the program has been structured, or by the limitations of the machine the checker is running on. A representative example of this class is:

```
##  Implementation restriction: Memory full.
```

The fourth, and last, class of messages printed by the Perspect checker are prefixed

by the string '`Internal error`'. If such a message is printed, it means that there is an error in the implementation of the Perspect checker. These messages are used for debugging and act as runtime verification of a number of assertions on the behavior of the program. A representative example of this fourth class is:

```
##  Internal error: Dangling pointer.
```

The input of the checker consists of the concatenation of all the files that are mentioned in the command line. However, this means that a subtle problem can occur when files are present in which the last line is not terminated by a carriage return character. The reason for this is that the file boundary does not act as a token separator.

For instance, suppose that we have two files `A.per` and `B.per`. The file `A.per` contains the string:

'`external A internal A`'

(without a trailing carriage return or newline). The file `B.per` contains the string:

'`external B internal B`'

Now because of the way the Perspect checker gets its input, it will take the concatenation of these strings to be the specification that should be checked. Because of this, the stream that it reads is:

'`external A internal Aexternal B internal B`'

Therefore, the command '`Perspect A.per B.per`' will print the following error messages:

```
##  Module identifiers don't match: Aexternal should be A. [3.1]
    File "B.per"; Line 1
##  Syntax error: Unexpected identifier B. [1.2]
    File "B.per"; Line 1
```

(In the UNIX world this problem is not likely to show, because in that environment it is hard to create a text file that does not terminate with a newline character. However, on the Macintosh, particularly under MPW, it is easier to create a file that does not end with a line separator than it is to create one that does.)

The Perspect checker contains one message that is not an error message. This message will not be printed without modifying the code of the program (in fact, the statements that print it are only present as a comment in the source). This message prints the set NFA as defined in 2.5.3.1 (To be honest: this is not quite what it prints. The checker uses a version of algorithm 2.5.4.2 that uses the modular structure of the specification in order to be more efficient. This means that the set that is printed

resembles NFA, but may be different.) It has the format:

```
##   sort identifier
#       term
#       term
#       ...
```

If this message is enabled, it will be printed for all sorts in the specification. It has primarily been used to study the behavior of the persistence check algorithm that is present in the checker. However, it might be useful in its own right.

3.4.2. Program

The Perspect checker has been written in Modula-2, and has been compiled with the TML Modula-2 compiler (version 1.0), which is a single-pass compiler that runs under the MPW environment as a MPW tool.

The checker consists of the following fourteen modules: `StdAlloc`, `StdInOut`, `Allocation`, `Input`, `Output`, `Messages`, `Idents`, `ParserTypes`, `Attributes`, `Scanner`, `Parser`, `Modules`, `Symbols` and `Perspect`. The definition modules of these modules are included in this thesis as appendix III.

The last module, `Perspect`, is the main program that invokes the checker. It just calls the parse routine `Parser.yyparse`, which invokes the rest of the checker while parsing. The complete text of this main module is:

```
MODULE Perspect;
 IMPORT Attributes;
 IMPORT Parser;
 VAR
  result: Attributes.YYSTYPE;
BEGIN
 IF Parser.yyparse(result) THEN
 END
END Perspect.
```

The other modules in the checker can be classified into six groups.

The first group consists of modules `StdAlloc` and `StdInOut`. These are just copies of the parts of the TML library modules that are used by the checker. They give some low-level memory allocation and block I/O functionality. The functions from these two modules are easily emulated in other environments. For instance, all calls from these modules correspond to standard UNIX functions. Apart from these two modules, the checker is completely platform independent.

The second group consists of modules `Allocation`, `Input` and `Output`. These are the higher level counterparts of the modules from the first group. They basically add buffering for a higher performance. The implementation of `Allocation` is a straightforward Modula-2 translation of the BSD 4.3 implementation of `malloc`. The routines

from `Input` and `Output` perform conversion from integers to strings.

The third group consists of modules `Messages` and `Idents`. Here the identifier table is implemented. Also, a general facility for producing error messages is provided. If a different format for error messages might be needed, only module `Messages` need to be changed.

The fourth group consists of modules `Attributes`, `Scanner` and `Parser`. These contain the scanner and parser used by the checker. The scanner is a straightforward, manually written program, because of the simple lexical syntax of Perspect. The parser was generated by a Modula-2 version of `yacc`, which consists of a simple `lex` program and an `ex` script that translate the C code that comes out of `yacc` to Modula-2. The structure of the code of this Modula-2 version of `yacc` is as follows:

module	*definition module*	*implementation module*
`Attributes`	manually written	empty
`Scanner`	fixed	manually written
`Parser`	fixed	generated

The definition module of module `Attributes` should define only one type, called `YYSTYPE`. It is the type of values passed on `yacc`'s parse stack. It should have the following format:

```
TYPE
 YYSTYPE =
  RECORD
   CASE: INTEGER OF
   0: ... |
   1: ... |
   ...
   ELSE
   END
  END;
```

The definition modules of modules `Scanner` and `Parser` are fixed. They are:

```
DEFINITION MODULE Scanner;
 IMPORT Attributes;
 PROCEDURE yylex
  (VAR token: INTEGER; VAR value: Attributes.YYSTYPE);
 PROCEDURE yyerror (token: INTEGER; value: Attributes.YYSTYPE);
 PROCEDURE yyabort;
END Scanner.

DEFINITION MODULE Parser;
 IMPORT Attributes;
 PROCEDURE yyparse (VAR yyresult: Attributes.YYSTYPE): BOOLEAN;
```

```
END Parser.
```

It will be clear that the attributes passed around by yacc are more visible in this Modula-2 scheme because of the stronger typing of Modula-2 as compared to C. The functions `yyerror` and `yyabort` are called in case an error occurs. The former is called when a syntax error is detected; the latter when `yacc`'s parse stack (which has a fixed size) overflows.

The Modula-2 version of the parser driver is a literal translation of the `yacc` driver under 4.3 BSD. Some work had to be done to eliminate the various `gotos` from this code. The size of this code is modest as is shown by the following line counts. The input file for `yacc` consisted of 252 lines of code. The template file, used by `yacc` and containing the parser driver, consisted of 182 lines of code. Finally, the generated implementation module of module `Parser` consisted of 808 lines of which 431 lines (each one containing one assignment statement) consisted of initialization code for the various tables used by the (table driven) parser.

The fifth group of modules in the checker consists of modules `ParserTypes` and `Modules`. The sixth group is the single module `Symbols`. The functions defined in the modules in the fifth group are called from `yyparse`, and call in their turn the functions in module `Symbols`. In a sense, these two last groups implement the same functionality: they both build a data structure reflecting the structure of the specification. The similarities between these two parts of the program are striking (for example, both have a datatype of 'lists of terms': `TermList` from module `ParserTypes` and `Tuple` from module `Symbols`).

A difference between these two parts of the checker is that the functions from module `Modules` have to take possible erroneous input into account. However, if the input turns out to be incorrect, the routines from `Symbols` are simply not called. This means that Symbols has always a consistent picture of the world. This makes it easier to implement the various 'semantic' checks in module `Symbols`, because they will always have a correct data structure to operate on.

The data structures in `ParserTypes` are opaque. They can only be operated on by means of iterator functions that are provided in the interface of this module. This is a clean programming style, but it gives some notational overhead. Therefore, the data structures in module `Symbols` are simply pointers to records. This representation is visible to the program, and the only way to access the information present in the data is by dereferencing pointers and selecting fields from records. Because the data structures have no clean interface and the type definition of the various types is not included in the definition module of module `Symbols`, all routines that operate on them have been made part of module `Symbols`.

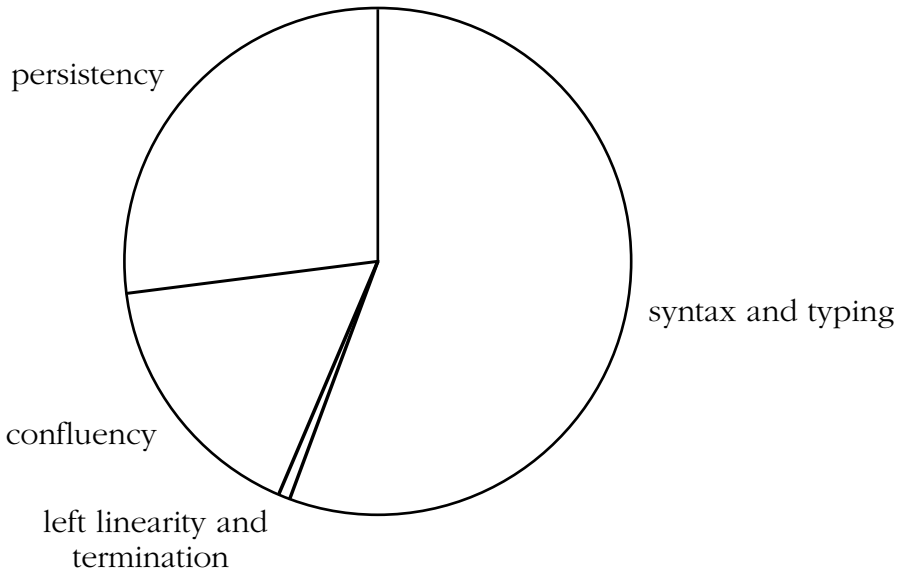
This is the reason that module `Symbols` is very large. Its size is approximately half of that of the program. It would have been cleaner to separate module `Symbols` in a half that creates the data structures and a half that operates on them. However, because both halves have to operate very intensively on the same data structure, this would have meant a lot of notational overhead.

Module `Symbols` also contains all the 'semantic' checks that have to be performed on a Perspect specification. These checks correspond to the requirements of section

3.1.5, and are all straightforward implementations of the algorithms described in chapter 2. In particular, the persistence check is performed following algorithm 2.5.4.2.

The total program consists of 4789 lines of Modula-2 code. The implementation module of module `Symbols` consists of 2792 lines. In this module 1127 lines are used for persistence analysis. Of those lines, 439 are general purpose term manipulation routines and associated garbage collection routines (a simple reference counting scheme is used); 448 are used for the routines that manipulate sets of terms and lists of sets of terms (which are specifically needed for the persistence check); 120 are needed for an implementation of algorithm 2.5.3.2 which is the one that calculates the set called NFA. Finally, 120 lines perform the persistence check proper.

The analysis of the persistence check in section 2.5.4.3, where it was shown that this algorithm uses at least exponential space in the worst case, gave a rather negative picture of its efficiency. Experimentation with the implementation of this algorithm gives a more positive view. The following pie chart indicates the relative amount of time spent on the various tasks in a sample run of Perspect, during which, among others, all the specifications in section 3.3 were checked.



3.4.3. Deviation from the formal definition

The language recognized by the Perspect checker differs slightly from the language that was defined in section 3.1. The latter language is easier to describe, the former is easier to implement (and more efficiently). The difference between these two languages lies in the way the confluence of the specification is verified. This requirement was described in section 3.1.5.3.

There are two properties that one would like the Perspect language to possess. One of these properties is satisfied by the defined language, the other is satisfied by the implemented language.

The first of these properties of the language is that correctness of a specification should be retained under normalization. The *normalized* form of a specification is obtained by removing all modular structure from the specification. (One has to rename hidden objects in this 'flat' specification, in order to remove potential name clashes). Also, the order of the functions in the normalized specification should be chosen in such a way that it corresponds to the order on the functions implied by the original specification. Now, if we take a Perspect specification, it would be desirable if the normalized form was also a correct Perspect specification. This property is satisfied by the language as it was defined in 3.1, but not by the language for which the checker was implemented.

The second desirable language property is that correctness of a specification should be retained under module addition. If `M1` and `M2` are the names of modules in a Perspect specification, the *sum* of `M1` and `M2` is defined as the same specification with module `M3` added at the end:

```
external M3 [= M1 + M2]
 import M1, M2
internal M3
 [empty]
```

In general adding the sum of two modules to a correct specification will again give a correct specification when no violations of the origin rule – name clashes – occur in the extended specification. However, this is not always true in the language as defined in 3.1. It *is* true in the language as recognized by the checker.

If one wants to implement the requirement of section 3.1.5.3, one has to check for overlaps in all pairs of equations occurring in modules that are relevant at one point of the specification. This means that if we take a sum of two modules, we have to check all equations that are relevant to one of those modules for potential overlaps. So, the checking of a sum according to the defined language can involve a lot of pairs of equations. In the implementation only pairs of equations are considered in which one of the equations in the pair is present in the module being checked. In this case, a sum does not lead to any new pairs to be checked.

While the implementation deviates from the definition of Perspect, it checks a language, (one could call it Perspect′) that also satisfies all properties that were described in section 3.2. This follows from the fact that a specification that the checker accepts is still confluent (though not *open* confluent).

In order to be more concrete, let us consider an example. We start by considering the following specification which is correct both according to the language definition and according to the checker:

```
external M
 sort A
 function
  a, fg(A), gh(A), f(A), g(A), h(A): A
 variable x: A
internal M
```

```
 equation
  fg(a), gh(a), f(a), g(a), h(a): a

external M1
 import M
internal M1
 equation
  f(g(x)): fg(x)

external M2
 import M
internal M2
 equation
  g(h(x)): gh(x)
```

Now, if we add the sum `M3` of `M1` and `M2` to this specification, it will no longer be correct according to the definition of the language, though the checker will not complain.

The reason this 'addition' of modules in the specification destroys the correctness of the specification is that the equation in module `M1` and the one in module `M2` together are not open confluent. This will become apparent when we consider the normalized form of the extended specification.

```
external M123
 sort A
 function
  a, fg(A), gh(A), f(A), g(A), h(A): A
 variable x: A
internal M123
 equation
  fg(a), gh(a), f(a), g(a), h(a): a
  f(g(x)): fg(x)
  g(h(x)): gh(x)
```

This specification is not correct, neither according to the definition of the language nor according to the checker. This can be seen by studying the output of the checker:

```
##  Module M123 is not open confluent.
#     f(g(h(x))):
#       fg(h(x))
#       f(gh(x))
#   Maybe the equations
#     f(g(x)): fg(x)
#     g(h(x)): gh(x)
#   should be more specific. [5.3]
```

So, the problem is that the term `f(g(h(x)))` both rewrites to `fg(h(x))` and to `f(gh(x))`, which have no common (open) normal form. This defies open confluence.

Note that while the specification of `M123` is both incorrect according to the definition and the implementation of Perspect, it still is a confluent term rewrite system.

It would have been nice if we had found a definition of a Perspect like language (call it Perspect″) that satisfies the following three requirements:

- The language definition is simple.
- Correctness of a text in the language is retained under normalization.
- Correctness of a text in the language is retained when adding sums to the specification.

We have not found a definition that implies confluence, and that satisfies these three requirements simultaneously.

Chapter 4

Compilation

How many Prolog programmers does it take to change a lightbulb?
no

In this chapter we will describe two schemes for compiling left linear complete term rewriting systems to conventional programs. In section 4.1 we will study compilation to Prolog, and in section 4.2 we will study compilation to Modula-2.

For both schemes the complete translation of a sample specification is given. In the Prolog section, the small text editor from the previous chapter is translated to a Prolog program that manages to capture the 'look and feel' of the specified program remarkably well. In the Modula-2 section it is shown that it is possible to use the machine representation of low level objects (like, in the example, the integers) in a nice transparent way.

The main advantage of these schemes is their conceptual simplicity; they were not designed with efficiency in mind. The simplicity of the Prolog scheme comes from the comprehensibility of the predicates in the translation. The simplicity of the Modula-2 scheme lies in the way it becomes possible to interface handwritten implementations to automatically generated ones. This is the reason why a program compiled using the Modula-2 scheme (while not very efficient as generated) can be made as efficient as desired by replacing low level modules by handcrafted ones.

The main statement of this chapter – the fact that it is possible to implement a term rewriting system as a conventional program – is trivial. However, the way the implementation is done might still be interesting.

4.1. Prolog

In this section we will describe a scheme for compiling complete term rewriting systems to Prolog. Because Prolog is not a modular language, it does not retain the modular structure of the original specification. The scheme is constructed in such a way that nine different variants result from the choice of two options. Each variant implements a different reduction strategy. Among these strategies are leftmost innermost reduction, leftmost outermost reduction, parallel outermost reduction and Gross-Knuth reduction.

4.1.1. Compilation scheme

4.1.1.1. Various relations. We start our description of the compilation scheme to Prolog by defining a number of relations between the closed terms that are associated with a term rewriting system. For these relations we use a notation that can also be used in Prolog programs.

The first relation is the rewriting relation. We represent this relation in this section by the symbol `--->` (a legal Prolog symbol) instead of the symbol $\rightarrow$ that was used until now. The reduction relation means that one step of the rewriting system has been executed, i.e., that one subredex of the term has been reduced one step. This relation is derived from the rewriting rules by instantiating the variables in both sides of a rule by closed terms, and by placing the resulting terms in an identical context.

From the rewriting relation `--->` we can derive four other relations by taking various closures. They are:

`=`	take zero reduction steps, syntactical identity on closed terms
`-=->`	take zero reduction steps or one reduction step, the reflexive closure of the relation `--->`
`--->>`	take zero or more reduction steps, the reflexive and transitive closure of the relation `--->`
`->->>`	take one or more reduction steps, the transitive closure of `--->`

As is shown in the following diagram, this is a natural set of relations derived from `--->`:

number of reductions	≤ 0	≤ 1	$\leq \infty$
≥ 0	`=`	`-=->`	`--->>`
≥ 1		`--->`	`->->>`

The use of Prolog syntax for the mathematical relation of term reduction may be a source of confusion. For that reason, we give a list of the various meanings the symbols `--->` and `--->>` take on throughout the description of the compilation scheme.

The symbols `--->` and `--->>` will occur:

- As abstract relations between closed terms, as in the previous paragraphs. These are the customary relations from the theory of term rewriting, only written using Prolog syntax.
- In the Prolog programs below, read as a logic programs. For `--->>` this interpretation corresponds exactly to the previous one, i.e., the expression that states that s `--->>` t is true will be provable if and only if s `--->>` t is indeed true. (This will also be the case when t is not in normal form, in contrast to the next interpretation.) The relation `--->` does not correspond exactly to the abstract notion of taking one reduction step. However, if taking one step in the reduction strategy

means taking only one reduction step, it *does* correspond exactly to the mathematical interpretation of `--->`.

- In the Prolog programs, read as Prolog programs. We now have an interpretation as Prolog predicates with one input and one output parameter. Again the predicate `--->` does not mean 'take one reduction step', but 'take one step in the current reduction strategy'. This may mean that more than one reduction step is taken. The meaning of `--->>` is now 'normalize'. This implies that the statement that s `--->>` t implies that t is the reduct of s, so now t is automatically in normal form.

4.1.1.2. A simple example. We will now show how to translate a term rewriting system to Prolog. While our scheme has a number of choices, we will first give only one specific instance of it, which implements leftmost outermost reduction.

We could now describe the translation *in abstracto*. However, for clarity we will just give a simple example. The example that we will use is addition on the natural numbers:

```
external Naturals
 sort NAT
 function 0, succ(NAT), add(NAT,NAT): NAT
internal Naturals
 variable X1, X2: NAT
 equation
  add(0,X2): X2
  add(succ(X1),X2): succ(add(X1,X2))
```

The translation of this specification consists of a description of the definitions of the abstract rewriting relations `--->` and `--->>` in Prolog.

```
/* part 1: independent of the specific term rewriting system */
:- op(700, xfx, [--->,--->>]).
X--->>Z :- X--->Y, Y--->>Z.
X--->>X.

/* part 2: for every rewriting rule one Prolog rule */
add(0,X2)--->X2.
add(succ(X1),X2)--->succ(add(X1,X2)).

/* part 3, the context rules: for every n-adic function symbol n Prolog rules */
succ(X1)--->succ(Y1)      :- X1--->Y1.
add(X1,X2)--->add(Y1,Y2) :- X1--->Y1, X2 =  Y2.
add(X1,X2)--->add(Y1,Y2) :- X1 =  Y1, X2--->Y2.
```

This Prolog program has two related but different meanings. The first interpretation is as a logic program. The second as a Prolog program. We will first consider the meaning as a 'pure' logic program.

It is easy to see that with this interpretation the following property holds:

∀ closed terms s and t:
the program proves s `--->>` t ⇔ s `--->>` t is a true statement

This equivalence is a more formal restatement of the fact that we have given the definition of `--->>` in Prolog syntax. It is not only true for this version of the translation scheme, but also for the various variants that will occur below.

The other interpretation of the program is as a Prolog program. It is easy to see that when we ask the program:

```
?- s--->>N.
```

for some closed term s, that Prolog will answer with:

`N =` *the result of leftmost outermost reduction of s*

provided that leftmost outermost reduction of s terminates with a normal form. If leftmost outermost reduction of s does not terminate, execution of the query will not terminate either (or more precisely, in practice it will probably terminate by running out of memory).

In order to be more explicit, we will give a concrete example. The input of the user is in italics. The query:

```
?- add(succ(succ(succ(0))),succ(succ(succ(succ(0)))))--->>N.
```

will give the output:

```
N = succ(succ(succ(succ(succ(succ(succ(0)))))))
```

which is the expected normal form.

4.1.1.3. The general scheme. The scheme that was used in the example from the previous section can be generalized in a relatively natural way. To do this we first have to make a number of =-relations explicit, and after that consider some of the `=` and `--->` relations as 'parameters'. Different reduction strategies can be realized by taking different relations as 'values' of those parameters. Once again we will demonstrate the form of our scheme by implementing the example of the natural numbers.

The Prolog program that now follows is almost identical to that in the previous section. However, some of the `=` symbols have been replaced by $=_1$, some of the `=` symbols by $=_2$ and some of the `--->` symbols by $--->_2$.

```
/* part 1: independent of the specific term rewriting system */
:- op(700, xfx, [-=->,--->,--->>,->->>]).
X-=->Y :- X--->Y.
X-=->X.
```

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

/* part 2: for every rewriting rule one Prolog rule */
add(0,X2)--->Y2                          :- X2 =1 Y2.
add(succ(X1),X2)--->succ(add(Y1,Y2)) :- X1 =1 Y1, X2 =1 Y2.

/* part 3, the context rules: for every n-adic function symbol n Prolog rules */
succ(X1)--->succ(Y1)      :- X1--->2Y1.
add(X1,X2)--->add(Y1,Y2) :- X1--->2Y1, X2 =2  Y2.
add(X1,X2)--->add(Y1,Y2) :- X1 =   Y1, X2--->2Y2.
```

The structure of the context rules is as follows. For each function symbol, there is a $--->_2$ on the diagonal, a $=_2$ to the upper right of the diagonal and a $=$ to the lower left of the diagonal.

(By also using $=_2$ to the lower left of the diagonal one does not gain anything. If the rule could have been executed because of this replacement, one of the earlier rules (the one with the $--->_2$ in the same place) already would have been applicable.)

Depending on the relations that are substituted for $=_1$, $=_2$ and $--->_2$ one gets a program implementing a different reduction strategy. Most of the well known reduction strategies can be obtained in this way.

In the table below it is shown which substitution corresponds to which strategy. The empty places in the diagram correspond to strategies that are not very natural, so they do not have a name of their own. The relation of the upper-right corner to the implementation scheme by van Emden and Yukawa will be explained in section 4.1.2.1.

The only normalizing strategies in the table are parallel outermost reduction and Gross-Knuth reduction. The only cofinal strategy in the table is Gross-Knuth reduction.

	$=_2$:	=	-=->	--->>
	$--->_2$:	--->	--->	->->>
$=_1$:	=	leftmost outermost	parallel outermost	van Emden-Yukawa
	-=->		Gross-Knuth	
	--->>			leftmost innermost

We will not give a formal proof of the fact that this table is correct. To see that the correspondence between Prolog programs and reduction strategies is as this table

says, it is useful to think of the `--->` operator as a procedure that takes its input in its first argument, and uses its second argument to output the term after taking one step according to the reduction strategy. In the same way the `--->>` operator can be seen as a procedure that tries to normalize its first argument according to the reduction strategy.

4.1.1.4. Optimizations. The scheme from section 4.1.1.3 has a number of interesting variations.

It is easy to see that when the `--->` condition at the right hand side of a context rule (in part 3 of the program) is met, further backtracking is superfluous (in all variants!). Still, Prolog will continue backtracking. To avoid this, it makes sense to add *cuts* to the program, in the following way:

```
add(X1,X2)--->add(Y1,Y2) :- X1--->Y1, !, X2 =  Y2, !.
add(X1,X2)--->add(Y1,Y2) :- X1 =  Y1,    X2--->Y2, !.
```

A second optimization consists of explicitly indicating the *typing* from the specification in the generated Prolog program. Because the system now only has to look at rewriting rules of the right kind, this produces a program that is more efficient. The example of the natural numbers could for instance be transformed to (admittedly, the typing in this example is a bit boring):

```
/* part 1: for every sort two Prolog rules */
:- op(700, xfx, [--->,--->>]).
nat(X--->>Z) :- nat(X--->Y), nat(Y--->>Z).
nat(X--->>X).

/* part 2: for every rewriting rule one Prolog rule */
nat(add(0,X2)--->X2).
nat(add(succ(X1),X2)--->succ(add(X1,X2))).

/* part 3, the context rules: for every n-adic function symbol n Prolog rules */
nat(succ(X1)--->succ(Y1))      :- nat(X1--->Y1).
nat(add(X1,X2)--->add(Y1,Y2)) :- nat(X1--->Y1),    X2 =  Y2.
nat(add(X1,X2)--->add(Y1,Y2)) :-     X1 =  Y1, nat(X2--->Y2).
```

It would be even better to represent the combination `nat(X--->Y)` by just one operator.

A third optimization, not in execution time, but in the size of the generated program (giving a smaller but slower program) is the following. Instead of writing down n Prolog rules for each n-adic function symbol occurring in the specification, the following three rules can be used, which are independent of the specification that is being translated:

```
T ---> S :- T=..[F|X], append(Z,[X1|X2],X),
  X1--->Y1, parallel(X2,Y2), append(Z,[Y1|Y2],Y), S=..[F|Y].
```

```
parallel([],[]).
parallel([X1|X2],[Y1|Y2]) :- X1=Y1, parallel(X2,Y2).
```

(While `parallel` seems to be equivalent to `=`, which it is, in other variants of the compilation scheme it is not.)

These rules can be seen as an 'interpreter' that simulates the context rules. It is also possible to give the third part of the Prolog program a 'compiler' that puts the context rules using `assertz` in the Prolog database. In order to be able to do that, one has to give a list of all function symbols occurring in the specification.

4.1.2. Comparison with other schemes

The compilation scheme from the previous section will now be compared to two well known schemes from the literature. Those implementations can be reformulated to resemble our scheme surprisingly closely.

4.1.2.1. Van Emden & Yukawa. In [van Emden, Yukawa, 1986] two Prolog programs are associated with a term rewriting system. These programs are constructed according to the so called 'interpretational' and 'compilational' approaches. The second of these approaches does not resemble the scheme described here; it is more related to the approach from section 4.2 (the Modula-2 scheme) formulated for Prolog. However, the 'interpretational' approach closely resembles our scheme.

Van Emden and Yukawa do not write Prolog. Their programs are *logic programs* in an abstract sense. Because of this, what they call '=' is not the Prolog unification operator, but the equality relation as defined by the equations (rules) of the specification. So, in [van Emden, Yukawa, 1986] a rule like:

```
eq2(X,Y) :- X=Y.
```

should be read as:

```
eq2(X,Y) :- rule(X,Y).
rule(⟨left hand side of first rewriting rule⟩,⟨right hand side of first rule⟩).
rule(⟨left hand side of second rule⟩,⟨right hand side of second rule⟩).
...
```

Bearing this in mind, we obtain the following 'interpretational' translation of the example specification of the natural numbers with addition (the predicate `canonical` that tests whether a term is in normal form is defined in a different part of the program, and its definition is omitted here):

```
eq1(X,X) :- canonical(X).
eq1(X,Z) :- not canonical(X), eq2(X,Y), eq1(Y,Z).
```

```
eq2(X,Y) :- rule(X,Y).
rule(add(0,X2),X2).
rule(add(succ(X1),X2),succ(add(X1,X2))).

eq2(succ(X1),succ(Y1))      :- eq1(X1,Y1).
eq2(add(X1,X2),add(Y1,Y2)) :- eq1(X1,Y1), eq1(X2,Y2).
```

Now eliminate the predicate called `rule` from the program, and rename:

```
eq1 →   --->>
eq2 →   --->
```

If we do this we get:

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

X--->>X :- canonical(X).
X--->>Z :- not canonical(X), X--->Y, Y--->>Z.

add(0,X2)--->X2.
add(succ(X1),X2)--->succ(add(X1,X2)).

succ(X1)--->succ(Y1) :- X1--->>Y1.
add(X1,X2)--->add(Y1,Y2) :- X1--->>Y1, X2--->>Y2.
```

This implementation gives the reduction strategy from the cell in the upper right corner of the diagram in section 4.1.1.3, labeled 'van Emden-Yukawa'. The main difference between our implementation of this reduction strategy and van Emden & Yukawa's, which is caused by the fact that they do not have an equivalent of the `->->>` relation, is that their solution does not require at least one rewriting step to be taken in the last two rules. This is why they need the predicate called `canonical`.

4.1.2.2. Drosten & Ehrich. Before we describe the implementation scheme of Drosten & Ehrich, we will first generalize our own scheme.

The implementation scheme that we give here consists of nine different variants. There is a way to add further variants. The Prolog rules that define the operator `--->` in the program consist of two parts, the rewriting rules (which were called 'part 2' in our scheme) and the context rules ('part 3'). If we exchange these two parts we get another group of nine implementations. Only three of them are distinct: it does not matter any more which variant of $=_1$ is chosen. All three variants implement a kind of innermost reduction, a strategy that was already present in the original scheme.

There is a problem with these new implementations however: If the context rules from part 3 come before those from part 2, it is never possible to apply two context rules directly in succession. Because the scheme is innermost, after the first context rule all proper subterms of the term that is being reduced will be in normal form, so

trying a second context rule does not make sense. However, Prolog will attempt this anyway. The program is correct but not optimal. At the end of this section we will show how the program can be slightly modified in order to solve this efficiency problem.

We will now turn to Drosten & Ehrich's implementation scheme as described in [Drosten, Ehrich, 1984]. According to their scheme the example becomes:

```
analyse(0,Z) :- normalize(0,Z).
analyse(succ(X1),Z) :- analyse(X1,Y1), normalize(succ(Y1),Z).
analyse(add(X1,X2),Z)
 :- analyse(X1,Y1), analyse(X2,Y2), normalize(add(Y1,Y2),Z).

rule(add(0,X2),X2).
rule(add(succ(X1),X2),succ(add(X1,X2))).

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

We are going to show that this resembles our implementation by restructuring the program and by renaming the predicates in it. First, we split the definition of `analyse` in two parts:

```
analyse(X,Z) :- analyse1(X,Y), normalize(Y,Z)

analyse1(0,0).
analyse1(succ(X1),succ(Y1)) :- analyse(X1,Y1).
analyse1(add(X1,X2),add(Y1,Y2)) :- analyse(X1,Y1), analyse(X2,Y2).
```

Now, replace in this program:

```
rule        →   -#->
analyse1    →   -##->
analyse     →   -##->>
normalize   →   -#->>
```

Then we get:

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

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

add(0,X2)-#->X2.
add(succ(X1),X2)-#->succ(add(X1,X2)).
```

```
0-##->0.
succ(I1)-##->succ(K1) :- I1-##->>K1.
add(I1,I2)-##->add(K1,K2) :- I1-##->>K1, I2-##->>K2.
```

This closely resembles our 'modified' scheme (in which part 2 and 3 have been exchanged). The difference is that `--->` and `--->>` each have two variants called `-#->`, `-##->` respectively `-#->>`, `-##->>`.

This difference between the 'modified' scheme and the Drosten & Ehrich scheme solves the problem of the application of two context rules (which are called `-##->` here) in succession. When we look at the definition of the normalization operator `-##->>`, it is easy to verify that the program will not try two `-##->` rules in succession.

So the Drosten & Ehrich scheme solves the problem with our modified scheme!

4.1.3. Example

In order to show the 'feel' of a prototype that can be built using a Prolog translation of a specification, we will now study an example. We will give the leftmost outermost translation of the specification of the small text editor from section 3.3.7.

4.1.3.1. The Prolog translation. We first give the result of translating the specification according to the scheme from section 4.1.1.3. Because we want to use the resulting Prolog code to create a prototype that acts like the specified editor, some *cosmetic* renamings have been applied in order to make it easier to interact with the final program.

The first change to the literal translation is a renaming that causes Prolog to use the built in Prolog lists for the specification's 'lists'. In order to accomplish this, we have to apply two renamings:

```
dot      →   '.'
empty    →   []
```

The operator `'.'` is a built in operator of the Prolog system, and does not need to be declared.

Now, a list that should be written according to the specification as:

```
dot(p,dot(e,dot(r,empty)))
```

is represented by:

```
'.'(p,'.'(e,'.'(r,[])))
```

which is equivalent to:

```
[p,e,r]
```

This last representation is the one that the Prolog system will use to output the term. It is clear that this is a much more readable form than the original one.

The other renaming that we have applied to the translation of the specification concerns the representation of the natural numbers. Suppose that, while editing, we want to select (using the mouse) the 43-th character of the text. In the original specification the event that accomplishes this is designated:

```
select(addDigit(4,2),addDigit(4,3))
```

Replacing the function `addDigit` with the operator `^`, makes it possible to write:

```
select(4^2,4^3)
```

which is again more readable. The operator is defined to be of type `yfx`. This causes numbers containing more than two digits to behave as expected. The expression:

```
6^6^6
```

will be interpreted as:

```
(6^6)^6
```

and so will evaluate (as is clearly the intention) to 666.

The program as it is given here does not *do* anything yet: it only defines the predicates `--->` and `--->>`. In the subsequent sections we give some additional code that makes *use* of these predicates in order to obtain a prototype that a user can interact with.

```
/* part 1: independent of the specific term rewriting system */
:- op(700, xfx, [--->,--->>]).
X--->>Z :- X--->Y, Y--->>Z.
X--->>X.

/* the function addDigit is replaced by the operator ^ for convenience */
:- op(300, yfx, [^]).

/* part 2: for every rewriting rule one Prolog rule */
1--->succ(0). 2--->succ(1). 3--->succ(2).
4--->succ(3). 5--->succ(4). 6--->succ(5).
7--->succ(6). 8--->succ(7). 9--->succ(8).
'^'(0,Na1)--->Na1.
'^'(succ(Na1),Na2)--->
 '^'(Na1,succ(succ(succ(succ(succ(
         succ(succ(succ(succ(succ(Na2))))))))))).
```

```
append([],Str1)--->Str1.
append('.'(Chr1,Str1),Str2)--->'.'(Chr1,append(Str1,Str2)).
deleteLast([])--->[].
deleteLast('.'(Char,[]))--->[].
deleteLast('.'(Chr1,'.'(Chr2,Str1)))--->
 '.'(Chr1,deleteLast('.'(Chr2,Str1))).
return--->enter(newLine).
click(Na1)--->select(Na1,Na1).
fromTo(Str1,Na1,Na2)--->
 document(front(Str1,Na1,Na2),middle(Str1,Na1,Na2),
  back(Str1,Na1,Na2)).
front([],Natural,Natural)--->[].
front(String,0,Natural)--->[].
front(String,Natural,0)--->[].
front('.'(Chr1,Str1),succ(Na1),succ(Na2))--->
 '.'(Chr1,front(Str1,Na1,Na2)).
middle([],Natural,Natural)--->[].
middle(String,0,0)--->[].
middle('.'(Chr1,Str1),0,succ(Na1))--->
 '.'(Chr1,middle(Str1,0,Na1)).
middle('.'(Chr1,Str1),succ(Na1),0)--->
 '.'(Chr1,middle(Str1,Na1,0)).
middle('.'(Char,Str1),succ(Na1),succ(Na2))--->
 middle(Str1,Na1,Na2).
back([],Natural,Natural)--->[].
back('.'(Chr1,Str1),0,0)--->'.'(Chr1,Str1).
back('.'(Char,Str1),0,succ(Na1))--->back(Str1,0,Na1).
back('.'(Char,Str1),succ(Na1),0)--->back(Str1,Na1,0).
back('.'(Char,Str1),succ(Na1),succ(Na2))--->back(Str1,Na1,Na2).
selection([])--->caret.
welcomeToMacintosh--->finder.
transition(finder,openApplication)--->
 state(document([],[],[]),[]).
transition(finder,quit)--->finder.
transition(finder,cut)--->finder.
transition(finder,copy)--->finder.
transition(finder,paste)--->finder.
transition(finder,clear)--->finder.
transition(finder,enter(Char))--->finder.
transition(finder,backSpace)--->finder.
transition(finder,select(Natural,Natural))--->finder.
transition(state(Doc1,Str1),openApplication)--->
 state(Doc1,Str1).
transition(state(Document,String),quit)--->finder.
transition(state(document(Str1,Str2,Str3),String),cut)--->
```

```
 state(document(Str1,[],Str3),Str2).
transition(state(document(Str1,Str2,Str3),String),copy)--->
 state(document(Str1,Str2,Str3),Str2).
transition(state(document(Str1,String,Str3),Str4),paste)--->
 state(document(append(Str1,Str4),[],Str3),Str4).
transition(state(document(Str1,String,Str3),Str4),clear)--->
 state(document(Str1,[],Str3),Str4).
transition(state(document(Str1,String,Str3),Str4),enter(Chr1))--->
 state(document(append(Str1,'.'(Chr1,[])),[],Str3),Str4).
transition(state(document(Str1,'.'(Char,String),Str3),Str4),
  backSpace)--->
 state(document(Str1,[],Str3),Str4).
transition(state(document(Str1,[],Str3),Str4),backSpace)--->
 state(document(deleteLast(Str1),[],Str3),Str4).
transition(state(document(Str1,Str2,Str3),Str4),
  select(Na1,Na2))--->
 state(fromTo(append(append(Str1,Str2),Str3),Na1,Na2),Str4).
show(finder)--->deskTop.
show(state(document(Str1,Str2,Str3),String))--->
 window(Str1,selection(Str2),Str3).
```

/ part 3, the context rules: for every n-adic function symbol n Prolog rules */*

```
succ(Na1)--->succ(Na2) :- Na1--->Na2.
'^'(Na1,Na2)--->'^'(Na3,Na4) :- Na1--->Na3, Na2=Na4.
'^'(Na1,Na2)--->'^'(Na3,Na4) :- Na1=Na3, Na2--->Na4.
'.'(Chr1,Str1)--->'.'(Chr2,Str2) :- Chr1--->Chr2, Str1=Str2.
'.'(Chr1,Str1)--->'.'(Chr2,Str2) :- Chr1=Chr2, Str1--->Str2.
append(Str1,Str2)--->append(Str3,Str4)
 :- Str1--->Str3, Str2=Str4.
append(Str1,Str2)--->append(Str3,Str4)
 :- Str1=Str3, Str2--->Str4.
deleteLast(Str1)--->deleteLast(Str2) :- Str1--->Str2.
enter(Chr1)--->enter(Chr2) :- Chr1--->Chr2.
click(Na1)--->click(Na2) :- Na1--->Na2.
select(Na1,Na2)--->select(Na3,Na4) :- Na1--->Na3, Na2=Na4.
select(Na1,Na2)--->select(Na3,Na4) :- Na1=Na3, Na2--->Na4.
document(Str1,Str2,Str3)--->document(Str4,Str5,Str6)
 :- Str1--->Str4, Str2=Str5, Str3=Str6.
document(Str1,Str2,Str3)--->document(Str4,Str5,Str6)
 :- Str1=Str4, Str2--->Str5, Str3=Str6.
document(Str1,Str2,Str3)--->document(Str4,Str5,Str6)
 :- Str1=Str4, Str2=Str5, Str3--->Str6.
fromTo(Str1,Na1,Na2)--->fromTo(Str2,Na3,Na4)
 :- Str1--->Str2, Na1=Na3, Na2=Na4.
fromTo(Str1,Na1,Na2)--->fromTo(Str2,Na3,Na4)
```

```
 :- Str1=Str2, Na1--->Na3, Na2=Na4.
fromTo(Str1,Na1,Na2)--->fromTo(Str2,Na3,Na4)
 :- Str1=Str2, Na1=Na3, Na2--->Na4.
front(Str1,Na1,Na2)--->front(Str2,Na3,Na4)
 :- Str1--->Str2, Na1=Na3, Na2=Na4.
front(Str1,Na1,Na2)--->front(Str2,Na3,Na4)
 :- Str1=Str2, Na1--->Na3, Na2=Na4.
front(Str1,Na1,Na2)--->front(Str2,Na3,Na4)
 :- Str1=Str2, Na1=Na3, Na2--->Na4.
middle(Str1,Na1,Na2)--->middle(Str2,Na3,Na4)
 :- Str1--->Str2, Na1=Na3, Na2=Na4.
middle(Str1,Na1,Na2)--->middle(Str2,Na3,Na4)
 :- Str1=Str2, Na1--->Na3, Na2=Na4.
middle(Str1,Na1,Na2)--->middle(Str2,Na3,Na4)
 :- Str1=Str2, Na1=Na3, Na2--->Na4.
back(Str1,Na1,Na2)--->back(Str2,Na3,Na4)
 :- Str1--->Str2, Na1=Na3, Na2=Na4.
back(Str1,Na1,Na2)--->back(Str2,Na3,Na4)
 :- Str1=Str2, Na1--->Na3, Na2=Na4.
back(Str1,Na1,Na2)--->back(Str2,Na3,Na4)
 :- Str1=Str2, Na1=Na3, Na2--->Na4.
state(Doc1,Str1)--->state(Doc2,Str2)
 :- Doc1--->Doc2, Str1=Str2.
state(Doc1,Str1)--->state(Doc2,Str2)
 :- Doc1=Doc2, Str1--->Str2.
selection(Str1)--->selection(Str2) :- Str1--->Str2.
window(Str1,Sel1,Str2)--->window(Str3,Sel2,Str4)
 :- Str1--->Str3, Sel1=Sel2, Str2=Str4.
window(Str1,Sel1,Str2)--->window(Str3,Sel2,Str4)
 :- Str1=Str3, Sel1--->Sel2, Str2=Str4.
window(Str1,Sel1,Str2)--->window(Str3,Sel2,Str4)
 :- Str1=Str3, Sel1=Sel2, Str2--->Str4.
transition(State1,Event1)--->transition(State2,Event2)
 :- State1--->State2, Event1=Event2.
transition(State1,Event1)--->transition(State2,Event2)
 :- State1=State2, Event1--->Event2.
show(State1)--->show(State2) :- State1--->State2.
```

4.1.3.2. Checking the user input. The program that was given in the previous section only works correctly when it is provided with correctly typed terms. However, the program that the user sees has to handle errors in the user input gracefully (for example by printing '`?`' or maybe even '`error`'). So, the final program has to know what terms represent correctly typed objects of the type that is expected from the user.

In this case, the user will input terms of type `Event`. Therefore the following Prolog lines define the predicate `event` that succeeds if and only if its only argument is an

correctly typed object of type `Event`. Because the sort `Event` is derived from the sorts `Natural` and `Char`, typechecking code for these sorts also has to be present.

The rules for typechecking a term are straightforward. However, to avoid having to write a complete enumeration of all constants of type `Natural` and `Char`, the check whether an atom is a digit or a letter is performed with the aid of the Prolog predicate called `name`. This predicate returns a list of the ASCII codes of the characters in the name of an atom.

```
event(openApplication).
event(quit).
event(cut).
event(copy).
event(paste).
event(clear).
event(enter(Chr1)) :- char(Chr1).
event(return).
event(backSpace).
event(click(Na1)) :- natural(Na1).
event(select(Na1,Na2)) :- natural(Na1), natural(Na2).
natural(Na1) :- name(Na1,[N]), 48=<N, N=<57.
natural(succ(Na1)) :- natural(Na1).
natural('^'(Na1,Na2)) :- natural(Na1), natural(Na2).
char(Chr1) :- name(Chr1,[N]), 97=<N, N=<122.
char(space).
char(newLine).
```

4.1.3.3. The driver. Finally, we give the driver of the small prototype that we are describing here. If one concatenates the Prolog fragments in sections 4.1.3.1, 4.1.3.2 and 4.1.3.3, one gets a program that simulates an edit session. It asks the user for objects of type `Event`, and prints the current object of type `Screen` after each input.

This is a standard Prolog program, which primarily invokes `--->>` in a loop. The two clauses that contain `--->>` are:

```
show(State1)--->>Screen1
```

and:

```
transition(State1,Event1)--->>State2
```

The first clause converts the current `State` to a `Screen` that can be intelligibly printed. The second clause performs a `State` transition according to the user input.

The loop has been written using tail recursion. Cuts have been added to aid a clever Prolog implementation to prevent running out of space after a long edit session.

A 'fake' `Event` called `shutDown` has been added for the driver. This `Event` never reaches the implementation that was generated from the specification, but instead

tells the driver to stop.

The main predicate that starts the simulation is simply called `run`. If during the simulation no incorrect `Events` have been input by the user, it will succeed; else it will fail.

```
run :- run(welcomeToMacintosh).
run(State1)
 :- show(State1)--->>Screen1,
  write(Screen1), nl,
  read(Event1), !,
  continue(State1,Screen1,Event1).
continue(State1,Screen1,Event1)
 :- Screen1 = deskTop, Event1 = shutDown, !.
continue(State1,Screen1,Event1)
 :- test(State1,Event1),
  transition(State1,Event1)--->>State2, !,
  run(State2).
test(_,Event1) :- event(Event1), !.
test(State1,Event1)
 :- write(error), nl, !,
  run(State1),
  fail.
```

4.1.3.4. A sample run. Below a representative run of the program is given. The input from the user is printed in italics. The user in this session types one line containing the word *asf*, and then edits it in order to modify it to *sdf*.

The look and feel of this simulation is remarkably close to that of a real Macintosh editor. While, of course, typing '`select(0,1)`' differs from selecting the first letter of the text using a mouse, one gets a good impression of how the program behaves.

```
?- run.
deskTop
|: openApplication.
window([],caret,[])
|: enter(a).
window([a],caret,[])
|: enter(s).
window([a,s],caret,[])
|: enter(f).
window([a,s,f],caret,[])
|: return.
window([a,s,f,newLine],caret,[])
|: backSpace.
window([a,s,f],caret,[])
|: select(0,1).
```

```
window([],selection([a]),[s,f])
|: cut.
window([],caret,[s,f])
|: click(1).
window([s],caret,[f])
|: enter(d).
window([s,d],caret,[f])
|: click(3).
window([s,d,f],caret,[])
|: quit.
deskTop
|: shutDown.
```

4.2. Modula-2

We will now study the concept of implementing an algebraic specification in some modular procedural programming language. To be more specific, we will use the language Modula-2 [Wirth, 1985] in the examples. However, the same concepts can be applied to similar languages like C or Ada.

In the previous section, we gave an explicit scheme for the implementation of a term rewriting system in Prolog. In this section we will be slightly more general: before giving a scheme for implementing a class of algebraic specifications in Modula-2, we will first define the *notion* of implementing an algebra. This means that we not only will give one specific implementation, but also make explicit what it means to be an implementation.

One nice property of this implementation notion, is that it is *modular*. This means that the modules in the implementation correspond in a one-to-one fashion to the modules in the specification.

An advantage of this is that this makes it possible to build *heterogeneous* implementations: implementations in which some modules are automatically compiled, and in which other modules are implemented by hand. An implementation of some module that is generated by the compilation scheme from this section is guaranteed to work correctly when the other modules in the specification are correctly implemented. Those other implementations do not have to be generated according to the same scheme.

We will not define just one notion of implementing an algebra in a procedural language, but two: a strong and a weak one. A strong implementation is only possible for algebras in which the word problem is decidable. This means that some algebras do not have a strong implementation. In contrast with this, each algebra has a weak implementation. Our compilation scheme is defined for left linear complete term rewriting systems and it produces a strong implementation.

Each algebra has a trivial weak implementation. This follows from the observation that a weak implementation of some algebra is also a weak implementation of each quotient of that algebra. This means that an implementation of the term algebra always

is a weak implementation, though not a very useful one. This trivial implementation is not sufficient when we demand in addition that for some of the modules in the specification (e.g. the Booleans, the integers and the strings) an implementation will be used that has been given in advance. Even then, if the specification is persistent, it is still guaranteed to have a weak implementation.

4.2.1. Compilation scheme

4.2.1.1. Algebras versus programs. The notion of an algebra and that of a module from a modular program are similar. This means that we will have to be careful in our nomenclature, if it is to be clear what we are referring to. In this section we will follow the convention that an algebra consists of sorts and functions while a module from a program, which we will call a program for short, has datatypes and procedures.

Of course, there are a number of differences between the notion of an algebra and that of a program. The most important difference is that a program has a *state* which the procedures can modify as a side effect. This means that an executing program is not static but changes in time.

A more philosophical difference concerns the existence and identity of objects. An object in an algebra is a mathematical object which exists and is unique. In a running program, there are, at any point in time, only finitely many objects in existence. Furthermore, some objects can be present in more than one copy.

A third difference between an algebra and a program is that a program operates in a finite world, which implies that it has limited resources. This means that a program cannot operate on arbitrarily large objects. The limits imposed on a program, however, are not given beforehand. So the behavior of a program is dependent on the environment that it is being run in. This shows that a program should *not* be compared to a finite algebra. In a finite algebra, it is known in advance what the limits of the algebra are. It is better to compare one run of a program to a finite subset of an infinite algebra, which need not to be closed under the functions of the algebra (i.e., which need not be a subalgebra).

A difference between algebras and programs that is closely related to the previous one is that a procedure in a program can *fail*, either by running for an infinitely long time, or by external intervention, such as running out of resources or an interrupt from the outside world. As an example, suppose that I call a procedure in order to add two numbers. While this procedure is running, I turn the computer off. Now, how does this correspond to the addition function in the algebra from which I took the numbers?

4.2.1.2. Compiling the signature. Suppose that I have an algebra consisting of some sorts σ and some functions f. In an implementation of this algebra, as will be defined in this section, there exists a datatype for each sort in the algebra, and *two* procedures for each function in the algebra. These datatypes and procedures should all be different. Also, these should be all datatypes and procedures that occur in the implementation.

In this section we will use the convention that the datatype has the same name as

the sort it corresponds to. The first kind of procedure has the same name as the function it corresponds to, while the name of the second kind of procedure (which behaves somewhat as an inverse to that of the first kind) is obtained by prefixing the name of the function by the string 'is'.

Of course, this way of naming can lead to name clashes. We will choose our examples in such a way that these clashes do not occur. In a practical system some deviation of our scheme for naming datatypes and procedures will be necessary. This practicality is not important for the rest of this description, so it will not be treated here.

A related problem is that some names in the signature of the algebra may be illegal in the program because they are illegal in the language that is used, or because they correspond to some reserved word. This problem can be treated similarly to the handling of name clashes, and will also be ignored.

For an example, consider the following signature (in Perspect syntax):

```
external Naturals
 sort Nat
 function Zero, Succ(Nat), Add(Nat,Nat): Nat
```

The following Modula-2 'signature' (which Modula-2 calls a 'definition module') corresponds to this signature :

```
DEFINITION MODULE Naturals;
 TYPE Nat;
 PROCEDURE Zero(): Nat;
 PROCEDURE Succ(n: Nat): Nat;
 PROCEDURE Add(n1: Nat; n2: Nat): Nat;
 PROCEDURE isZero(m: Nat): BOOLEAN;
 PROCEDURE isSucc(m: Nat; VAR n: Nat): BOOLEAN;
 PROCEDURE isAdd(m: Nat; VAR n1: Nat; VAR n2: Nat): BOOLEAN;
END Naturals.
```

In this example it can be seen how the arguments of the procedures in the implementation should be typed. If f is a function in the algebra that has domain $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n$ and range σ, then the procedure with the name 'f' has arguments with types named σ_1, σ_2, …, σ_n while it returns a value from the datatype named σ. The function named 'isf' has one argument with type named σ, and n **VAR**-arguments with types named σ_1, σ_2, …, σ_n while it returns a BOOLEAN.

Clearly, the datatype BOOLEAN has a special status in the implementation scheme that we are defining here. This does *not* mean that the algebra that is being implemented should contain a sort that corresponds to the Booleans. Also, it does *not* mean that if there are one or more sorts in the specification that correspond to the Booleans, that these sorts should or could be implemented by the 'native' Booleans, i.e., the Modula-2 datatype called BOOLEAN.

The **VAR**-parameters that occur in the procedures called isf are used to return

multiple values from the procedure. The reason for this is twofold. On the one hand, Modula-2 has no other easy mechanism for returning multiple values. On the other hand, it does only make sense to return values in the **VAR**-parameters in the case that the procedure returns the constant TRUE. If the constant FALSE is returned, the **VAR**-parameters can be left alone. This option would not have been present when multiple values would have been returned as the result of the procedure.

4.2.1.3. Proper objects. Suppose we have a program that has a definition module as described in the previous section. We then distinguish between two kind of objects in the datatypes in such a program. An object that can be created by calling the various procedures in the module we call *proper* objects. The other objects we call *improper*.

In a sense the datatype that consists of the proper objects of a program is analogous to the minimal subalgebra of an algebra. However, datatypes in a program can only be specified by the primitive constructions that are built into the programming language, so in general it will not be possible to create a datatype that has only proper objects.

As an example, consider the following Perspect specification:

```
external Booleans
 sort Bool
 function True, False: Bool
internal Booleans
 [empty]
```

and the following Modula-2 program that gives a (strong) implementation of this specification:

```
DEFINITION MODULE Booleans;
 TYPE Bool;
 PROCEDURE True(): Bool;
 PROCEDURE False(): Bool;
 PROCEDURE isTrue(b: Bool): BOOLEAN;
 PROCEDURE isFalse(b: Bool): BOOLEAN;
END Booleans.

IMPLEMENTATION MODULE Booleans;
 TYPE Bool = INTEGER;

 PROCEDURE True(): Bool;
 BEGIN
  RETURN -1
 END True;

 PROCEDURE False(): Bool;
```

```
    BEGIN
     RETURN 0
    END False;

    PROCEDURE isTrue(b: Bool): BOOLEAN;
    BEGIN
     RETURN b < 0
       (* 'the high bit is set' *)
    END isTrue;

    PROCEDURE isFalse(b: Bool): BOOLEAN;
    BEGIN
     RETURN b MOD 2 = 0
       (* 'the low bit is cleared' *)
    END isFalse;

  END Booleans.
```

(A restriction of Modula-2 has been ignored here, which means that this is not a correct Modula-2 program on all systems. In Modula-2 an opaque datatype, i.e., a datatype that is not defined in the definition module, must take exactly the same storage as a pointer. An object of type `INTEGER` does not always satisfy this restriction. To get a working program one may need to replace the `INTEGER` datatype by a datatype like `LONGINT` or `ADDRESS`. The general compilation scheme that we will describe below does not suffer from this problem, because there all opaque datatypes will turn out to be pointers.)

In this program the only proper objects of datatype `Bool` are the integers -1 and 0. All other objects of type `Bool`, the integers less than -1 and those greater that 0, are improper. Note that the program does not need to behave sensible when given improper objects (it even may abort if it likes to). For example `isTrue(-2)` and `isFalse(-2)` both evaluate to `TRUE`, although we have an implementation of an algebra in which `True` ≠ `False`.

4.2.1.4. Functionality. This section describes an implementation scheme for modular *procedural* languages. However, from now on we will only allow programs that behave *functionally*. By this we mean that when we repeat a program fragment a number of times under different circumstances, all `BOOLEAN`s returned by functions of the form '`isf`' should be identical.

One kind of side effect is unavoidable: the consumption of memory by the program. In this respect repetition of a program can lead to different behavior. So we will have to weaken our requirement a bit. It should have been: 'if the program would be run with an infinite memory', it should behave functionally. For a discussion of how a program could handle a situation in which it cannot obtain enough memory, see below in section 4.2.1.10.

From now on, this restriction, which should hold for both strong and weak implementations, should be taken for granted and will not be repeated.

4.2.1.5. Weak implementation. We are now going to define when we will call a program a *weak implementation* of some algebra. The intuitive meaning of this is that the objects in the datatypes of the program are 'really' terms over the signature of the algebra.

In order for a program to be a weak implementation of an algebra, it has to be possible to associate with each proper object (as defined in section 4.2.1.3) from the datatypes in the program a specific term over the signature of the algebra, such that a number of requirements are satisfied. These requirements are the following (the various typing restrictions that are necessary for these requirements to be meaningful are omitted for clarity):

(i) Let x_1, x_2, …, x_n be proper objects from the program with associated terms t_1, t_2, …, t_n. Further, let f be some function from the algebra. If the procedure named 'f' is called with arguments x_1, x_2, …, x_n it should terminate after a finite duration, and return some object x with associated term t. This term t should evaluate to the same object in the algebra as the term $f(t_1, t_2, \ldots, t_n)$.

(ii) Let x be a proper object from the program, with associated term $t \equiv f(t_1, t_2, \ldots, t_n)$. Further, let f be some function from the algebra. If we call the procedure named '`isf`' with x as first parameter it should terminate after a finite duration. If the outermost function symbol of t *is* equal to f, then it should return `TRUE`; else it should return `FALSE`. If the procedure returns `TRUE`, the **`VAR`**-parameters of the procedure should be set to objects x_1, x_2, …, x_n, whose associated terms are *identical* to the arguments of t, i.e., t_1, t_2, …, t_n, (in contrast to the situation in requirement (i) is *not* sufficient that those terms evaluate to the same object in the algebra; they should be identical). If the procedure returns `FALSE`, the **`VAR`**-parameters should be left unmodified. Note that for each proper object x exactly one function '`isf`' returns `TRUE` when called with x as first argument, while all other functions return `FALSE`.

From (ii) it follows that the collection of terms associated with the objects in the program should be closed under taking subterms.

Some consideration will show that a weak implementation of an algebra is also a weak implementation of each quotient of that algebra. Also, a weak implementation of an algebra is also a weak implementation of each subalgebra of the algebra.

From this, it is easy to see how to construct a trivial implementation of an arbitrary algebra. Just implement the term algebra over the signature of the algebra. The implementation that one gets in this way is called the *free* weak implementation of the algebra.

4.2.1.6. Strong implementation. To define the concept of a *strong implementation*, we have to add one requirement. The collection of all terms that are associated with an object in the program has to satisfy the following constraint: For each object in the algebra, there should be exactly one term from this collection (i.e., that occurs as a term associated with some object in the program), such that that term evaluates to that object.

From this it clearly follows that only minimal algebras have a strong implementa-

tion. It is also true that only those algebras have a strong implementation, in which equality of terms is decidable. This is slightly less trivial, but will be left to the reader.

The converse is also true: Each minimal algebra in which equality of terms over that algebra is decidable has a strong implementation.

4.2.1.7. Term rewriting implementation. If an algebra is given by a semi-complete term rewriting system, there is an even more restrictive notion: that of a *term rewriting implementation* of the term rewriting system. The extra requirement here is that only the normal forms of the rewriting system should occur as term associated with an object in the program.

All implementations of some fixed term rewriting system behave identically. Therefore, we will also speak of *the* term rewriting implementation of a term rewriting system.

4.2.1.8. Compilation. We will now outline an explicit scheme for transforming a left linear complete term rewriting system to a term rewriting implementation. A complete, but simple example of this compilation scheme will be presented in section 4.2.2. It is probably easiest to first take a look at this example before reading the abstract description of the compilation scheme in this section.

We will implement the objects in the various datatypes as pointers to variant records. Let σ be some sort in the term rewriting system. Then implement the datatype called 'σ' as a pointer to a variant record, that has a variant for each function f with range σ. This variant has one field for each of the arguments of function f. (Only the functions that occur in normal forms need to have a variant in the record. Perspect gives one enough information to deduce automatically what functions need be included.)

Now the implementation of the procedures called '`isf`' is simple. Just check whether we have the variant corresponding to function f, and if so, copy all fields in the record to the **`VAR`**-parameters.

The implementation of the procedures corresponding to the functions in the signature of the rewriting system is slightly more involved. Suppose procedure 'f' is called with arguments x_1, x_2, …, x_n. The terms corresponding to the x_i are then necessarily in normal form. The procedure then first tries to find out whether the term $f(x_1, x_2, \ldots, x_n)$ matches one of the rewriting rules in the rewriting system. If so, then the procedures corresponding to the right hand side of such a rule is called, and the result of that expression is returned. If not, a new object corresponding to $f(x_1, x_2, \ldots, x_n)$ is created, and returned.

To check whether an object corresponds to a term that matches an open term (a left hand side of a rewriting rule), the object is 'unpacked' by applying procedures of the form '`isf`'. In this way the left hand side of the rule is traversed in a depth-first left-first manner. It is undoubtedly possible to describe this procedure 'in abstracto' here. It is also probable that just referring to the example in section 4.2.2 will be much clearer.

4.2.1.9. Left linearity. The restriction of left linearity in section 4.2.1.8 makes that it is not necessary to add routines to decide whether two objects correspond to the same normal form. This is not problematic but because our example will only illustrate the left linear case, and because the Perspect framework that we use here implies left linearity, this will not be made explicit.

To give an impression of the form of a procedure for checking equality, we give here the routine for the sort `Nat` from the example in the next section:

```
PROCEDURE eqNat(n1: Nat; n2: Nat): BOOLEAN;
VAR n11, n12, n21, n22: Nat;
BEGIN
 IF isZero(n1) THEN
  RETURN isZero(n2)
 END;
 IF isOne(n1) THEN
  RETURN isOne(n2)
 END;
 IF isAdd(n1, n11, n12) THEN
  RETURN
   isAdd(n2, n21, n22) AND eqNat(n11, n21) AND eqNat(n12, n22)
 END;
 HALT
END eqNat;
```

4.2.1.10. Memory management. There are two issues in these 'compiled specifications' that concern memory management.

First, there is the question what should happen when there is not enough memory left to create a new object. One could simply abort, which is not very graceful. Alternatively, one could have a variable pointing to a user-defined procedure that should allocate the memory to hold the new object. If this solution is chosen, the user probably will do the aborting for us.

A similar issue occurs when one uses a manual implementation instead of the generated one, that has some built-in limitation. For example, suppose we implement the natural numbers by the built-in datatype `INTEGER`. Now, what should happen on an overflow? If we ignore it (the solution most programming languages take), we have an incorrect implementation. Again, having a user installable function pointing to a routine that should 'take care of the problem' for us seems the cleanest solution.

Second, a language like Modula-2 has no built-in facilities for garbage collection. This means that once an object has been created, it will not be disposed of automatically when it is no longer accessible by the program. This is not very nice, and will make the problem of running out of memory worse.

There exists a nice scheme in which for each sort, two procedures are added, characterized by the prefixes '`duplicate`' and '`dispose`'. These procedures respectively add and remove links to objects. It is possible to integrate these procedures with the procedures from our scheme. This is necessary, as the following example shows. Sup-

pose we first create the number 0 by calling:

```
z := Zero()
```

and then create the number 1 by calling:

```
sz := Succ(z)
```

Now, after removing the '0' by calling:

```
disposeNat(z)
```

we do not want to find that the '1' is damaged by this operation. Clearly, procedure `Succ` should call `duplicateNat`.

Procedures like `duplicateNat` and `disposeNat` can conveniently be implemented with a reference counting scheme. However, when for instance the natural numbers are implemented as `INTEGER`s, these procedures will obviously be inert.

It is well known that reference counting schemes give a large overhead on the running time of a program. An alternative to the `duplicate` & `dispose` scheme could be the garbage collection scheme from [Böhm, Weiser, 1988]. This scheme does not need any support from the compiler or the user program, so it is clearly applicable here. It is not clear whether it executes more efficiently than the reference counting scheme, though.

4.2.2. Example

In this section we will give an explicit example of the compilation scheme from the previous section. Because we believe that the example will be much clearer than an abstract description, the description of the scheme in the previous section was rather sketchy.

In sections 4.2.1.9 and 4.2.1.10 some additions to the compilation scheme were described: procedures for testing equality, failing gracefully and memory management. To keep the example simple, these additions will not be incorporated here.

4.2.2.1. Specification. Our example consists of two modules called `Naturals` and `Multiplication`. The specification consisting of these modules is correct Perspect, so together they form a left linear complete term rewriting system: suitable for compilation to Modula-2.

The specification is:

```
external Naturals
 sort Nat
 function Zero, One, Add(Nat,*Nat): Nat
 variable n, n1, n2, n3: Nat
```

```
internal Naturals
 equation
  Add(Zero,n), Add(n,Zero): n
  Add(n1,Add(n2,n3)): Add(Add(n1,n2),n3)

external Multiplication
 import Naturals
 function Mul(Nat,Nat): Nat
internal Multiplication
 equation
  Mul(Zero,Nat): Zero
  Mul(One,n): n
  Mul(Add(n1,n2),n3): Add(Mul(n1,n3),Mul(n2,n3))
```

The idea behind this example is that we suppose we have a machine that has addition 'built in', but on which multiplication is a 'defined operation'. This motivates the way our specification has been separated in two modules.

We will first give the Modula-2 program that we get when we apply the compilation algorithm from the previous section. After that we will replace the implementation of the 'low level' module by a manual implementation, to show that this does not pose any problems.

4.2.2.2. Generated implementation. We will now give the implementation that one gets when one just applies the compilation algorithm. Because the compilation algorithm has not been actually implemented, the compilation was done 'by hand', which means that the compiled code looks 'too good'. In an automatic system, the various variables would probably have had longer and more distracting names.

First we show the definition part of an auxiliary module that gives the runtime support needed by the implementation. This module is called '`Allocation`', and contains the procedure used to allocate memory. The implementation part of this module is rather long, not very relevant and therefore omitted.

```
DEFINITION MODULE Allocation;
 FROM SYSTEM IMPORT ADDRESS;
 PROCEDURE NEW(size: INTEGER): ADDRESS;
END Allocation.
```

We will now give the definition modules that have been generated according to the scheme of section 4.2.1.2.

The '`FROM Naturals IMPORT Nat`' clause in the Modula-2 program corresponds directly to the '`import Naturals`' clause in the specification. In a sense the definition modules in the Modula-2 implementation are identical to the external parts of the Perspect modules. One can imagine an integrated specification/implementation language (which should probably be called 'Specula') in which each module has *three* parts: an interface part (external part/definition module), a specification part and an

implementation part. The advantages of this integration between specification and implementation language seem marginal because of the high level of similarity that is already present between current specification and implementation languages.

```
DEFINITION MODULE Naturals;
 TYPE Nat;
 PROCEDURE Zero(): Nat;
 PROCEDURE One(): Nat;
 PROCEDURE Add(n1: Nat; n2: Nat): Nat;
 PROCEDURE isZero(m: Nat): BOOLEAN;
 PROCEDURE isOne(m: Nat): BOOLEAN;
 PROCEDURE isAdd(m: Nat; VAR n1: Nat; VAR n2: Nat): BOOLEAN;
END Naturals.

DEFINITION MODULE Multiplication;
 FROM Naturals IMPORT Nat;
 PROCEDURE Mul(n1: Nat; n2: Nat): Nat;
END Multiplication.
```

We now give the implementation part of module Naturals. It is admittedly a bit repetitive. We hope that this repetitiveness makes the algorithm that produced the compiled specification more clear.

```
IMPLEMENTATION MODULE Naturals;

 FROM SYSTEM IMPORT ADDRESS;
 FROM Allocation IMPORT NEW;

 TYPE
  NatKind = (ZeroKind, OneKind, AddKind);
  NatRecord =
   RECORD
    CASE kind: NatKind OF
    ZeroKind: |
    OneKind: |
    AddKind:
     n1: Nat;
     n2: Nat
    END
   END;
  Nat = POINTER TO NatRecord;

 PROCEDURE Zero(): Nat;
 VAR m: Nat;
 BEGIN
  m := NEW(SIZE(NatRecord));
```

```
 m^.kind := ZeroKind;
 RETURN m
END Zero;

PROCEDURE One(): Nat;
VAR m: Nat;
BEGIN
 m := NEW(SIZE(NatRecord));
 m^.kind := OneKind;
 RETURN m
END One;

PROCEDURE Add(n1: Nat; n2: Nat): Nat;
VAR n21, n22, m: Nat;
BEGIN
(* Add(Zero,n2): n2 *)
 IF isZero(n1) THEN
  RETURN n2
 END;
(* Add(n1,Zero): n1 *)
 IF isZero(n2) THEN
  RETURN n1
 END;
(* Add(n1,Add(n21,n22)): Add(Add(n1,n21),n22) *)
 IF isAdd(n2, n21, n22) THEN
  RETURN Add(Add(n1, n21), n22)
 END;
(* normal form *)
 m := NEW(SIZE(NatRecord));
 m^.kind := AddKind; m^.n1 := n1; m^.n2 := n2;
 RETURN m
END Add;

PROCEDURE isZero(m: Nat): BOOLEAN;
 BEGIN
  IF m^.kind = ZeroKind THEN
   RETURN TRUE
  END;
  RETURN FALSE
 END isZero;

PROCEDURE isOne(m: Nat): BOOLEAN;
 BEGIN
  IF m^.kind = OneKind THEN
   RETURN TRUE
```

```
    END;
    RETURN FALSE
   END isOne;

  PROCEDURE isAdd(m: Nat; VAR n1: Nat; VAR n2: Nat): BOOLEAN;
   BEGIN
    IF m^.kind = AddKind THEN
     n1 := m^.n1;
     n2 := m^.n2;
     RETURN TRUE
    END;
    RETURN FALSE
   END isAdd;

END Naturals.
```

The implementation of module `Multiplication` is completely analogous to that of module `Naturals`. Note that there is no procedure called `isMul`. The reason for this is that the range of function `Mul`, which is of course the sort `Nat`, is a sort imported from an other module than the one in which `Mul` has been introduced.

```
IMPLEMENTATION MODULE Multiplication;

 FROM SYSTEM IMPORT ADDRESS;
 FROM Allocation IMPORT NEW;

 FROM Naturals IMPORT Nat, Zero, One, Add, isZero, isOne, isAdd;

 PROCEDURE Mul(n1: Nat; n2: Nat): Nat;
 VAR n11, n12: Nat;
 BEGIN
 (* Mul(Zero,n2): Zero *)
  IF isZero(n1) THEN
   RETURN Zero()
  END;
 (* Mul(One,n2): n2 *)
  IF isOne(n1) THEN
   RETURN n2
  END;
 (* Mul(Add(n11,n12),n2): Add(Mul(n11,n2),Mul(n12,n2)) *)
  IF isAdd(n1, n11, n12) THEN
   RETURN Add(Mul(n11, n2), Mul(n12, n2))
  END;
 (* impossible because of persistence *)
  HALT
```

```
 END Mul;

END Multiplication.
```

4.2.2.3. Interfacing to a hand written implementation. The implementation of module `Naturals` is inefficient because it represents the natural numbers in a unary way (using approximately 20 bytes per unary digit). We are now going to rewrite module `Naturals` so that it will use the native datatype `INTEGER` of Modula-2. The only part of our program that has to be changed is the implementation part of module `Naturals`. The definition part of module `Naturals` and the two parts of module `Multiplication` are completely unaffected by this action.

(Once again we sweep the possible inappropriateness of using `INTEGER` instead of `LONGINT` or `ADDRESS` under the rug (conform the remark in 4.2.1.3))

```
IMPLEMENTATION MODULE Naturals;

 TYPE Nat = INTEGER;

 PROCEDURE Zero(): Nat;
 BEGIN
  RETURN 0
 END Zero;

 PROCEDURE One(): Nat;
 BEGIN
  RETURN 1
 END One;

 PROCEDURE Add(n1: Nat; n2: Nat): Nat;
 BEGIN
  RETURN n1 + n2
 END Add;

 PROCEDURE isZero(m: Nat): BOOLEAN;
  BEGIN
   RETURN m = 0
  END isZero;

 PROCEDURE isOne(m: Nat): BOOLEAN;
  BEGIN
   RETURN m = 1
  END isOne;

 PROCEDURE isAdd(m: Nat; VAR n1: Nat; VAR n2: Nat): BOOLEAN;
  BEGIN
```

```
    IF m > 1 THEN
     n1 := m - 1;
     n2 := 1;
     RETURN TRUE
    END;
    RETURN FALSE
   END isAdd;

END Naturals.
```

This implementation turns out to be nice. First, the implementation part of module `Naturals` becomes very simple. For example, the implementation of procedure '`isZero`' turns out to be '`= 0`'.

Second, when we trace what happens when we call function `Mul`, it turns out that it performs a repeated addition, which is not bad at all. The only overhead that remains is that every time a multiplication by one has to be 'detected'. One could try to reduce this (slight) inefficiency by introducing the equation:

```
Mul(Add(n1,One),n3): Add(Mul(n1,n3),n3)
```

in front of the equations in the specification of module `Multiplication`. This would introduce the fragment:

```
(* Mul(Add(n11,One),n2): Add(Mul(n11,n2),n2) *)
 IF isAdd(n1, n11, n12) AND isOne(n12) THEN
  RETURN Add(Mul(n11, n2), n2)
 END;
```

in procedure `Mul`. An inlining compiler could then short circuit the sequence '`isAdd` → `n12 = 1` → `isOne`', and after that eliminate it. This would result in the implementation of multiplication by a loop iterating an addition.

Chapter 5

Conclusion

Flon's law: You can write bad software in any language.

In this thesis we have presented four results.

First, we have described a way to mechanically verify persistence of specifications that satisfy a number of requirements. Second, we have defined a framework, Perspect, for operating this approach in practice. Third, we gave a scheme for compiling Perspect specifications to Prolog. Fourth, we have described an explicit framework for implementing specifications in a procedural language, and presented a scheme for compiling Perspect specifications in a modular fashion to Modula-2, that fits in this framework.

We will now describe for each result separately, what appears to us to be its main strengths and weaknesses.

5.1. The persistence check. The main merit of the persistence check algorithm from section 2.5 is that it exists at all. It is surprising that it is decidable whether two left-linear term rewriting systems with the same signature have the same sets of normal forms. It is useful to have a test for missing cases in specifications that use definition by cases. It is also nice that the algorithm is able to determine the generators of a sort without outside help.

The algorithm has two disadvantages. It only applies to specifications that are 'really' executable term rewriting systems. This is a small class of specifications compared to the whole universe of specifications that one would like to write. The other disadvantage of the check algorithm is that it may use a lot of memory. This problem becomes apparent when the specification that is checked is incorrect.

5.2. Perspect. Perspect, as described in sections 3.1 and 3.2, can be seen as a set of requirements on specifications that together have some nice properties. Its main merit is to show that it is possible to satisfy these properties (decidability, executability) with simple requirements. Apart from the way Perspect handles termination, the requirements that Perspect poses on a specification are in a sense *canonical*, i.e., they are natural properties to require of a persistent complete modular term rewriting system. On the other hand, although a termination ordering exists that could be called canonical, the one used in Perspect is much simpler, and in practice it is almost as

good as the canonical, strongest, termination ordering.

An advantage of Perspect is that it is a *language*. It is not a 'persistence checking laboratory' in which the user might be confused by the interface of the program, but instead it is explicit in what it does and what the user should comply with. Also, a manually written text in a for humans readable language seems to be a good way to formulate a specification; better than some intermediate state in an interaction with an interactive program, or some set of computer generated files.

A third advantage of Perspect is that it is *small*. Perspect avoids the complications that can be caused by, possibly unfavorable, interactions between the features in a language. The keywords **`echo`** and **`rec`**, and the '`*`' marker aside (which are made necessary by the termination checking component of the Perspect system), it is in some sense the minimal modular specification language.

The main disadvantage of Perspect is that it is too restrictive. For example, one might be tempted to write a specification of an algebra with a finite sort in Perspect, but experience has shown that this is not something to be undertaken lightly. Also, Perspect does not catch all errors in a specification, as again experience has shown.

5.3. Compilation to Prolog. The scheme for compiling term rewriting systems to Prolog that was developed in section 4.1 was not intended to be used in real applications. Prolog is a high level language and is therefore not appropriate as a target for a compilation. The reason for giving an implementation of a specification in Prolog was mainly the simplicity of the scheme.

The attractiveness of the compilation scheme to Prolog lies in its simplicity. It consists of a literal translation to Prolog syntax of the definition of equality in an equational specification. Also, it is simpler than two other well-known schemes from the literature.

It is also nice that one can combine various rewriting strategies in one uniform scheme.

5.4. Implementing a specification in Modula-2. While the implementation notion as defined in section 4.2 is trivial, it is also useful. If one wants to transform a specification into a working program, and not only wants to use the specification as a 'guideline' for the structure of the resulting program (why should one then take the effort to write a formal specification?), this scheme merits consideration.

A disadvantage of the way we instantiated the implementation scheme is that it is an *innermost* reducing implementation. It would probably be better to use some form of graph reduction in the generated code. While further research could indicate an elegant way for incorporating this in the system as described here, we did not pursue this topic further.

When using this scheme for obtaining an implementation of a significant specification it is advisable to write the specification as an arbitrary specification (so not restricted to the class of term rewriting systems) and then write the implementation manually as a strong implementation as defined in section 4.2.

5.5. Final conclusion. From our work with modular initial algebra specifications we got the following impression. Algebraic specifications are nice because they have a simple semantics and have a great range of applicability: both the low level parts as well as the high level parts of a system can be described algebraically. They are cumbersome because specifying systems algebraically turns out to be very laborious. Of course, this depends somewhat on the system that one uses: Perspect – because of the heavy restrictions that it imposes – is rather bad in this area. However, all systems that we encountered had this problem. Therefore, we do not consider modular initial algebra specifications, in the form they have today, to be a useful tool for practical software development.

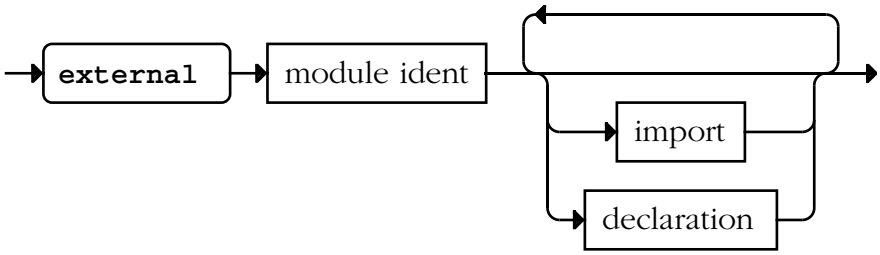
Appendices

The gods, greatly amused, revive you for some more fun.

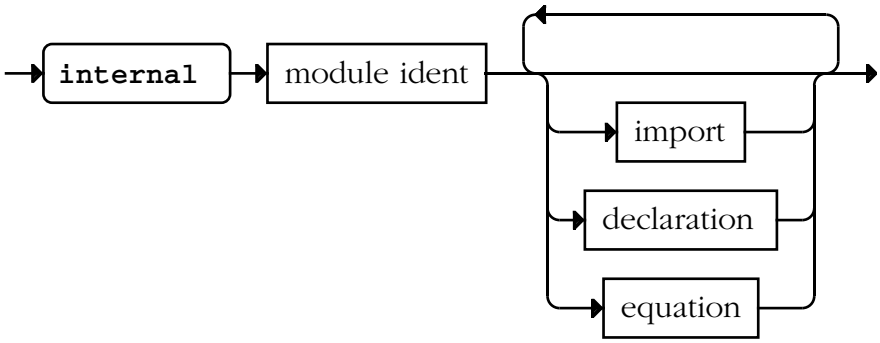
I. Perspect syntax diagrams

(For the context free grammar in EBNF format, from which these diagrams have been derived, see section 3.1.1.2.)

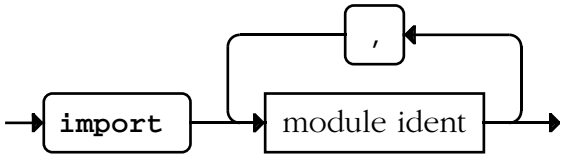
external



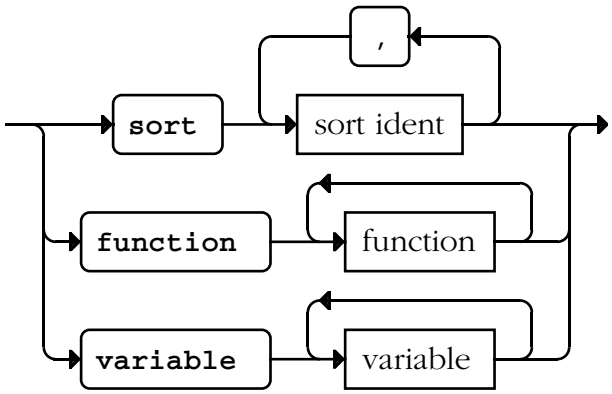
internal



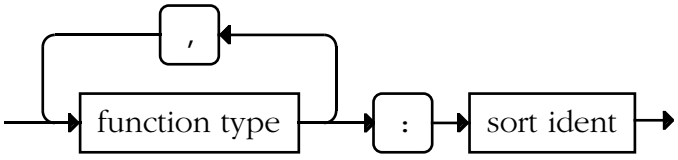
import



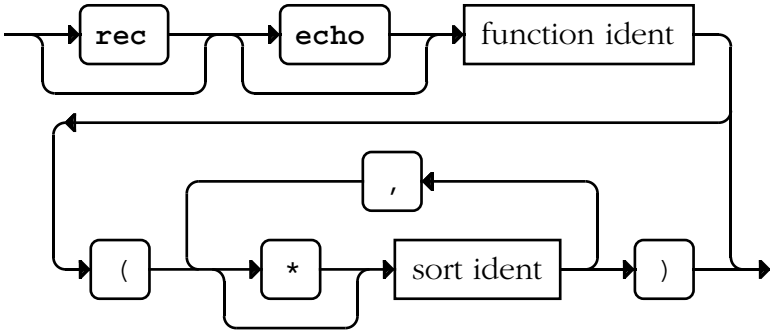
declaration



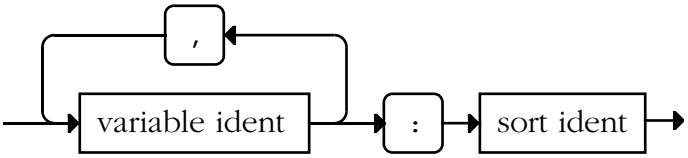
function



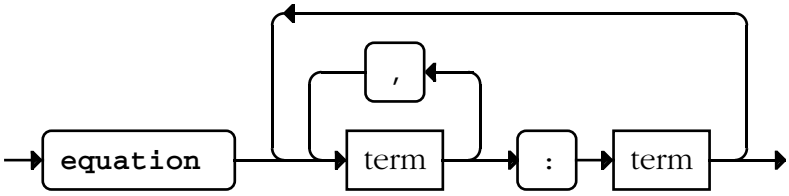
function type



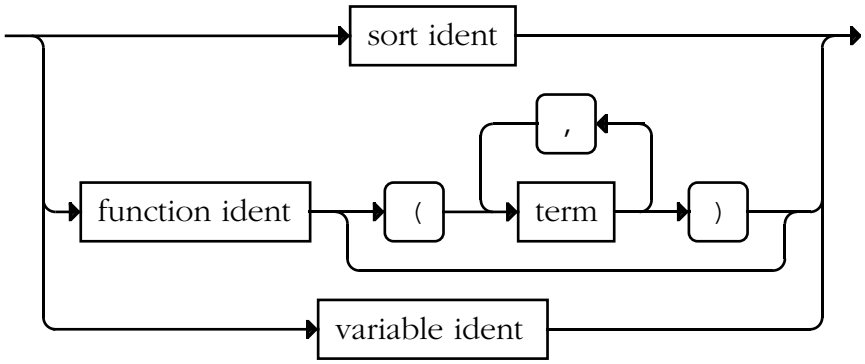
variable



equation



term



II. Error messages from the Perspect system

Environment errors

```
##  Can't open: File filename.
```

Language errors

```
##  End of file reached inside a comment. [1.1]

##  Illegal character in comment, ASCII value: value. [1.1]

##  Syntax error: Unexpected token. [1.2]

##  Module identifiers don't match: identifier should be identifier.
[3.1]

##  A module called identifier already exists. Module skipped. [3.1]

##  Echo-declaration of identifier.
#   Echo's are only allowed in the internal part of a module.
[3.2]
```

```
  ##  Domains don't match in echo-declaration of function identifier.
[3.2]

  ##  Ranges don't match in echo-declaration of function identifier.
[3.2]

  ##  Echo of function identifier.
  #   This function is declared in module identifier, but echoed in
module identifier. [3.2]

  ##  Echo of local function identifier.
  #   Echo declarations are only allowed for exported functions.
[3.2]

  ##  Function identifier already has an echo in this module. [3.2]

  ##  Echo of function identifier should not have rec external
counterpart. [3.2]

  ##  The external counterpart of the echo declaration of function
identifier
  #   should not contain stars. [3.2]

  ##  Module identifier doesn't exist. [3.3]

  ##  Nameclash: identifier already in use. [3.4]

  ##  Nameclash of identifier on (implicit) import of identifier in
identifier. [3.4]

  ##  Import of identifier skipped.
  #   Replaced by all safe imports in the external part of identifier.
[3.4]

  ##  Unknown sort identifier. [3.5]

  ##  identifier is not a sort. [3.5]

  ##  Unknown function identifier. [3.5]

  ##  identifier is not a function. [3.5]

  ##  Unknown variable identifier. [3.5]

  ##  identifier is not a variable. [3.5]
```

```
  ##  Unknown in term: object. [3.5]

  ##  The variable identifier should not have arguments. [3.5]

  ##  The function identifier has wrong arguments. [3.5]

  ##  Equation incorrectly typed.
  #     term has type identifier, but
  #     term has type identifier. [3.5]

  ##  Module identifier is not left linear.
  #   The equation
  #     term:
  #       term
  #   is not left linear. [5.1]

  ##  The equation
  #     term:
  #       term
  #   has variables occurring right, that don't occur left,
  #   or has unnamed variables occurring right. [5.1]

  ##  The equation
  #     term:
  #       term
  #   has variables occurring left, that don't occur right. It
should be
  #     term:
  #       term
  #   [5.1]

  ##  The first function of the external part of module identifier
  #   can't have the rec attribute. [5.2]

  ##  The first function of the internal part of module identifier
  #   can't have the rec attribute. [5.2]

  ##  Module identifier is not monotone terminating.
  #     term:
  #       term
  #   Maybe some functions should be reordered. [5.2]

  ##  Module identifier is not open confluent.
  #     term:
  #       term
```

```
#       term
#   Maybe the equations
#     term: term
#     term: term
#   should be more specific. [5.3]

##  Module identifier is not strongly persistent.
#   There are new normal forms of sort identifier of the form
#     term
#     term
#     ...
#   added in this module (junk). [5.4]

##  Module identifier is not strongly persistent.
#   Some normal forms of sort identifier of the form
#     term
#     term
#     ...
#   are now reducible (confusion).
#   Maybe the equation
#     term: term
#   should be more specific. [5.4]
```

Implementation restrictions

```
##  Implementation restriction: Memory full.

##  Implementation restriction: Identifier too long.
#   Truncated to count characters.
#   The identifier becomes: identifier

##  Implementation restriction: Expression too complicated.
#   Parser stack overflow.
```

Internal errors

```
##  Internal error: Keyword defined twice.
#   Internal codes are: code and code.

##  Internal error: Dangling pointer.

##  Internal error: Trying to hash NIL Ident.

##  Internal error: Unidentified object object.
```

```
##  Internal error: Garbage collection error in ParserTypes
#   count objects not deallocated.

##  Internal error: Garbage collection error.
#    allocations: count
#    extra references: count

##  Internal error: Reducing a term that is not linear.
```

III. Definition modules of the Perspect checker

```
DEFINITION MODULE StdAlloc;
 IMPORT SYSTEM;
 PROCEDURE NewPtr (size: INTEGER): SYSTEM.ADDRESS;
END StdAlloc.

DEFINITION MODULE StdInOut;
 IMPORT SYSTEM;
 TYPE
  String = ARRAY [0..255] OF CHAR;
  StringPtr = POINTER TO String;
  StringVec = ARRAY [0..4000] OF StringPtr;
  StringVecPtr = POINTER TO StringVec;
 VAR
  ArgC: INTEGER;
  ArgV: StringVecPtr;
 CONST
  RDONLY = 0000H;
 PROCEDURE open
  (filename: ARRAY OF CHAR; mode: INTEGER): INTEGER;
 CONST
  InputFD = 0;
  OutputFD = 1;
 PROCEDURE close (fd: INTEGER): INTEGER;
 PROCEDURE read
  (fd: INTEGER; buffer: SYSTEM.ADDRESS; count: INTEGER): INTEGER;
 PROCEDURE write
  (fd: INTEGER; buffer: SYSTEM.ADDRESS; count:INTEGER): INTEGER;
END StdInOut.

DEFINITION MODULE Allocation;
 IMPORT SYSTEM;
 PROCEDURE Allocate (VAR ptr: SYSTEM.ADDRESS; size: INTEGER);
 PROCEDURE Free (VAR ptr: SYSTEM.ADDRESS);
END Allocation.
```

```
DEFINITION MODULE Input;
 VAR
  eof: BOOLEAN;
  next: CHAR;
 PROCEDURE Get;
 PROCEDURE Read (VAR c: CHAR);
 PROCEDURE InputLoc
  (VAR file: ARRAY OF CHAR; VAR n: INTEGER;
   VAR pos, line: INTEGER);
END Input.

DEFINITION MODULE Output;
 PROCEDURE Write (c: CHAR);
 PROCEDURE WriteInt (i: INTEGER);
 PROCEDURE WriteString (s: ARRAY OF CHAR);
 PROCEDURE WriteNString (s: ARRAY OF CHAR; n: INTEGER);
 PROCEDURE WriteLn;
END Output.

DEFINITION MODULE Messages;
 PROCEDURE OpenError (message: ARRAY OF CHAR);
 PROCEDURE AddToError (message: ARRAY OF CHAR);
 PROCEDURE CloseError
  (reference: ARRAY OF CHAR; errorLocation: BOOLEAN);
END Messages.

DEFINITION MODULE Idents;
 TYPE Ident;
 PROCEDURE KeyWord (c: ARRAY OF CHAR; kind: INTEGER);
 PROCEDURE GetIdent
  (c: ARRAY OF CHAR; n: INTEGER; VAR kind: INTEGER): Ident;
 PROCEDURE WriteIdent (VAR ident: Ident);
 CONST HashSize = 101;
 PROCEDURE Hash (ident: Ident): INTEGER;
END Idents.

DEFINITION MODULE ParserTypes;
 IMPORT Idents;
 TYPE IdentList;
 PROCEDURE EmptyIdentList (): IdentList;
 PROCEDURE AppendToIdentList
  (theList: IdentList; isStar: BOOLEAN; theElement: Idents.Ident):
   IdentList;
 PROCEDURE DisposeIdentList (theList: IdentList);
```

```
    TYPE ProcOnIdent = PROCEDURE(BOOLEAN,Idents.Ident);
    PROCEDURE ForEachIdentIn
     (theList: IdentList; theAction: ProcOnIdent);
   TYPE Type;
   PROCEDURE NewType
    (isRec, isEcho: BOOLEAN; theIdent: Idents.Ident;
     theDomain: IdentList): Type;
   PROCEDURE DisposeType (theType: Type);
   TYPE TypeList;
   PROCEDURE EmptyTypeList (): TypeList;
   PROCEDURE AppendToTypeList
    (theList: TypeList; theElement: Type): TypeList;
   PROCEDURE DisposeTypeList (theList: TypeList);
    TYPE ProcOnType =
     PROCEDURE(BOOLEAN,BOOLEAN,Idents.Ident,IdentList);
    PROCEDURE ForEachTypeIn
     (theList: TypeList; theAction: ProcOnType);
   TYPE
    Term;
    TermList;
   PROCEDURE NewTerm
    (theIdent: Idents.Ident; theArgs: TermList): Term;
   PROCEDURE DisposeTerm (theTerm: Term);
   PROCEDURE DecomposeTerm
    (theTerm: Term; VAR theIdent: Idents.Ident;
     VAR theArgs: TermList);
   PROCEDURE EmptyTermList (): TermList;
   PROCEDURE AppendToTermList
    (theList: TermList; theElement: Term): TermList;
   PROCEDURE DisposeTermList (theList: TermList);
    TYPE ProcOnTerm = PROCEDURE(Term);
    PROCEDURE ForEachTermIn
     (theList: TermList; theAction: ProcOnTerm);
  END ParserTypes.

  DEFINITION MODULE Attributes;
   IMPORT Idents;
   IMPORT ParserTypes;
   TYPE
    YYSTYPE =
     RECORD
      CASE: INTEGER OF
      0: boolean: BOOLEAN |
      1: ident: Idents.Ident |
      2: identList: ParserTypes.IdentList |
```

```
      3: type: ParserTypes.Type |
      4: typeList: ParserTypes.TypeList |
      5: term: ParserTypes.Term |
      6: termList: ParserTypes.TermList
      ELSE
      END
    END;
END Attributes.

DEFINITION MODULE Scanner;
 IMPORT Attributes;
 PROCEDURE yylex
  (VAR token: INTEGER; VAR value: Attributes.YYSTYPE);
 PROCEDURE yyerror (token: INTEGER; value: Attributes.YYSTYPE);
 PROCEDURE yyabort;
END Scanner.

DEFINITION MODULE Parser;
 IMPORT Attributes;
 PROCEDURE yyparse (VAR yyresult: Attributes.YYSTYPE): BOOLEAN;
END Parser.

DEFINITION MODULE Modules;
 IMPORT Idents;
 IMPORT ParserTypes;
 IMPORT Symbols;
 TYPE Module = Symbols.Module;
 PROCEDURE EmptyModule (theIdent: Idents.Ident): Module;
 PROCEDURE CheckModuleIdent
  (theModule: Module; theIdent: Idents.Ident);
 PROCEDURE CloseModule (theModule: Module);
 PROCEDURE AddImport
  (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident);
 PROCEDURE AddSort
  (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident);
 PROCEDURE AddFunction
  (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident;
   theDomain: ParserTypes.IdentList; theRange: Idents.Ident;
   isRec: BOOLEAN);
 PROCEDURE EchoFunction
  (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident;
   theDomain: ParserTypes.IdentList; theRange: Idents.Ident;
   isRec: BOOLEAN);
 PROCEDURE AddVariable
  (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident;
```

```
    theTypeIdent: Idents.Ident);
  PROCEDURE AddEquation
   (theModule: Module;
    leftHandSide, rightHandSide: ParserTypes.Term);
END Modules.

DEFINITION MODULE Symbols;
 IMPORT Idents;
 TYPE
  Module;
 PROCEDURE NewModule (theIdent: Idents.Ident): Module;
 PROCEDURE ModuleIdent (theModule: Module): Idents.Ident;
 PROCEDURE AddModuleToSpec (theModule: Module);
 PROCEDURE FindModuleInSpec
  (theIdent: Idents.Ident; VAR foundModule: Module): BOOLEAN;
 PROCEDURE AddImport
  (theModule: Module; visible: BOOLEAN; theImported: Module);
 TYPE
  Sort;
 PROCEDURE NewSort
  (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident);
 PROCEDURE FindSort
  (theModule: Module; theIdent: Idents.Ident;
   VAR foundSort: Sort): BOOLEAN;
 TYPE
  Domain;
 PROCEDURE EmptyProduct (): Domain;
 PROCEDURE Product
  (theDomain: Domain; theSort: Sort; isStar: BOOLEAN): Domain;
 PROCEDURE StarInDomain (theDomain: Domain): BOOLEAN;
 PROCEDURE EqualDomain
  (theDomain1, theDomain2: Domain): BOOLEAN;
 TYPE
  Function;
 PROCEDURE NewFunction
  (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident;
   theDomain: Domain; theRange: Sort; isRec: BOOLEAN);
 PROCEDURE FindFunction
  (theModule: Module; theIdent: Idents.Ident;
   VAR foundFunction: Function): BOOLEAN;
 PROCEDURE FunctionDomain (theFunction: Function): Domain;
 PROCEDURE FunctionRange (theFunction: Function): Sort;
 PROCEDURE EchoFunction
  (theModule: Module; theFunction: Function; isRec: BOOLEAN);
 TYPE
  Variable;
```

```
  PROCEDURE NewVariable
   (theModule: Module; visible: BOOLEAN; theIdent: Idents.Ident;
    theType: Sort);
  PROCEDURE FindVariable
   (theModule: Module; theIdent: Idents.Ident;
    VAR foundVariable: Variable): BOOLEAN;
  TYPE
   Term;
   Tuple;
  PROCEDURE NewTerm
   (theModule: Module; theHead: Idents.Ident; theArgs: Tuple):
    Term;
  PROCEDURE TermIsApplication
   (theTerm: Term; VAR theFunction: Function; VAR theArgs: Tuple):
    BOOLEAN;
  PROCEDURE TermType (theTerm: Term; VAR theType: Sort): BOOLEAN;
  PROCEDURE WriteTerm (theTerm: Term);
  PROCEDURE EmptyTuple (): Tuple;
  PROCEDURE AppendToTuple
   (theTuple: Tuple; theTerm: Term): Tuple;
  PROCEDURE TupleDomain
   (theTuple: Tuple; theDomain: Domain): BOOLEAN;
  TYPE
   Equation;
  PROCEDURE NewEquation
   (leftHandSide, rightHandSide: Term): Equation;
  PROCEDURE AddEquation
   (theModule: Module; theEquation: Equation);
  TYPE TermSet;
  TYPE TermSetList;
 END Symbols.
```

References

[Apple, 1985]
Apple Computer, Inc, *Inside Macintosh*. Volumes I, II, and III, Addison-Wesley Publishing Company. Reading, Massachusetts, etc., 1985.

[Apple, 1987]
Apple Computer, Inc, *Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley Publishing Company. Reading, Massachusetts, etc., 1987.

[Bergstra, 1988]
J.A. Bergstra, 'Process Algebra for Synchronous Communication and Observation'. Report P8815 of the Programming Research Group, Department of Mathematics and Computer Science, University of Amsterdam. Amsterdam, 1988.

[Bergstra, 1989]
J.A. Bergstra, 'Process Algebra for Synchronous Communication and Observation'. Report P8815b of the Programming Research Group, Department of Mathematics and Computer Science, University of Amsterdam. Amsterdam, 1989.

[Bergstra, Heering, Klint, 1985]
J.A. Bergstra, J. Heering, P. Klint, 'Algebraic definition of a simple programming language'. Report CS-R8504 of the Department of Computer Science, Centre for Mathematics and Computer Science. Amsterdam, 1985.

[Bergstra, Heering, Klint, 1989]
Algebraic Specification, Edited by J.A. Bergstra, J. Heering, P. Klint. Addison-Wesley Publishing Company. Wokingham, England, etc., 1989.

[Bergstra, Klop, Middeldorp, 1989]
J.A. Bergstra, J.W. Klop, A. Middeldorp. *Termherschrijfsystemen*. Kluwer. Deventer, 1989.

[Bergstra, Mauw, Wiedijk, 1989]
J.A. Bergstra, S. Mauw, F. Wiedijk, 'Uniform algebraic specifications of finite sorts with equality'. Report P8904 of the Programming Research Group, Department of Mathematics and Computer Science, University of Amsterdam. Amsterdam, 1989.

[Bergstra, Tucker, 1988]
J.A. Bergstra, J.V. Tucker, 'The Inescapable Stack: An Exercise in Algebraic Specification with Total Functions'. Report P8804 of the Programming Research Group,

Department of Mathematics and Computer Science, University of Amsterdam. Amsterdam, 1988.

[Bidoit, 1981]
M. Bidoit, *Une Methode de Presentation des Types Abstraits: Applications*. These 3-eme cycle, Université de Paris Sud. Orsay, 1981.

[Bidoit, Kreowski, Lescanne, Orejas, Sannella, 1991]
M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, D. Sannella, *Algebraic System Specification and Development, A Survey and Annotated Bibliography*. Lecture Notes in Computer Science 501. Springer-Verlag. Berlin, etc., 1991.

[Böhm, 1987]
H-J. Böhm, 'Constructive Real Interpretation of Numerical Programs', In: *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, *SIGPLAN Notices* **22**, 7, July 1987, pp. 214-221.

[Böhm, Cartwright, O'Donnell, Riggle, 1986]
H-J. Böhm, R. Cartwright, M.J. O'Donnell, M. Riggle, 'Exact Real Arithmetic: A Case Study in Higher Order Programming'. In: *Proceedings of the 1986 Lisp and Functional Programming Conference*, pp. 162-173.

[Böhm, Weiser, 1988]
H-J. Böhm, M. Weiser, 'Garbage Collection in an Uncooperative Environment'. In: *Software-Practice and Experience*, **18**, 9, September 1988, pp. 807-820.

[Book, Gallier, 1985]
J.H. Gallier, R.V. Book, 'Reductions in tree replacement systems'. In: *Theoretical Computer Science* **37**, 1, 1985, pp. 123-150.

[Bouma, Walters, 1987]
L.G. Bouma, H.R. Walters, 'Implementing algebraic specifications'. Report P8714 of the Programming Research Group, Department of Mathematics and Computer Science, University of Amsterdam. Amsterdam, 1987.

[Broy, Wirsing, 1982]
M. Broy, M. Wirsing, 'Partial abstract data types'. *Acta Informatica*, **18** (1), pp. 47-64. 1982.

[Clocksin, Mellish, 1981]
W.F. Clocksin, C.S. Mellish, *Programming in Prolog*. Springer-Verlag. Berlin, 1981.

[Dershowitz, 1982]
N. Dershowitz, 'Orderings for term-rewriting systems'. *Theoretical Computer Science* **17**, pp. 279-301.

[Dershowitz, Jouannaud, 1991]
N. Dershowitz, J.P. Jouannaud, 'Rewriting Systems'. In: *Handbook of Theoretical Computer Science*. North-Holland. Amsterdam, 1991.

[Drosten, Ehrich, 1984]
K. Drosten, H.-D. Ehrich, 'Translating algebraic specifications to Prolog programs'. Bericht Nr. 84-08, Institut für Informatik, TU Braunschweig. Braunschweig, 1984.

[Ehrich, Mahr, 1985]
H. Ehrich, B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. Springer-Verlag. Berlin, etc., 1985.

[van Emden, Yukawa, 1986]
M.H. van Emden, K. Yukawa, 'Equational Logic Programming'. Technical Report CS-86-05, University of Waterloo. Waterloo, Ontario, 1986.

[Futatsugi, Goguen, Jouannaud, Meseguer, 1985]
K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer, 'Principles of OBJ2'. In: *Proceedings 12th ACM Symposium on Principles of Programming Languages*,pp. 52-66. New Orleans, 1985.

[Ganzinger, Schäfers, 1990]
H. Ganzinger, R. Schäfers, 'System support for modular order-sorted Horn clause specifications'. *Proceedings of the 12th International Conference on Software Engineering*, Nice, IEEE Computer Society Press, pp. 150-163. 1990.

[Geser, Hussmann, Mueck, 1987]
A. Geser, H. Hussmann, A. Mueck, 'A compiler for a class of conditional rewrite systems'. *Proceedings of the International Workshop on Conditional Term Rewriting*, Orsay, Springer Lecture Notes in Computer Science 308, pp. 84-90. Berlin, etc., 1987.

[Goguen, Jouannaud, Meseguer, 1985]
J.A. Goguen, J.-P. Jouannaud, J. Meseguer, 'Operational semantics of order-sorted algebra'. In: *Proceedings 12th International Conference on Automata, Languages and Programming*, Lecture Notes in Computer Science 194, pp. 221-231. Springer-Verlag. Berlin, etc., 1985.

[Goguen, Thatcher, Wagner, 1976]
J.A. Goguen, J.W. Thatcher, E.G. Wagner, 'An initial algebra approach to the specification, correctness and implementation of abstract data types'. Report RC-6487, IBM T.J. Watson Research Center. Yorktown Heights, 1976.

[Goldberg, 1984]
A. Goldberg, *Smalltalk-80, The Interactive Programming Environment*. Addison-

Wesley Publishing Company. Reading, Massachusetts, etc., 1984.

[Guttag, Horning, 1978]
J.V. Guttag, J.J. Horning, 'The algebraic specification of abstract data types'. *Acta Informatica* **10**, pp. 27-52, 1978.

[Hendriks, 1991]
P.R.H. Hendriks, *Implementation of Modular Algebraic Specifications*. Ph.D. thesis, University of Amsterdam. Amsterdam, 1991.

[Hoekzema, Rodenburg, 1987]
D.J. Hoekzema, P.H. Rodenburg, 'Specification of the fast Fourier transform algorithm as a term rewriting system'. Logic Group Preprint Series 27, Department of Philosophy, University of Utrecht. Utrecht, 1987.

[Huet, Hullot, 1980]
G. Huet, J.M. Hullot, 'Proofs by induction in equational theories with constructors'. *Proceedings of the 21th symposium on Foundations of Computer Science*, 1980.

[Huet, Oppen, 1980]
G. Huet, D.C. Oppen, 'Equations and Rewrite Rules: a Survey'. Technical Report CSL-111, SRI International. Menlo Park, 1980.

[Hussmann, 1987]
H. Hussmann, 'Rapid prototyping for algebraic specifications – RAP system user's manual'. second revised edition, Report MIP-8504, Universität Passau. Passau, 1987.

[IEEE, 1981]
The Institute of Electrical and Electronics Engineers, Inc, 'A Proposed Standard for Binary Floating-Point Arithmetic'. In: *Computer* **14**, 3, March 1981, pp. 51-62.

[IEEE, 1985]
The Institute of Electrical and Electronics Engineers, Inc, 'IEEE Standard for Binary Floating-Point Arithmetic', ANSI/IEEE Standard 754-1985. Reprinted in: *Sigplan Notices* **22**, 2, February 1987, pp. 9-25.

[Janssen, 1983]
T.M.V. Janssen, *Foundations and Applications of Montague Grammar*. Ph.D. thesis, University of Amsterdam. Amsterdam, 1983.

[Kaplan, 1987]
S. Kaplan, 'A compiler for conditional term rewriting systems'. *Proceedings of the 2nd Conference on Rewriting Techniques and Applications*, Bordeaux, Springer Lecture Notes in Computer Science 256, pp. 25-41. Berlin, etc., 1987.

[Kernighan, Ritchie, 1988]
B.W. Kernighan, D.M. Ritchie, *The C programming language*. Second Edition, Prentice Hall. Englewood Cliffs, New Jersey, 1988.

[Klaeren, Indermark, 1989]
H. Klaeren, K. Indermark, 'A new technique for compiling recursive function symbols'. *Algebraic Methods: Theory, Tools and Applications*. Springer Lecture Notes in Computer Science 394, pp. 69-90. Berlin, etc., 1989.

[Klop, 1980]
J.W. Klop, *Combinatory Reduction Systems*. Ph. D. Thesis, University of Utrecht. Also: Mathematical Centre Tract 127, Centre for Mathematics and Computer Science. Amsterdam, 1980.

[Klop, 1991]
J.W. Klop, 'Term Rewriting Systems'. In: *Handbook of Logic in Computer Science*, Volume II. Edited by S. Abramsky, e.a. Oxford University Press. Oxford, 1991.

[Knuth, 1973]
D.E. Knuth, *The Art of Computer Programming, Volume 3 / Sorting and Searching*. Addison-Wesley Publishing Company. Reading, Massachusetts, etc., 1973.

[Knuth, Bendix, 1970]
D.E. Knuth, P.B. Bendix, 'Simple word problems in universal algebras'. In: *Computational Problems in Abstract Algebra*, pp. 263-297. Edited by J. Leech. Pergamon Press. 1970.

[Kruskal, 1960]
J.B. Kruskal, 'Well-Quasi-Ordering, the Tree Theorem, and Vazsonyi's Conjecture', *Transactions of the AMS* **95**, 1960, pp. 210-225.

[Mulder, 1990]
J.C. Mulder, *Case studies in process specification and verification*. Ph.D. thesis, University of Amsterdam. Amsterdam, 1990.

[Newman, 1942]
M.H.A. Newman, 'On Theories with a Combinatorial Definition of Equivalence', *Annals of Mathematics* **43**, 2, 1942, pp. 223-243.

[O'Donnell, 1977]
M.J. O'Donnell, *Computing in systems described by equations*. Lecture Notes in Computer Science 58. Springer-Verlag. Berlin, etc., 1977.

[Padawitz, 1983]
P. Padawitz, *Correctness, Completeness, and Consistency of Equational Data*

Type Specifications. Ph.D. thesis, Technical Report 83-15, Technische Universität Berlin. Berlin, 1983.

[Pemberton, 1989]
S. Pemberton, Private communication.

[Rémy, Uhrig, 1988]
J.-L. Rémy, S. Uhrig, 'An algorithm for testing sufficient completeness of a simple class of conditional specifications'. Report CRIN 88-R-155, Centre de Recherche en Informatique de Nancy. Nancy, 1988.

[Thiel, 1984]
J.J. Thiel, 'Stop losing sleep over incomplete data type specifications'. *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pp. 76-82, 1984.

[Walters, 1991]
H.R. Walters, *On Equal Terms: Implementing Algebraic Specifications*. Ph.D. thesis, University of Amsterdam. Amsterdam, 1991.

[Wiedijk, 1987]
F. Wiedijk, 'Termherschrijfsystemen in Prolog'. Report P8704 of the Programming Research Group, Department of Mathematics and Computer Science, University of Amsterdam. Amsterdam, 1987.

[Wiedijk, 1988]
F. Wiedijk, 'Voorlopig rapport over de specificatie-taal Perspect'. Report P8811 of the Programming Research Group, Department of Mathematics and Computer Science, University of Amsterdam. Amsterdam, 1988.

[Wirth, 1985]
N. Wirth, *Programming in Modula-2*. Third, Corrected Edition. Springer-Verlag. Berlin, etc., 1985.

[Wolz, Boehm, 1989]
D. Wolz, P. Boehm, 'Compilation of LOTOS data type specifications'. *Proceedings of the 9th International Symposium on Protocol Specification, Testing, and Verification*. Enschede, 1989.

Samenvatting
(Summary in dutch)

In dit proefschrift getiteld *Persistence in Algebraic Specifications* – in het Nederlands *Persistentie in Algebraïsche Specificaties* – bestuderen we een aantal aspecten van het begrip persistentie uit de theorie van de algebraïsche specificaties. Zowel een theoretische beschrijving van dit begrip als een aantal praktische toepassingen hiervan komen aan de orde.

Een specificatie heet persistent als de betekenis van de datatypen die erin worden gespecificeerd niet afhangt van de module waarin het datatype wordt gebruikt. We argumenteren dat het niet persistent zijn van specificaties een heuristiek is om specificatiefouten op te sporen. Een aantal specificatiefouten die in specificaties uit de praktijk zijn aangetroffen worden nader geanalyseerd, en er wordt gekeken in hoeverre de persistentie van de specificaties door deze fouten is aangetast.

We presenteren een redelijk efficiënt algoritme voor het controleren van de persistentie van modulaire termherschrijfsystemen. Hoewel dit algoritme in het ergste geval tenminste exponentieel veel tijd en ruimte nodig heeft, suggereren experimenten dat in de praktijk redelijk met dit algoritme te werken is.

Het algoritme is alleen toepasbaar als de te controleren specificatie aan een aantal andere eigenschappen voldoet. Om concreet met deze algoritme te kunnen experimenteren, definiëren we een kleine specificatietaal die *Perspect* is gedoopt. In deze taal worden een aantal voorbeeldspecificaties gegeven (waaronder een specificatie van de rationale getallen, een specificatie van de primitief recursieve functies, een specificatie van plaatjes die uit monochrome pixels zijn opgebouwd en een specificatie van een kleine tekstverwerker). Ervaringen opgedaan met deze specificaties en met de implementatie van een Perspect-checker worden beschreven.

Perspect is een simpele taal voor het opschrijven van modulaire herschrijfsystemen. Er zijn vier eisen waaraan een Perspect specificatie moet voldoen :

(i) De herschrijfregels uit een Perspect specificatie mogen *geen condities* bevatten en moeten *links-lineair* zijn.

(ii) Een Perspect specificatie moet aan een sterke vorm van *terminatie* voldoen (de terminatie moet volgens een impliciet aan te geven pad-ordening controleerbaar zijn).

(iii) Een Perspect specificatie moet aan een sterke vorm van *confluentie* voldoen (confluentie moet ook bij het herschrijven van open termen gelden).

(iv) Een Perspect specificatie moet aan een sterke vorm van *persistentie* voldoen (normaalvormen moeten tussen de modules behouden blijven).

De definities van de verschillende begrippen die in deze eisen gebruikt worden zijn expliciet in het proefschrift opgenomen. Bij het schrijven van specificaties blijkt in de praktijk eis (ii) de meeste problemen te geven.

De taal Perspect heeft de volgende eigenschappen:

- Het is beslisbaar of een tekst een correcte Perspect specificatie is.
- Iedere correcte Perspect specificatie is persistent.
- Iedere correcte Perspect specificatie is executeerbaar.
- Elke primitief recursieve algebra is specificeerbaar door middel van een correcte Perspect specificatie.
- De semantiek van Perspect is compositioneel.

Perspect blijkt in de praktijk te restrictief voor realistische specificaties. Om aan dit probleem tegemoet te komen definiëren we een spectrum van vijf varianten van Perspect, met aan het ene uiterste een taal waarin in een specificatie desgewenst met een puur verbale beschrijving kan worden volstaan, en aan het andere uiterste Perspect zelf.

Vervolgens beschrijven we een tweetal compilatieschema's voor complete termherschrijfsystemen (en dus voor Perspect specificaties).

Het eerste schema behelst vertaling naar Prolog. In één uniform schema worden hier negen verschillende implementaties van het herschrijfsysteem gegeven, waaronder implementaties die herschrijven volgens de leftmost innermost reductiestrategie, volgens de leftmost outermost reductiestrategie, volgens de parallel outermost reductiestrategie en volgens de Gross-Knuth reductiestrategie. Dit algemene schema wordt vergeleken met twee andere schema's uit de literatuur. Een voorbeeld van een vertaling volgens dit schema – een vertaling van de specificatie van de kleine tekstverwerker – wordt expliciet uitgeschreven en een voorbeeld van een executie van deze vertaling wordt gegeven.

Het tweede compilatieschema dat wordt beschreven is een schema voor het vertalen van algebraïsche specificaties naar Modula-2. Er worden drie abstracte implementatiebegrippen ingevoerd: zwakke implementatie, sterke implementatie en termherschrijfimplementatie. Voor de laatste variant geven we een expliciet vertaalschema. Er wordt een expliciet uitgeschreven voorbeeld gegeven van de vertaling van een specificatie waarin vermenigvuldiging van de gehele getallen wordt gedefinieerd in termen van de optelling. De vertaling van de module waarin de gehele getallen en de optelling worden ingevoerd wordt vervolgens vervangen door een met de hand geschreven implementatie die gebruik maakt van het ingebouwde datatype van Modula-2. De module die de vermenigvuldiging definieert hoeft hierbij niet aangepast te worden.

Het proefschrift eindigt met enige beschouwingen over de bruikbaarheid van het gepresenteerde onderzoek in het bijzonder (conceptueel aantrekkelijk, maar in de praktijk niet erg handig) en van modulaire initiële algebra specificaties in het algemeen (in de huidige vorm te moeizaam voor serieuze toepassingen).

Stellingen

behorende bij het proefschrift van Freek Wiedijk getiteld
Persistence in Algebraic Specifications

1. Algebraïsche specificaties zijn vaak onbegrijpelijk, en vrijwel altijd incorrect.

2. Het staven van wiskundige bewijzen dient geautomatiseerd te worden. Zo zou het mogelijk moeten zijn een bewijs van de hoofdstelling over Jordan-krommen ('een deelverzameling van het platte vlak die homeomorf is met de cirkel verdeelt het platte vlak in precies twee samenhangscomponenten') in de computer in te voeren, en op correctheid te controleren. Om dit voorbeeld te kunnen verwerken, dient een bewijsverificator op de axiomatische verzamelingenleer gebaseerd te zijn.

3. Er bestaat geen eindig orthogonaal termherschrijfsysteem dat de volgende twee eigenschappen heeft:
   (i) De signatuur van het herschrijfsysteem bevat de soort NAT en de functies

   $$\begin{aligned} 0 &: \quad \rightarrow \mathrm{NAT} \\ 1 &: \quad \rightarrow \mathrm{NAT} \\ + &: \mathrm{NAT} \times \mathrm{NAT} \rightarrow \mathrm{NAT} \end{aligned}$$

   (ii) De normaalvormen van de soort NAT zijn 0, 1, 1+1, (1+1)+1, ((1+1)+1)+1, …

4. Het ordetype α van een aftelbare totale ordening waarvoor geldt $\alpha\cdot\alpha = \alpha$ heeft één van de volgende vijf vormen:

   $1+\eta+1$
   $\eta\cdot\beta$
   $(1+\eta)\cdot(1+\beta)$
   $(\eta+1)\cdot(\beta+1)$
   $(1+\eta_2+1)\cdot(1+\beta+1)$

   Hierin is β een willekeurig ordetype, η het ordetype van de rationale getallen, en η_2 het ordetype van de rationale getallen waarin de breuken met een tweemacht als noemer verdubbeld zijn.

5. Indien iemand twaalf maal willekeurig een stap neemt in één van de twaalf richtingen van de wijzerplaat, dan is de kans dat hij daarna op zijn beginpunt is teruggekeerd gelijk aan 17365421304 / 8916100448256, ofwel iets minder dan een vijfde

procent. (Hoe dit spel – met dodelijke afloop – gespeeld kan worden, is te lezen op bladzijden 118 en 119 van Charles Harness' roman *De ring van Ritornel* in de uitgave van Meulenhoff uit 1976)

6. De driedimensionale kubus heeft 20 verschillende tweedimensionale bouwplaten (of 11 verschillende, indien gespiegelde bouwplaten niet als verschillend worden opgevat). De vierdimensionale hyperkubus heeft 455 (of 261) verschillende driedimensionale hyperbouwplaten. De vijfdimensionale hyperkubus heeft 17296 (of 9694) verschillende vierdimensionale hyperbouwplaten.

7. Conforme supergravitatie (voor N=1) kan op een natuurlijke wijze worden opgevat als een veldentheorie met 1344 bosonische en 1344 fermionische vrijheidsgraden.

8. De *Watch It!* INIT, waarvan het INIT resource bestaat uit de 525 bytes:

```
6008494E49540101000041FAFFF4A1282F08A992303CA851A14643FA01162288303CA8
5141FA0006A0474E754E56000048E7E0E0206E00083210670000EE303C0006343A013A
B441650000E0670AE24A51C8FFF4600000D4323C00069240303C000743FA013A45FA01
162419E2A2C498E3AAB49A660000B451C8FFF041FA00BC2D4800082038020C48407200
320082FCA8C04840320082FCA8C042414841200106800000079E80FC0E100681000000
9682FC012C48414241484182FC000C4841B07A00746606B27A0070675E41FA006830C0
308141FA00A843FA0060303C000722D851C8FFFC41FA00D6303A004CC0FC0006D0C043
FA004E303C000512188319524951C8FFF841FA00B7303A002AC0FC0006D0C043FA0030
303C000412180201001F8319524951C8FFF44CDF07074E5E4EF90041D89A000000003F
003F003F003F00408084408440846084608040804040803F003F003F003F003F003F00
3F003F007F80FFC0FFC0FFC0FFC0FFC0FFC07F803F003F003F003F00000800083F003F
003F003F00408080408040806080608040804040803F003F003F003F00FFFFFFFFFFFF
FFFFFFFFE1FFC0FFC0FFC0FFC0FFE1FFFFFFFFFFFFFFFFFFFFFF040404040000020204
040000000001060000000000070000000000060100000000040402000000040404000 0
000404080000000C30000000003C00000000300C000008080404000004040404000000
```

verhoogt het realisme in het Macintosh interface, en dient daarom in de standaard systeem software van de Macintosh te worden opgenomen.

9. Een programma dat de limiet `Lim[If[x==0,1,0],x->0]` evalueert als `1` verdient de benaming *Mathematica* niet.

10. In tegenstelling tot wat Jorge Luis Borges in het korte verhaal *Blauwe Tijgers* suggereert, betekent het feit dat in de quantum-mechanica 'aantal deeltjes' geen behouden grootheid is niet de ondergang van de rekenkunde.

11. Indien iemand zich afvraagt of hij op dat moment droomt of wakker is, en het antwoord schuldig moet blijven, is het feitelijk zo dat hij wakker is, aangezien het droombewustzijn zich niet leent voor een dergelijke filosofische twijfel. Het gevolg is dat men, door deze observatie tot zijn filosofie te laten doordringen, de vraag of men droomt of niet, ook in de realiteit kan beantwoorden.

12. De mooiste maat uit Stravinsky's *Le Sacre du Printemps* is de laatste maat vóór het begin van de *Cercles mystérieux des adolescentes*.

Uitnodiging

Hierbij nodig ik U uit tot het bijwonen van mijn promotie op donderdag 12 december 1991 om 12.00 uur in de aula van de Universiteit van Amsterdam (Oude Lutherse kerk, Singel 411).

Na de promotie is er een receptie in de Tetterodebibliotheek van de aula.

Freek Wiedijk
Admiraal de Ruyterweg 285 I
1055 LV Amsterdam
020 - 6881825

In de omgeving van de aula kan slecht geparkeerd worden. De aula is goed bereikbaar met de tramlijnen 1, 2, 5 (halte Spui) en 4, 9, 14, 16, 24, 25 (halte Rokin).