

*Implementation of
Modular Algebraic
Specifications*

P.R.H. Hendriks

Implementation of Modular Algebraic Specifications

P.R.H. Hendriks

Implementation of Modular 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 vrijdag 24 mei 1991 te
15.00 uur, door

Paul Robert Hendrik Hendriks

geboren te Schaesberg

Promotor: prof. dr. P. Klint
Faculteit: Wiskunde en Informatica

© 1991, P.R.H. Hendriks. All rights reserved.

The research described in this thesis has been carried out while the author was employed at the CWI (Centrum voor Wiskunde en Informatica), Amsterdam.

Partial support for this research was received from the European Communities under ESPRIT projects 348 (GIPE - Generation of Interactive Programming Environments) and 2177 (GIPE II).

Chapter 4 is reprinted with kind permission of Addison-Wesley. An earlier version of it was published as Chapter 7 of

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

The cover is designed by Will van Hoof.

To my parents

Contents

| | |
|--|------|
| Contents | v |
| Preface | xi |
| Introduction | xi |
| Contents description | xii |
| Acknowledgements | xiii |
| 1 Specification Formalisms | 1 |
| 1.1 Introduction | 1 |
| 1.2 Algebraic Specification Formalism - ASF | 1 |
| 1.3 Syntax Definition Formalism - SDF | 6 |
| 1.4 The combined formalism ASF+SDF | 8 |
| 1.5 Use of the specification formalisms in this thesis | 13 |
| 2 The ASF System | 15 |
| 2.1 Introduction | 15 |
| 2.2 User-level architecture | 16 |
| 2.3 An example | 17 |
| 2.4 Internal structure | 18 |
| 2.4.1 Typechecking and normalization | 20 |
| 2.4.2 Generation of Prolog code | 20 |
| 2.4.3 Parsing of specification and input modules | 22 |
| 2.4.4 Reduction of terms to normal form | 23 |
| 2.5 Correctness and completeness | 24 |
| 2.5.1 Termination | 25 |
| 2.5.2 Confluence | 26 |
| 2.5.3 Conditions | 26 |
| 2.6 Possible improvements | 27 |
| 2.6.1 Optimization of generated code | 28 |

| | | |
|---------|---|----|
| 2.6.2 | Optimization of the reduction machine | 29 |
| 2.6.3 | Normalization versus modular compilation | 29 |
| 3 | Extending ASF with Associativity and Lists | 31 |
| 3.1 | Introduction | 31 |
| 3.2 | Algebraic specifications | 33 |
| 3.2.1 | Definitions | 33 |
| 3.2.2 | Implementation in Prolog | 35 |
| 3.2.2.1 | Implementation of equations without conditions | 35 |
| 3.2.2.2 | Implementation of conditions | 36 |
| 3.3 | Algebraic specifications with associativity | 37 |
| 3.3.1 | Example | 37 |
| 3.3.2 | Definitions | 37 |
| 3.3.3 | Semantics | 38 |
| 3.3.4 | Implementation in Prolog | 38 |
| 3.4 | Algebraic specifications with lists | 44 |
| 3.4.1 | Example | 44 |
| 3.4.2 | Definitions | 45 |
| 3.4.3 | Semantics | 48 |
| 3.4.4 | Implementation in Prolog | 50 |
| 3.5 | Conclusions | 53 |
| 4 | A Case Study in ASF+SDF: Typechecking Mini-ML | 55 |
| 4.1 | Introduction | 55 |
| 4.2 | Mini-ML | 56 |
| 4.2.1 | Concrete syntax of Mini-ML | 56 |
| 4.2.2 | Typechecking Mini-ML | 57 |
| 4.2.2.1 | Syntax and semantics of types and generalized types | 57 |
| 4.2.2.2 | The typechecker | 59 |
| 4.3 | Algebraic specification of Mini-ML typechecking | 61 |
| 4.3.1 | Basic notions | 61 |
| 4.3.1.1 | Layout | 61 |
| 4.3.1.2 | Booleans | 62 |
| 4.3.1.3 | Natural-Numbers | 63 |
| 4.3.1.4 | Tables | 64 |

| | | |
|---------|--|-----|
| 4.3.2 | The syntax of Mini-ML | 66 |
| 4.3.3 | Types and tools to handle types | 67 |
| 4.3.3.1 | Types | 67 |
| 4.3.3.2 | Type-Substitutions | 70 |
| 4.3.3.3 | Type-Environments | 73 |
| 4.3.3.4 | Type-Instant-Generalize | 75 |
| 4.3.3.5 | Type-Equations | 77 |
| 4.3.4 | The typechecker of Mini-ML | 79 |
| 4.3.4.1 | Type-Errors | 79 |
| 4.3.4.2 | Mini-ML-Typecheck | 80 |
| 4.4 | Implementations of the Mini-ML specification | 86 |
| 4.5 | Further research | 88 |
| 5 | The ASF+SDF System | 91 |
| 5.1 | Introduction | 91 |
| 5.2 | User interface | 92 |
| 5.2.1 | Starting the system | 92 |
| 5.2.2 | Editing | 96 |
| 5.2.3 | Miscellaneous features | 103 |
| 5.2.3.1 | Debugger | 103 |
| 5.2.3.2 | Configuration file | 106 |
| 5.3 | Syntax of ASF+SDF | 107 |
| 5.3.1 | Syntax of the syntax part of an ASF+SDF module | 108 |
| 5.3.2 | Syntax of the equations part of an ASF+SDF module | 117 |
| 5.4 | Internal structure | 118 |
| 5.4.1 | Syntax manager - SM | 120 |
| 5.4.2 | Equation manager - EQM | 121 |
| 5.4.3 | Module manager - MM | 121 |
| 5.4.4 | Generic syntax-directed editor - GSE | 122 |
| 5.4.5 | Supervisor - SV | 122 |
| 5.5 | Assessment | 123 |
| 6 | Textual Modularization and its Semantic Consequences | 125 |
| 6.1 | Introduction | 125 |
| 6.1.1 | Modularization | 125 |

| | | |
|---------|---|-----|
| 6.1.2 | Related work | 126 |
| 6.2 | Definitions | 128 |
| 6.2.1 | Non-modular specification formalisms | 128 |
| 6.2.2 | Modular specification formalisms | 128 |
| 6.3 | Semantic consequences | 132 |
| 6.3.1 | Non-compositional semantics | 134 |
| 6.3.2 | Theory semantics | 135 |
| 6.3.3 | Model class semantics | 142 |
| 6.3.4 | Conclusions | 144 |
| 6.4 | Algebraic specification of textual modularization | 145 |
| 6.4.1 | Sets | 145 |
| 6.4.2 | Non-modular specification formalisms | 147 |
| 6.4.3 | Modular specification formalisms | 149 |
| 6.4.3.1 | Modular-Specifications | 149 |
| 6.4.3.2 | Import-Graph | 152 |
| 6.4.3.3 | Normalization | 154 |
| 6.4.3.4 | Typechecking | 154 |
| 6.4.4 | Modular specification of signatures | 158 |
| 7 | Incremental Processing of Modular Specifications | 163 |
| 7.1 | Introduction | 163 |
| 7.2 | Functionality of system components | 165 |
| 7.2.1 | Item managers | 165 |
| 7.2.2 | Module manager | 166 |
| 7.3 | Algorithms | 167 |
| 7.3.1 | Incremental transitive closure | 167 |
| 7.3.2 | Module manager | 172 |
| 7.3.3 | Difference analysis algorithm - DAA | 176 |
| 7.3.3.1 | DAA with grains | 177 |
| 7.3.3.2 | DAA with grains for list operators | 178 |
| 7.3.3.3 | DAA for operators with independent children | 179 |
| 7.3.3.4 | Applications of DAA | 182 |
| 7.4 | Application: the ASF+SDF system | 186 |
| | Index | 189 |

| | |
|--------------------------------|-----|
| References | 195 |
| Samenvatting | 203 |
| Inleiding | 203 |
| Algebraïsch specificeren | 203 |
| Inhoudsbeschrijving | 207 |

Preface

Introduction

During the development of software as well as during its maintenance and use, an evident need exists for describing what the software does and how it does it. As mediums for description there are several possibilities.

- Natural language is often used to communicate about software; everyone can read it, but it has obvious drawbacks: it is not very precise, may give rise to more than one interpretation, and the internal consistency of descriptions cannot be validated mechanically.
- Several formal languages have been developed to eliminate the problems mentioned above. In several cases, software tools are provided to check for internal consistency and ambiguities in specifications written in these languages.
- Programming languages can also be viewed as specification languages; both the consistency of programs can be checked, and code can be generated. Even generated code can be viewed as a specification of a problem and its solution. It is, however, only readable to the happy few!

Much of the research in computer science, especially in the area of programming languages, is dedicated to developing structured, formal languages and enhancing their executability. In this thesis some aspects of importance in implementing modular algebraic specification formalisms are described.

Many-sorted, algebraic specification formalisms have a profound theoretical basis and are widely used to specify abstract data types. Each specification consists of a signature (a set of sorts and functions) and a set of (possibly conditional) equations. In the implementation, an algebraic specification is viewed as a term rewriting system by interpreting each equation as a rewrite rule from left to right. Many of the algebraic specification formalisms support some form of modularization as it is obvious that large specifications would otherwise be difficult to read and write.

In this thesis, the **Algebraic Specification Formalism ASF** [BHK89a] is used as a starting point. Its implementation is described, and extensions like the combination of ASF with the **Syntax Definition Formalism SDF** [HK89b, HHKR89] are discussed. The textual modularization used in ASF as well as in ASF+SDF is

formalized, and the semantic consequences of this modularization strategy are studied.

The development of ASF and ASF+SDF and the study of their implementation is part of a larger research effort aiming both at the automatic generation of programming environments from formal language definitions and at the construction of an interactive meta-environment for developing such definitions. These are the goals of ESPRIT-project 348 (GIPE - Generation of Interactive Programming Environments) and its successor ESPRIT-project 2177 (GIPE II).

Contents description

The first chapter of this thesis contains an introduction to the Algebraic Specification Formalism ASF, the Syntax Definition Formalism SDF, and the combination of these formalisms ASF+SDF. The chapter does not contain a formal description of all features of these formalisms, but merely introduces them by giving examples. Many aspects of the development, use, and implementation of these formalisms are described throughout this thesis. Many of the issues involved are, however, relevant to other specification formalisms. Textual modularization discussed in Chapters 6 and 7 is, for instance, also applicable to non-algebraic specification formalisms.

Chapter 2 gives a description of a simple, batch-oriented environment to compile and test specifications written in ASF. The architecture of this system and the implementation techniques applied in it are discussed. The system itself is mainly written in Prolog. After checking the correctness of a specification, it generates Prolog code which is used to reduce given terms to normal form. The system has been used to test several large specifications, and it has also been used to experiment with extensions of algebraic specification formalisms and their implementation.

An experiment in extending ASF and studying its implementation is described in Chapter 3. The algebraic specification formalism is enhanced with list constructors. The use of list constructors results not only in more elegant specifications, but also allows more powerful implementations to be generated for such specifications. As concatenation of lists is an associative binary operation, associativity is also discussed in Chapter 3. List constructors and associativity were not only inspired by the wish to improve the elegance of specifications and the possibilities to generate code for such specifications, but also by the wish to combine the formalisms ASF and SDF. Both features occur naturally in SDF, and their semantic consequences have to be handled in the algebraic specification formalism and its implementation.

Chapter 4 presents the first larger specification written in the combination of the formalisms ASF and SDF. The primary goal of this specification was to investigate

what a specification of the static semantics of a programming language supporting polymorphism and requiring type inference looks like. A major side effect of creating this specification was its influence on the development of the formalism SDF and its combination with ASF. The specification evolved keeping step with the development of these formalisms and their implementation in the ASF+SDF system.

The user interface and global architecture of the ASF+SDF system, an interactive system to develop and test specifications written in ASF+SDF are described in Chapter 5. Although the implementation of this system is not yet completely finished and will clearly evolve in the near future, several specifications have already been developed using the current version of the system. The system not only provides an environment to manipulate specifications of abstract data types, but when specifying a programming language it incrementally generates a programming environment. Thus supporting the developer of a programming language with tools.

As the modularization technique used in ASF as well as in ASF+SDF is based on textual expansion, the last two chapters of this thesis deal with textual modularization. Textual modularization means that a specification can be split into one or more modules each with a name. The (re)use of a module in another one is indicated by putting the name of the former in the list of imports of the latter. The semantics of such an import is given by replacing the import in the importing module by the text of the imported module. In Chapter 6 a formal definition and an algebraic specification of textual modularization are given. Its applicability is studied by investigating the semantic consequences of adding textual modularization to non-modular specification formalisms. The final Chapter 7 gives the description of the main algorithms needed in the incremental processing of specifications written in formalisms using textual modularization. The ASF+SDF system is a particular application of the general architecture described here.

Acknowledgements

Paul Klint created the environment which enabled me to do the research that resulted in this thesis. He introduced me to the topics, algebraic specifications, programming environments and their generation; in an inspiring way, he created a scientific environment where I worked with great pleasure. Jan Heering has his own influence on both “environments”. He took care of the more theoretical, philosophical part while Paul took a more pragmatic view. I am grateful to both of them for the opportunities created and the inspirational power of their activities.

The many formal and informal discussions with my fellow GIPE-members were also a source of inspiration for my activities at the CWI. For the Dutch part of the

project the GIPE-ers were: Arie van Deursen, Niek van Diepen, Hans van Dijk, Casper Dik, Wilco Koorn, Monique Logger, Emma van der Meulen, Jan Rekers, Frank Tip, Ard Verhoog, and Pum Walters. In my contacts with the French partners of the GIPE-project, discussions with Dominique Clément, Janet Incerpi, and Gilles Kahn were extremely fruitful.

In day to day life, roommates have great influence on the ambience of the work environment. I am very grateful to have shared my room with Ard Verhoog, Niek van Diepen, Monique Logger, Roland Bol, Jan Rekers, Emma van der Meulen, and Arie van Deursen. They sometimes had to stand my bad tempers, but, even worse, they also had to endure spontaneous outbursts of “singing” when everything seemed to work! I will always remember the many, spur-of-the-moment discussions on topics varying from games, paper folding, biblical knowledge, visual arts, and music through to little nieces.

I am grateful to the members of the reading committee: Jan Bergstra, Gilles Kahn, and Karst Koymans for their constructive remarks and the work they did. All others, especially Paul Klint and Jan Heering, who read the manuscript and commented on it, are thanked for their contribution.

My friends provided me with indispensable distractions whenever necessary. I am very grateful to two of them who contributed directly to the development of this thesis: Will van Hoof designed the beautiful cover, and Miranda Aldham-Breary read the entire manuscript and corrected its English. Erik van den Berg, Ton van Bortel, Marjo Janssen, Zweitse Klous, and Gertjan Koldenhof commented on early versions of the Dutch abstract.

The enthusiasm with which my niece Linda investigates her surrounding world is a source of inspiration for any researcher. It is great fun to play with her.

My brother Frans, his wife Marlie, my sister Lucy, and her husband Jan are not only close relatives, they are very good friends who supported me whenever needed.

My father and mother are the basis of all my activities. They have supported me in all circumstances and I am very proud to be able to dedicate this thesis to them. Sadly, my father will not be present when I am defending my thesis; his support and views on life will remain with me always.

Specification Formalisms

Short introductions to the Algebraic Specification Formalism ASF, the Syntax Definition Formalism SDF, and the combination of these formalisms ASF+SDF are given.

1.1 Introduction

In this chapter a short introduction to the main formalisms of importance in this thesis is given. The Algebraic Specification Formalism ASF [BHK89a] is described in Section 1.2, the Syntax Definition Formalism SDF [HK89b, HHKR89] in Section 1.3, and the combination of these formalisms is described in Section 1.4. To illustrate these formalisms several examples of specifications of natural numbers and (finite) sets of natural numbers are given.

1.2 Algebraic Specification Formalism - ASF

ASF [BHK89a] is a many-sorted, algebraic specification formalism similar to OBJ [GMP83] and its successors OBJ2 [FGJM85] and OBJ3 [GKKMMW88, KKM88], ACT-ONE [EM85], and PLUSS [Gau84, BCKSV88]. The general idea is to give a signature and a set of (conditional) equations over that signature. The signature consists of a set of sorts (just names) and a set of functions whose arguments and results are typed with sorts.

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

```

sorts NAT, SET-of-NAT

functions
  zero  :                               -> NAT
  succ  : NAT                           -> NAT
  plus  : NAT # NAT                     -> NAT
  mult  : NAT # NAT                     -> NAT
  empty :                               -> SET-of-NAT
  add   : NAT # SET-of-NAT              -> SET-of-NAT
  union : SET-of-NAT # SET-of-NAT -> SET-of-NAT

variables
  n, n1, n2 : -> NAT
  s, s1, s2 : -> SET-of-NAT

equations
[1]  plus(zero, n)      = n
[2]  plus(succ(n1), n2) = succ(plus(n1, n2))
[3]  mult(n, zero)      = zero
[4]  mult(n1, succ(n2)) = plus(mult(n1, n2), n1)
[5]  add(n, add(n, s))  = add(n, s)
[6]  add(n1, add(n2, s)) = add(n2, add(n1, s))
[7]  union(empty, s)    = s
[8]  union(add(n, s1), s2) = add(n, union(s1, s2))

```

Addition and multiplication on natural numbers are specified in equations [1] through [4]. The fact that `plus` is defined by induction on its first argument and `mult` by induction on the second one is irrelevant. This difference merely illustrates what happens when generating code from an algebraic specification as is illustrated in Section 2.4.2. Equation [5] expresses that adjacent identical elements in a set may be replaced by a single occurrence of the element. This has to be combined with equation [6] to allow arbitrary occurrences of identical elements in a set. Equation [6] states the irrelevance of the order in which the elements are added to the set. The equations [7] and [8] finally give the definition of the union operator on sets.

The meaning of an ASF-specification is its *initial algebra* [MG85]. An initial algebra is characterized by the fact that it contains *no junk* and *no confusion*. *No junk* means that it contains only elements that correspond to a *closed term* (a term without variables) over the signature of the specification. *No confusion* means that all closed terms in the specification which have an identical interpretation in the algebra can be proved equal using the equations. The initial algebra of a given specification is unique up to *isomorphism*.

It is always possible to construct the initial algebra of an algebraic specification. First of all, we take the free term algebra over the signature of the specification. The carriers of this algebra are the sets of all closed terms of the same sort. For each function f in the signature and for all closed terms of appropriate sorts t_1, t_2, \dots, t_n the interpretation for f is defined as $f(t_1, t_2, \dots, t_n)$. Next, a congruence relation is defined on this algebra by defining two closed terms to be equal if and only if their equality can be deduced from the set of (conditional) equations using many-sorted conditional equational logic [GM82] (see also Section 3.2.1). Finally, closed terms that are in the same congruence class are identified.

To generate an implementation for an algebraic specification it is viewed as a term rewriting system by interpreting each equation as a rewrite rule from left to right. A generated implementation can rewrite a closed term into its *normal form* (a closed term for which no rewrite rule is applicable). In the case of the above example, the term rewriting system is not *terminating* because equation [6] is infinitely applicable to a term which represents a set of natural numbers containing at least two different elements. When we restrict ourselves to sort `NAT` terms, the term rewriting system is terminating and *confluent*. The latter implies that the normal form of each term is unique. Both properties are desirable characteristics for term rewriting systems as they can then be used to decide equality of closed terms. If an ordering on natural numbers is specified, we can give a specification of (finite) sets of natural numbers whose corresponding term rewriting system is confluent and terminating by transposing two elements in a set only if they are not in order.

ASF also allows for modular division of the specification. It has several features to support this:

- *Exports:*
Each module may have an `exports` section consisting of a (possibly incomplete) signature. These sorts and functions are visible outside the module.
- *Hidden sorts and functions:*
Each module may have sections (labeled `sorts` and `functions`) containing declarations of hidden sorts and functions. These sorts and functions are only visible inside the module.
- *Imports:*
The `imports` section contains the names of modules that are used in a module. When importing a module it is possible to bind its parameters, to rename its signature (see below) or to perform a combination.
- *Parameters:*
In the `parameters` section, (possibly incomplete) signatures which are formal parameters of a module are declared. These parameters can be bound to actual

sorts and functions of a module when the parameterized module is imported.

- **Renamings:**

Upon import of a module parts of the signature of the module can be renamed if changes in names of sorts or functions are desirable to avoid, for instance, name clashes.

In [BHK89a] an extensive description of ASF is given and it also describes the *normalization strategy*, which defines how compound modules have to be evaluated in the context of the total specification in which they appear. The result of normalization is a module without imports, but some unbound parameters may remain. The semantics of a module in a specification is the initial algebra of its normal form if the latter contains no unbound parameters after normalization.

The modular structure of a specification is visualized in structure diagrams. Each module is represented by its name surrounded by a box (see, for example, Figure 1.1). A parameter of a module is represented by its name, surrounded by an ellipse, appearing at the upper side of the box (see Figure 1.2). The import of one module in another module is represented by a nested box (Figure 1.5). Binding of parameters is shown by drawing a line from the parameter to the actual module (Figure 1.3). If a parameter of a module is not bound upon its import in another module, a dotted line is drawn connecting the ellipses of both parameters as can be seen in Figure 6.9 of Section 6.4.2. Details like the signature, the equations, and renaming of the signature are not shown in these diagrams.

Normally, a specification of natural numbers and (finite) sets of natural numbers would be divided into three modules:

- a module `Natural-Numbers` in which the natural numbers are specified,
- a parameterized module `Sets`, and
- `Sets-of-Natural-Numbers` in which the parameter of `Sets` is bound to `Natural-Numbers`.

This would result in the following specification:

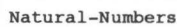


Fig. 1.1. Structure diagram of `Natural-Numbers`

```
module Natural-Numbers
begin
  exports
  begin
    sorts NAT
```

```

functions
  zero :          -> NAT
  succ : NAT      -> NAT
  plus : NAT # NAT -> NAT
  mult : NAT # NAT -> NAT
end

variables
  n, n1, n2 : -> NAT

equations
[1]  plus(zero, n)      = n
[2]  plus(succ(n1), n2) = succ(plus(n1, n2))
[3]  mult(n, zero)      = zero
[4]  mult(n1, succ(n2)) = plus(mult(n1, n2), n1)

end Natural-Numbers

```

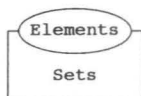


Fig. 1.2. Structure diagram of Sets

```

module Sets
begin

parameters
Elements begin
  sorts ITEM
end Elements

exports
begin
  sorts SET
  functions
    empty :          -> SET
    add   : ITEM # SET -> SET
    union : SET # SET  -> SET
  end
end

variables
  i, i1, i2 : -> ITEM
  s, s1, s2 : -> SET

equations
[5]  add(i, add(i, s)) = add(i, s)

```

```

[6]  add(i1, add(i2, s)) = add(i2, add(i1, s))
[7]  union(empty, s)      = s
[8]  union(add(i, s1), s2) = add(i, union(s1, s2))

end Sets

```

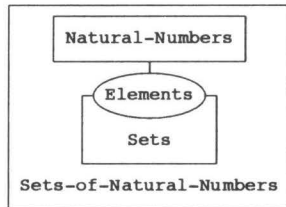


Fig. 1.3. Structure diagram of Sets-of-Natural-Numbers

```

module Sets-of-Natural-Numbers
begin
imports
  Sets
  { Elements bound by
    [ ITEM -> NAT ]
    to Natural-Numbers
    renamed by
    [ SET -> SET-of-NAT ] }
end Sets-of-Natural-Numbers

```

1.3 Syntax Definition Formalism - SDF

ASF has a simple, fixed syntax which permits the use of functions with fixed arity and of a limited form of unary and binary infix operators. It soon became obvious that a more liberal use of syntax would be convenient and would improve readability of the specifications. Moreover, in the context of writing language definitions, facilities for specifying syntax are clearly essential. Without them, major parts of each language definition have to deal with syntactic matters (as can be seen in [BHK89c]). The Syntax Definition Formalism SDF [HK89b, HHKR89] allows the user to define arbitrary context-free syntax. An SDF-specification defines both the concrete and abstract syntax of a language. SDF has been developed independently of any particular specification formalism. In principle, it can be combined with any specification formalism based on first-order signatures.

In the following example an SDF-specification of the syntax of natural numbers and sets of natural numbers is shown. The natural numbers (`Nat`) are defined as non-empty lists of the digits 0 through 9. Furthermore, the syntax of the successor function and two binary, associative infix operators for addition (+) and multiplication (*) is specified. The fact that multiplication has higher precedence than addition is expressed in the `priorities` section. Sets of natural numbers (`Set-of-Nat`) are specified as possibly empty lists of natural numbers separated by commas and surrounded by curly brackets. The union of sets is defined as a binary, associative infix operator `+`. On both sorts `bracket` functions are defined which specify which brackets may be used in the syntax of these sorts. Finally, the `variables` section contains the specification of the syntax of the variables which will be used later on, when adding semantics to this SDF-specification.

```

module Sets-of-Natural-Numbers

  sorts Nat Set-of-Nat

  lexical syntax
    [ \t\n] -> LAYOUT
    [0-9]+ -> Nat

  context-free syntax
    succ "(" Nat ")" -> Nat
    Nat "+" Nat -> Nat assoc
    Nat "*" Nat -> Nat assoc
    "(" Nat ")" -> Nat bracket
    "{" {Nat ","}* "}" -> Set-of-Nat
    Set-of-Nat "+" Set-of-Nat -> Set-of-Nat assoc
    "(" Set-of-Nat ")" -> Set-of-Nat bracket

  priorities
    Nat "+" Nat -> Nat < Nat "*" Nat -> Nat

  variables
    m -> CHAR*
    k [12]* -> CHAR+
    n [12]* -> Nat
    x [1-3] -> {Nat ","}*
    y [12] -> {Nat ","}+
```

An SDF-specification is a combined definition of the abstract syntax (in the form of a signature) and the concrete syntax (in the form of BNF-rules, read in reverse order) of a language. It consists of at most five components:

- The `sorts` section contains the names of the non-terminals of the grammar which can be derived from an SDF-specification. These names are also the names of the sorts in the derived signature.

- The `lexical syntax` section incorporates the specification of a regular grammar which is used to generate a lexical analyzer. It contains one or more function declarations each consisting of a regular expression and a result sort. The elements of the input stream which are to be skipped by the lexical analyzer are defined using the predefined sort `LAYOUT`. Character classes like `[0-9]` are used to abbreviate the lexical definition. A sort or character class followed by a `*` stands for zero or more repetitions of the sort. A `+` stands for one or more repetitions.
- The context-free grammar can be extracted from the `context-free syntax` section. Each rule in this section (except the functions which are furnished with the `bracket-attribute`) also adds information to the derived signature. The notations `{SORT "t"}*` and `{SORT "t"}+` are used to denote lists of elements of `SORT` separated by the symbol `"t"`. By extending signatures with `*` and `+` as described in Chapter 3 each rule will correspond to exactly one function in the derived signature.
- In the `priorities` section the precedence of the rules in the `context-free syntax` section can be specified in order to disambiguate sentences. In the above example it is specified that multiplication has higher priority than addition of natural numbers.
- The `variables` section defines the variables which may be used in the `equations` section when combining an SDF specification with an ASF specification as can be seen in Section 1.4. The variables of the predefined sort `CHAR` will be used to define equations in which lexical items have to be split up (see equations `[1]` through `[11]` of module `Natural-Numbers` in the following section).

1.4 The combined formalism ASF+SDF

The formalism ASF+SDF is now sketched. It is a language definition formalism intended for the definition of both syntax and semantics. Modules in this formalism are similar to ASF modules, except that the signature is replaced by an SDF-definition. The concrete syntax in the SDF-definition defines the syntactic form of the expressions which may be used in the equations. Conversely, specifications in the combined ASF+SDF formalism can be reduced to ASF specifications as follows:

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

Several specifications have been written in (preliminary versions of) ASF+SDF:

- The typechecker for a sublanguage of ML (Mini-ML) in Chapter 4 (see also [Hen89b]).
- The static and dynamic semantics of the toy language PICO [HK89a].
- The typechecker and interpreter for a simple programming language ASPLE, the dynamic semantics of the machine language SML, and a compiler from ASPLE to SML [Meu88].

In the design of ASF+SDF some minor changes/improvements have been incorporated which are not due to the mere combination of both formalisms:

- All *begin* and *end* keywords have been removed from the syntax of the formalism.
- Like the exports, the *hiddens* of a module are also wrapped in one section with keyword *hiddens*. The same is done with parameters: Each parameter is specified in an individual section which starts with the keyword *parameter* followed by the name of the parameter.
- Variables can be exported or can be declared in a parameter.
- The order of the different sections in the syntax part of a module (*imports*, *parameter*, *exports*, *hiddens*, and *priorities*) is free. It is even possible to double these sections. Such a duplication is identical to a declaration of the constituents in one section. The same freedom of order holds for the parts of an *exports*, *hiddens*, or *parameter* section (i.e. the *sorts*, *lexical syntax*, *context-free syntax*, and *variables* sections).
- In ASF a renaming is a list of name pairs without distinction between names of sorts or functions. In the combined formalism a renaming is split up in parts expressing the renaming of sorts, lexical syntax, context-free syntax, and variables.
- To prevent lengthy repetitions, context-free functions may be abbreviated to their *terminal skeleton* which is the list of terminals left of the \rightarrow -sign in their declaration. Such abbreviations may appear in renamings and priority declarations.
- In contrast to ASF, modules in which parameters with the same name occur more than once (either by multiple definition in the module itself or by importing unbound parameters) are no longer forbidden. The parts of such a parameter are now united as long as the origin rule (see [BHK89b] and Definition 6.3 of Section 6.2.2) is not violated.
- The sequence of modifiers (renamings and parameter bindings) in an import is no longer limited to a single renaming, a list of parameter bindings, a renaming

followed by parameter bindings, or parameter bindings followed by a renaming (for an example, see module `Signature-Typechecking` in Section 6.4.4).

As an example of ASF+SDF, once again, a specification of the natural numbers and (finite) sets of natural numbers is given. Starting with module `Layout` which is either directly or indirectly imported in all other modules of the specification. Consequently, it defines the layout of the whole specification. Space, tab (`\t`), and new-line (`\n`) are specified as layout-characters. These will be skipped when parsing equations and terms of a module.



Fig. 1.4. Structure diagram of `Layout`

```
module Layout
  exports
    lexical syntax
      [ \t\n] -> LAYOUT
```

Module `Natural-Numbers` contains the specification of natural numbers written in ordinary decimal notation. Equation [1] serves to remove leading zeros of numbers: it identifies, for instance, 007 with 7. The exported functions for addition and multiplication are defined in equations [2] through [5]. In these equations the hidden successor function is used, which is itself defined in the other equations. In equations [1], and [6] through [15] examples of the use of the function

```
nat "(" CHAR* ")" -> Nat
```

can be seen. Such functions are generated automatically for each output sort of a function in a `lexical syntax` section. They are used whenever the string of characters in a lexical item has to be analyzed in an equation. The natural number 007, for example, matches the left-hand side of equation [1]. It is of type `Nat`, begins with a zero, and the rest of the characters "07" matches the variable `k` of type `CHAR+`.

The term rewriting system corresponding to this specification is not confluent as the term `succ(0) + 0` can be rewritten to the normal forms `1 + 0` and `1`. Consequently, an implementation generated from this specification cannot be used to test equality of terms. It is possible, however, to give a specification of natural numbers whose term rewriting system is confluent and terminating. A naive specification (without auxiliary hidden functions) would need about two hundred equations in which the tables of addition and multiplication of digits are given. A more tricky

specification which uses auxiliary functions to specify these tables is given in the SDF reference manual [HHKR89].

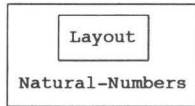


Fig. 1.5. Structure diagram of Natural-Numbers

```

module Natural-Numbers

imports Layout

exports
  sorts Nat
  lexical syntax
    [0-9]+ -> Nat
  context-free syntax
    Nat "+" Nat -> Nat assoc
    Nat "*" Nat -> Nat assoc
    "(" Nat ")" -> Nat bracket

priorities
  "+" < "*"

hiddens
  context-free syntax
    succ "(" Nat ")" -> Nat
  variables
    m      -> CHAR*
    k [12]* -> CHAR+
    n [12]* -> Nat

equations
[1]  nat("0" k) = nat(k)
[2]  0 + n      = n
[3]  succ(n1) + n2 = succ(n1 + n2)
[4]  n * 0      = 0
[5]  n1 * succ(n2) = n1 * n2 + n1
[6]  succ(nat(m "0")) = nat(m "1")
[7]  succ(nat(m "1")) = nat(m "2")
[8]  succ(nat(m "2")) = nat(m "3")
[9]  succ(nat(m "3")) = nat(m "4")
[10] succ(nat(m "4")) = nat(m "5")
[11] succ(nat(m "5")) = nat(m "6")
[12] succ(nat(m "6")) = nat(m "7")
  
```

```

[13] succ(nat(m "7")) = nat(m "8")
[14] succ(nat(m "8")) = nat(m "9")
[15] succ(nat(k1 "9")) = nat(k2 "0")
      when succ(nat(k1)) = nat(k2)

```

In the following module *Sets*, equation [16] removes identical elements in sets, and the irrelevance of the order of elements is expressed in [17]. Equation [18] gives an elegant definition of the union operator on sets. If an empty list of items is substituted for variables like *x1*, *x2*, and *x3* in equations [16] and [18], an adjacent comma is removed to retain a syntactically correct expression.

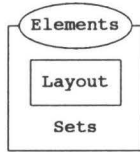


Fig. 1.6. Structure diagram of *Sets*

```

module Sets
  imports Layout
  parameter Elements
    sorts Item

  exports
    sorts Set
    context-free syntax
      "{" {Item "","}"* "}" -> Set
      Set "+" Set           -> Set assoc
      "(" Set ")"           -> Set bracket

  hiddens
    variables
      i      -> Item
      x [1-3] -> {Item "","}"*
      y [12]  -> {Item "","}" +

  equations
    [16] {x1, i, x2, i, x3} = {x1, x2, i, x3}
    [17] {y1, y2}           = {y2, y1}
    [18] {x1} + {x2} = {x1, x2}

```

Finally, in module *Sets-of-Natural-Numbers* *Sets* is imported and its parameter *Elements* is bound to the actual module *Natural-Numbers*. In the result of this parameter binding the sort *Set* is renamed to *Set-of-Nat*.

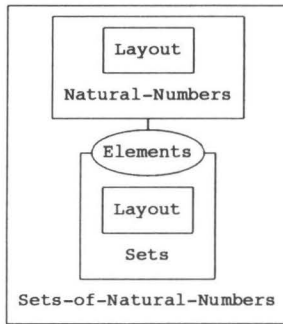


Fig. 1.7. Structure diagram of Sets-of-Natural-Numbers

```

module Sets-of-Natural-Numbers

imports
  Sets
  Elements bound by
    sorts Item => Nat
  to Natural-Numbers
  renamed by
    sorts Set => Set-of-Nat

```

1.5 Use of the specification formalisms in this thesis

The formalism ASF+SDF is used in Chapter 4 to specify a typechecker for a sublanguage of ML [HMM86, HMT87]. The specification of textual modularization given in Chapter 6 is written in this formalism.

Chapters 2 and 5 give descriptions of the two systems in which, respectively, ASF and ASF+SDF can be tested. SDF introduces associativity (the *assoc* attribute) and lists (the *** and *+* in variables and functions from the *context-free syntax* section). These features have consequences for the semantics of the ASF+SDF formalism. As a preparatory study, the extension of ASF itself with associativity and lists as well as its implementation is discussed in Chapter 3.

The ASF System

A simple, batch-oriented environment to compile and test ASF specifications is described. It consists of a parser and typechecker, a normalizer which removes the modular structure from specifications, a code generator which translates specifications to Prolog, a reduction machine which reduces input terms for a given specification, and simple tracing facilities for the reduction machine. An overview is given of the architecture of the ASF system and of the implementation techniques applied in it. Limitations of the system are discussed.

2.1 Introduction

The implementation of a modular algebraic specification formalism should provide for an environment in which specifications written in the formalism can be developed and tested. In [HKK89] a non-exhaustive catalog of available implementations of term rewriting systems is given. This chapter describes the global architecture of the ASF system which is a simple, batch-oriented system.

The ASF system processes specifications in the following, straightforward, manner. First, specifications are checked for syntactic and static semantic (i.e., typing) errors. If a specification passes this first phase, it is compiled into Prolog code. Finally, the resulting Prolog program is used to reduce given terms to normal form.

The ASF system contains the following components:

- a parser,
- a typechecker,
- a normalizer which removes the modular structure from the specification,
- a code generator which translates the specification to Prolog,
- a reduction machine which reduces input terms for a given specification, and
- simple tracing facilities for the reduction machine.

An earlier version of this chapter is part of the ASF system user's guide which was published as internal report [Hen88a]. An extended abstract of it was published in the proceedings of the SION conference CSN'88 [Hen88b].

The system is completely operational and can be used to compile specifications of reasonable size (more than 50 pages). The current implementation has been in use for the past few years for the development of many small and several large specifications. It has been ported to various machines (Vax, Sun, and Gould) and various institutes. The implementation consists of about 2600 lines of C-Prolog [PWBBP85], and 1600 lines of other supporting programs (e.g., C [KR78], LEX [LS86], and YACC [Joh86]).

The system has also been used to experiment with the implementation of new features added to ASF later on. Inequalities in conditions of equations as introduced in [HK89a] as well as rewriting module associativity and lists as described in Chapter 3 were implemented in it. These features were implemented as preparatory studies for the implementation of ASF+SDF (see Chapter 5). They will not be discussed in this chapter.

The user-level architecture of the ASF system is described in Section 2.2. Section 2.3 gives an example of how to use the system. Its internal structure is described in Section 2.4. Section 2.5 deals with the correctness and completeness of the implementation and Section 2.6 discusses some possible improvements of the implementation.

2.2 User-level architecture

At the user level, the ASF system has three commands:

- **asfcheck:**
Performs syntax checking and typechecking of an ASF specification.
- **asf:**
Performs syntax checking, typechecking, normalization and Prolog code generation.
- **asfex:**
Reduces a set of input terms using previously generated Prolog code. The reduction steps performed may be displayed by setting the trace option of **asfex**.

Strictly speaking, the **asfcheck** command is redundant. It only exists for reasons of efficiency. The user-level architecture of the ASF system is shown in Figure 2.1.

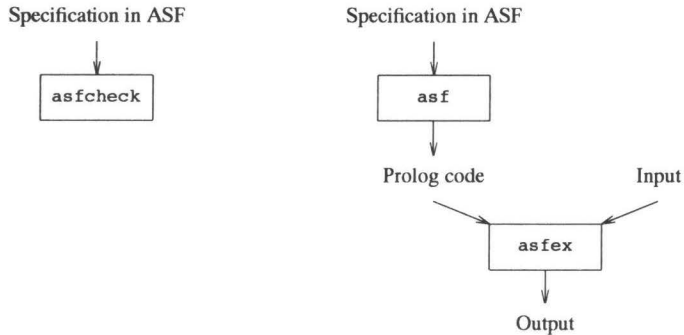


Fig. 2.1. User-level architecture of the ASF system

2.3 An example

To illustrate the system the specification of natural numbers and (finite) sets of natural numbers given in Section 1.2 is used. The fact that this specification is not terminating (see also Section 2.5.1) is irrelevant to its processing. Only if a term representing a set that contains at least two different natural numbers is evaluated, the implementation loops.

Assume that the specification of Section 1.2 resides in a file named `example.asf`. First, this specification is checked and compiled into Prolog code using the command:

```
asf example.asf
```

Now assume that the following modules with input terms reside on file `example.inp`:

```
module Natural-Numbers
begin
variables
  n1, n2 : -> NAT
terms
  [1]  mult(succ(zero), plus(succ(zero), zero))
  [2]  plus(succ(succ(n1)), n2)
end Natural-Numbers

module Sets-of-Natural-Numbers
begin
```

```

terms
  [1]   add(succ(zero), add(plus(zero, succ(zero)), empty))
  [2]   union(empty, add(mult(zero, succ(succ(zero))), empty))

end Sets-of-Natural-Numbers

```

The command

```
asfex example.inp
```

will check the input module and produce the following output:

```

module Natural-Numbers
begin
  [1]   mult(succ(zero), plus(succ(zero), zero))
        = succ(zero)
  [2]   plus(succ(succ(n1)), n2)
        = succ(succ(plus(n1, n2)))

end Natural-Numbers

module Sets-of-Natural-Numbers
begin
  [1]   add(succ(zero), add(plus(zero, succ(zero)), empty))
        = add(succ(zero), empty)
  [2]   union(empty, add(mult(zero, succ(succ(zero))), empty))
        = add(zero, empty)

end Sets-of-Natural-Numbers

```

in file `example.ex`. When the trace option of `asfex` is set, this output will also show the intermediate reduction steps.

2.4 Internal structure

The *internal* structure of the ASF system is shown in Figure 2.2. It consists of the following major components:

- `asfcheck`:
The typechecker of ASF acts on the abstract structure of a specification which is generated by a parser. The latter is available as `asfparse`.
- `asf`:
The same abstract structure is used in `asf` as input for the normalizer and the generator of Prolog code.

- **asfnorm:**
The normalizer removes the modular structure from all modules in the specification. Thus, in the output of the normalizer all imports have been removed and all renamings and parameter bindings have been carried out.
- **asfimpl:**
The generator of Prolog code adds type information to the equations and creates the specification dependent part of the code.
- **asfex:**
The generated code plus the code for a “reduction machine” (the specification independent part) constitute the complete code. The input for the code is first transformed to abstract structures using the input parser, which is available as `inpparse`.

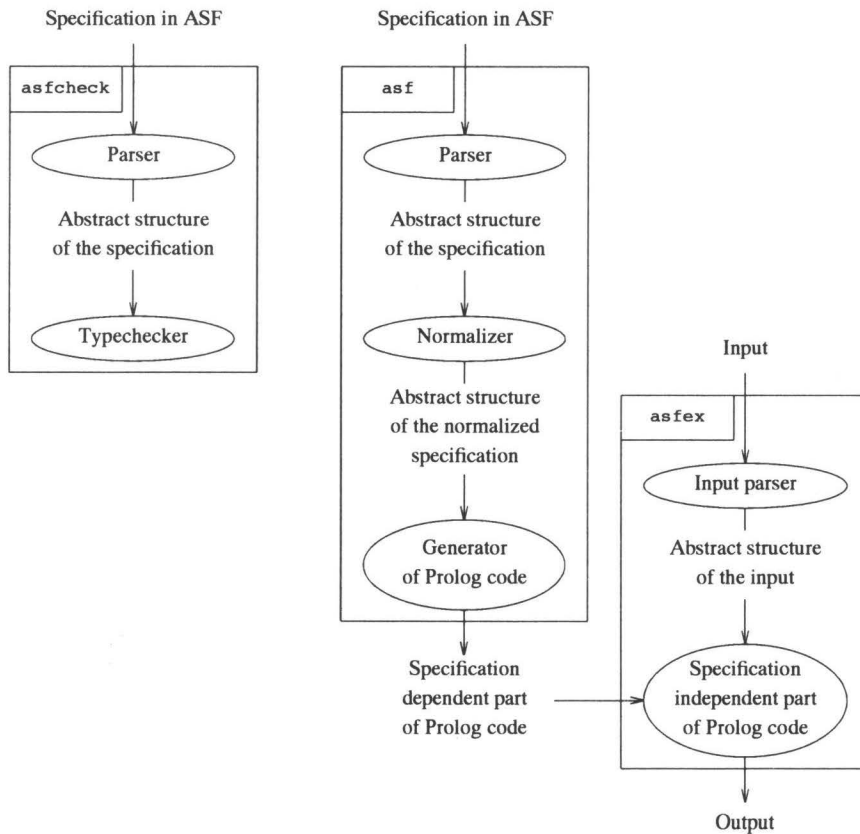


Fig. 2.2. Internal structure of the ASF system

2.4.1 Typechecking and normalization

Although typechecking and normalization may be considered to be different at the user level, they have a lot in common in their implementation. The typechecker has to do a great deal of normalization, because it must at least construct the visible signature (the combination of the export signature and the signatures of the parameters) of each module. Therefore, the typechecker and the normalizer have been implemented by a single program.

The normalizer generates the abstract structure of the normalized specification. This abstract structure is analogous to the one generated by the parser. The normalizer renames hidden sorts and functions when the module in which they are declared is imported into another module. The user of the system will only observe these renamings in traces of terms in which imported hidden functions occur. Function symbols and operators which consist of an identifier surrounded by dots are renamed by postfixing the identifier with a hyphen and a natural number. Operators which consist of sequences of one or more operator symbols are postfixes with one or more asterisks (*). This assures that the new function symbols and operators are legal ASF function symbols and operators. These automatic renamings are such that name clashes with other functions and operators are avoided.

2.4.2 Generation of Prolog code

In this section one of the possibilities to implement an algebraic specification is described. An algebraic specification is viewed as a term rewriting system by interpreting the conclusion of each equation as a rewrite rule from left to right. For a more general overview of implementation strategies of algebraic specifications see [BW89].

Several ways of implementing a term rewriting system in Prolog are known. In [DE84, EY87, Wie87] several methods are described in which Prolog predicates model a certain reduction strategy. In [DE84] Drosten and Ehrich give predicates for leftmost innermost reduction. The interpretational approach of van Emden and Yukawa [EY87] yields a parallel outermost strategy and Wiedijk gives a systematic overview of the different possibilities to implement a term rewriting system in Prolog in this manner [BW89, Wie87].

In contrast to the above-mentioned interpretational approaches, the ASF system generates faster code by using a variant of the compilational approach described in [EY87] (see also [HK89a]). This method corresponds naturally to the way in which one would implement functions in Prolog. It regards each function of the specification as a relation of its input and output. One of the major shortcomings of the compilational approach is its leftmost innermost reduction strategy. This may cause non-termination for terms that have a normal form.

The compilational approach of [EY87] has been modified because the original method can only handle specifications in which each equation is of the form $f(t_1, t_2, \dots, t_n) = t$ where t_1, t_2, \dots, t_n may not contain *defined functions*. These are functions which occur as main symbol in the left-hand side of an equation. In [EY87] Prolog predicates are generated only for defined functions. The implementation described here circumvents this constraint by generating predicates for all functions and adding a “catch all”-rule for each function.

For each n -ary function in the specification an $(n+1)$ -ary predicate and an n -ary function are created. The predicate represents the graph of the function: its first argument is the result of the application of the function to its arguments. The function represents the case of normal forms (i.e. irreducible terms). A catch-all rule is used to build the term if no equation is applicable. The following illustrates the generated code for the specification of (finite) sets of natural numbers as given in Section 1.2:

```

/* >>> Equations                                     <<< */

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

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

add(Res, I, add(I, S))                                /* [5] */
    :- add(Res, I, S).
add(Res, I1, add(I2, S))                              /* [6] */
    :- add(Tmp, I1, S),
       add(Res, I2, Tmp).

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

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

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

```
union(union(X1, X2), X1, X2).
```

In Section 2.6.1 an improved version of this code will be presented.

The generated code consists of two parts:

- *The specification dependent part:*
This part consists mainly of the Prolog code generated from the equations and the signatures of each (normalized) module. The signatures are needed to typecheck input modules. The equations are disambiguated to prevent incorrect use of equations due to overloading of function symbols.
- *The specification independent part:*
This part of the code consists of C-Prolog clauses for typechecking input modules, for reducing terms to normal form, for printing terms, and for creating traces of reductions.

The generator of Prolog code only creates the specification dependent part. Upon execution, this code is added to the specification independent part.

Before generating code for an equation the use of variables is checked. The evaluation of conditions is determined by the way in which they are used. Let \mathcal{V} be the set of variables used in the left-hand side of the conclusion of the equation. The conditions are checked in the order in which they are specified. There are two kinds of conditions that are allowed by the system:

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

Finally, it is checked to see that all variables in the right-hand side of the conclusion of the equation are members of the resulting set \mathcal{V} . Hence, an error-message is given if in both sides of a condition or in the right-hand side of an equation variables are used that have not been introduced before.

2.4.3 Parsing of specification and input modules

The ASF system contains two parsers: one for ASF specifications and one for input modules. Both parsers have been written in LEX [LS86], YACC [Joh86] and C [KR78]. They both transform their input into abstract structures represented by C-Prolog clauses.

The input for the generated code consists of a number of input-modules, each labeled with the name of a module and containing a set of terms. The module names state which equations may be used while reducing a term to normal form. The input should have the following concrete syntax:

```

<input>          ::= <input-module>+
<input-module> ::= module <module-ident>
                  begin
                      [ <variables> ]
                      [ <terms> ]
                  end <module-ident>
<terms>          ::= terms <tagged-term>+
<tagged-term>    ::= <tag> <term>

```

Where <module-ident>, <variables>, <tag> and <term> are defined as in [BHK89b]. Each input-module has two optional sections:

- *variables section:*
The variables used in the terms section of the module should be declared with their sort in this section. These sorts should of course be defined in the signature of the corresponding module in the specification after normalization.
- *terms section:*
This section contains the terms to be reduced to normal form. Terms are always typechecked before they are reduced.

2.4.4 Reduction of terms to normal form

Terms in an input-module are translated in the same manner as terms in specification modules. The term

```
mult(succ(zero), plus(succ(zero), zero))
```

will, for instance, be translated to the Prolog goals:

```

?- zero(Tmp1), succ(Tmp2, Tmp1),
   zero(Tmp3), succ(Tmp4, Tmp3), zero(Tmp5),
   plus(Tmp6, Tmp4, Tmp5),
   mult(Res, Tmp2, Tmp6).

```

In this way the arguments of a term are first reduced to normal form in left to right order. The standard Prolog interpretation ensures that the reduction machine searches for the first equation (in the normalized specification) whose left-hand side matches. If the equation at hand is a conditional one, first all conditions must be satisfied. The generated code is such that all subterms of the right-hand side of the equation are reduced to normal form before returning the result of evaluation. In the

above example, the normal form `succ(zero)` of the term to be reduced is the value of the variable `Res`.

If during the reduction of a term a conditional equation is encountered, its conditions are evaluated in the order in which they are specified. As mentioned before, the use of variables in a condition determines how it is evaluated.

It is also possible to reduce *open terms* (i.e., terms with variables) as long as no values are given to the variables occurring in such terms. A term like

```
plus(succ(succ(n1)), n2)
```

is translated into:

```
?- succ(Tmp1, n1), succ(Tmp2, Tmp1), plus(Res, Tmp2, n2).
```

The program will now return `succ(succ(plus(n1, n2)))`. Note, however, that variables are treated as constants in the reduction of an open term. This ensures that the Prolog code returns normal forms of open terms which are equationally provable from the equations given in the specification. In particular no induction on the structure of terms is used to deduce equality of terms.

When the trace option is set, each instance of an equation used in the reduction of the terms in the input is printed. This is achieved by Prolog clauses which simulate a Prolog interpreter of the generated code which has the side effect of printing the trace information. Doing this, the generated code does not have to be changed to provide the trace-option. This avoids the need to regenerate the code when the trace option is activated or deactivated.

2.5 Correctness and completeness

The generated Prolog code interprets the equations of the specification as rewrite rules for a (conditional) term rewriting system.

The generated interpretation is *correct*, i.e., for all (possibly open) terms t_1 and t_2 the following holds: if the implementation I returns t_2 as the result of evaluation of t_1 , then the equality of t_1 and t_2 can be proved using the equations \mathcal{E} of the specification. In short notation:

$$I \models t_1 \rightarrow t_2 \Rightarrow \mathcal{E} \vdash t_1 = t_2$$

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

More interesting is the question whether the converse (*completeness* of the implementation) holds. Or, more precisely, if two terms t_1 and t_2 are given such

that they can be proved equal using \mathcal{E} , the implementation can be used to show them to be equal:

$$\mathcal{E} \vdash t_1 = t_2 \Rightarrow \exists t \ I \models t_1 \rightarrow t \wedge I \models t_2 \rightarrow t ?$$

In general, this is too much to hope for, because it is impossible to decide whether an equation is derivable from a given set of equations. Incompleteness might be caused by non-termination of the implementation, non-confluence, and the inability to decide conditions. These properties will be treated briefly, for an extensive treatment see [Kap87]. In general, it is undecidable whether any of these three properties holds for a set of equations. Syntactic criteria exist, however, which ensure that the term rewriting system corresponding to a set of (conditional) equations is complete. Such criteria have not been implemented in the current version of the system.

2.5.1 Termination

It is very easy to write a set of equations which, when interpreted as rewrite rules, will not terminate. Some of the easiest examples are:

$$[1] \quad a = a$$

and commutative laws like:

$$[1] \quad x + y = y + x$$

Several articles [Kap87, JW87] investigate the use of *simplification orderings* to prove termination of term rewriting systems. A simplification ordering [Der87, Rus85] is a well-founded ordering $>$ on open terms such that:

- each term t is less than a term in which it occurs:

$$f(\dots, t, \dots) > t$$

- and the ordering preserves contexts:

$$t_1 > t_2 \Rightarrow f(\dots, t_1, \dots) > f(\dots, t_2, \dots)$$

In [DF85] a description is given of an algorithm that will construct a simplification ordering that proves termination of a term rewriting system or terminates in failure. It tries to construct a simplification ordering from a given set of equations by assuming that all terms in the conditions and the right-hand side of the conclusion of an equation have to be smaller than the left-hand side of the conclusion.

A general overview of the theory concerning termination of term rewrite systems is given in [Der87].

2.5.2 Confluence

A term rewriting system is confluent if for all terms t , t_1 and t_2 such that t reduces to t_1 as well as t_2 , there exists a term v such that both t_1 and t_2 reduce to v . In short:

$$t \rightarrow t_1 \wedge t \rightarrow t_2 \Rightarrow \exists v \ t_1 \rightarrow v \wedge t_2 \rightarrow v$$

An example of a specification of which the corresponding term rewriting system is not confluent is the following:

- [1] $a = b$
- [2] $a = c$

In such a specification the implementation is incapable of proving the equality of, for example, b and c .

It is possible to transform a given set of (conditional) equations into a confluent (and terminating) term rewriting system if a simplification ordering on terms is given. In [Kap87] a sketch of such a completion procedure is given which is improved in [JW87].

There exist syntactic criteria, like *regularity*, which ensure confluence. A term rewriting system is regular if it is *left-linear* (no variable occurs more than once in the left-hand side of an equation) and *non-ambiguous* (no two rules exist with overlapping left-hand sides). Such criteria are easy to check but they have not been implemented because they are overly restrictive.

2.5.3 Conditions

To implement conditional equations correctly it is necessary to be able to decide equations modulo a given set of equations (see also [Kap87, DOS88]). Again, confluence and termination are needed to find a solution of an equation. In

- [1] $a = b$
- [2] $a = c$
- [3] $a = c \implies d = e$

the generated code cannot reduce d to e because it cannot deduce the equality of a and c . The following example shows a set of equations for which the generated code will not terminate if a or c are to be reduced.

- [1] $a = b \implies c = d$
- [2] $c = d \implies a = b$

If a condition of an equation contains a variable which does not occur in the left-hand side of the conclusion, the implementation has to find a substitution for it which solves the condition. In Kaplan's article [Kap87] such conditions are forbidden. As mentioned before, our system is more liberal in allowing conditions in which variables may be introduced (see Section 2.4.2).

A warning is given if variables are not used in the way mentioned above, but this does not guarantee completeness of the implementation as becomes clear in the following example:

```

module Natural-Numbers
begin
  exports
begin
  sorts BOOL, NAT
  functions
    true  :          -> BOOL
    false :          -> BOOL
    0      :          -> NAT
    s      : NAT      -> NAT
    _+_    : NAT # NAT -> NAT
    lt     : NAT # NAT -> BOOL
end
variables
  x, y, z : -> NAT
equations
[1]    x + 0      = x
[2]    x + s(y) = s(x + y)
[3]    lt(x, x) = false
[4]    x + s(y) = z ==> lt(x, z) = true
[5]    x + s(y) = z ==> lt(z, x) = false
end Natural-Numbers

```

The generated code of this specification, for example, will show the term $lt(0, s(0))$ to be irreducible. The term matches with the left-hand side of the conclusion of [4], after which the term substituted for z (in this case: $s(0)$) is reduced and unified with $0 + s(y)$. This unification fails. Next, equation [5] is examined, but in this case the terms 0 and $s(0) + s(y)$ have to be unifiable. No further equation is applicable and hence the term is irreducible.

2.6 Possible improvements

The ASF system described here is completely operational and can be used to compile specifications of reasonable size (e.g. 50 pages). There are, however, many potential improvements to the system, which will now be discussed briefly.

2.6.1 Optimization of generated code

In the example of sets of natural numbers, the generated code (see Section 2.4.2) could be simplified by observing that the functions `zero`, `succ`, and `empty` are never used as head symbol in the left-hand side of an equation. Hence, there are no clauses for the predicates of these functions except for the catch-all rules. As a consequence, the generated code could be simplified to:

```

/* >>> Equations                                     <<< */

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

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

add(Res, I, add(I, S))                                /* [5] */
    :- add(Res, I, S).
add(Res, I1, add(I2, S))                             /* [6] */
    :- add(Tmp, I1, S),
       add(Res, I2, Tmp).

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

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

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

It would be possible to do without the catch-all rules for `plus`, `mult`, and `union` if only closed expressions were to be reduced. This same information could also be used in the decomposition of an input term into Prolog goals. The goal

```

?- plus(Tmp, succ(zero), zero),
   mult(Res, succ(zero), Tmp).
```

would now be the result of decomposition of the term

```
mult(succ(zero), plus(succ(zero), zero)).
```

This simplification has not been implemented, because it requires global information from the specification. As in the example of natural numbers and natural numbers modulo 2 it can be useful to specify a function (i.e., the successor) without

any equations in one module and import this module in another one in which equations are added for the function. Using such global information is difficult to reconcile with our desire to achieve a modular implementation of ASF.

2.6.2 Optimization of the reduction machine

It frequently occurs that the same (sub)term is reduced more than once during the reduction of a given input term. Such repeated reductions can be avoided by storing terms and their computed normal form in a database. Before reducing a term, the reduction machine can consult the database to see whether or not it has been reduced previously and the stored normal form can be used. Of course, some balance will have to be found between storing all intermediate results and recomputing them. Some simple experiments show that these techniques might lead to substantial savings in execution time.

2.6.3 Normalization versus modular compilation

In the ASF system the modular structure of the specification is not reflected in the generated code.

The user is often interested in just one module of his specification, but in the current system code is generated for all modules. Each module is normalized and code is generated for it independently of the code generated for the other ones. This will not only increase the compilation time of specifications, but also the size and execution time of generated code.

Whereas the user will often only change a few modules, the current ASF system will completely typecheck, normalize and generate code for all modules in the specification including the ones that have not been changed. A modular implementation would only process the changed modules.

In the ASF+SDF system (see Chapter 5) the problems of recompiling an entire specification after each modification, and of modular generation of code are tackled:

- It is an interactive system meaning that a user can develop and test a specification incrementally. The generated implementation is updated after each editing operation on the specification.
- Modular generation of code is *not* implemented in the ASF+SDF system. Instead of generating code for individual modules and combining the code of several modules to construct the code of a specific module in which they are imported, code is generated for the complete specification and, if needed, appropriate subparts of it are selected representing the code of a specific module.

Extending ASF with Associativity and Lists

As a preparatory study for the integration of ASF and SDF, the problem of how (syntactic) list constructs in SDF are integrated into an algebraic specification formalism is considered. To this end, ASF is extended with binary associative operators and list constructors. A formal description of both extensions is given and it is shown how they can be translated to Prolog code.

3.1 Introduction

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

- natural numbers as lists of one or more digits:

```
nat : DIGIT+ -> NAT,
```

- tables which are a list of zero or more pairs of keys and their corresponding entries:

```
pair  : KEY # ENTRY -> PAIR
table : PAIR*       -> TABLE, and
```

- programs, in a simple programming language, which are defined as a list of one or more declarations followed by a list of zero or more statements:

```
prog : DECLARATION+ # STATEMENT* -> PROGRAM.
```

Such lists of terms of the same sort are, of course, definable in standard algebraic specification formalisms. Consequently, these list operations will not add

An earlier, and extended version of this chapter was published as internal report [Hen89a].

expressive power to the formalism. Use of these lists improves readability of specifications in many cases (see Section 3.4.1 for an example) and can be used to generate specialized code (see Section 3.4.4).

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

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

Using this representation, it is easy to specify how to remove leading zeros from a natural number, but an auxiliary function is needed to access the last digit of the list to express that the successor ($\text{succ} : \text{NAT} \rightarrow \text{NAT}$) of a natural number ending in 1 is identical to the same list of digits ending in 2.

Concatenation of lists is an associative binary operation. As a consequence, the semantics of algebraic specifications with lists can be expressed in terms of algebraic specifications with associativity. For this reason, algebraic specifications with associative binary operators and their implementation in terms of rewriting modulo associativity are discussed first. Associativity of a binary function is denoted by adding the *assoc*-attribute to it. This predicate is also available in the specification languages *AXIS* [CACDHGGR88, RC88], *CEC* [BGS88a, BGS88b], *OBJ2* [FGJM85], and its successor *OBJ3* [GKKMMW88, KKM88].

A translation of an algebraic specification with lists to a specification with associative operators is given in Section 3.4.3. It is not practical to use such a translation to generate code. One, it is necessary to double equations if a confluent and terminating rewriting system modulo lists is to be translated into a rewriting system modulo associativity with the same properties. Two, the translation will give superfluous code because it will try to apply a rewrite rule to each sublist of the list of arguments of an associative operator. The lists as proposed here are such that their semantics cannot be changed and hence we know in advance that there are no rewrite rules for the concatenation operator.

Before describing the extensions of an algebraic specification formalism with associativity and list operations in, respectively, Sections 3.3 and 3.4, a general scheme to generate an implementation for algebraic specifications is given in Section 3.2. Section 3.5 contains conclusions and some remarks.

3.2 Algebraic specifications

3.2.1 Definitions

An *algebraic specification* $\langle \Sigma, \mathcal{E} \rangle$ consists of a signature Σ and a set of (possibly conditional) equations \mathcal{E} . A *signature* $\Sigma \equiv \langle S_\Sigma, \mathcal{F}_\Sigma \rangle$ consists of a set of sort symbols S_Σ and a set of function symbols \mathcal{F}_Σ . An implicit typing function of \mathcal{F}_Σ to $S_\Sigma^* \times S_\Sigma$ exists which assigns to each element f of \mathcal{F}_Σ an input type $s_1 \# s_2 \# \dots \# s_n$ where $n \geq 0$ and an output type s . Such a function will be denoted by

$$f : s_1 \# s_2 \# \dots \# s_n \rightarrow s.$$

In most formalisms, overloading of function symbols is allowed, i.e., more than one typing of a function symbol $f \in \mathcal{F}_\Sigma$ is possible. To assure unique typing of *each* term it is essential that no functions with identical name and input type exist. To simplify the theoretical description overloading is forbidden as this can be remedied by encoding the type information in the function names.

Let a set of typed variables \mathcal{X} be given, i.e., to each variable $x \in \mathcal{X}$ a unique type $s \in S_\Sigma$ is attached and this will be denoted by $x : \rightarrow s$. Given a signature $\Sigma = \langle S_\Sigma, \mathcal{F}_\Sigma \rangle$ the set $\mathcal{T}_\Sigma^s(\mathcal{X})$ of terms of type s over Σ can be defined as the smallest set such that:

- x is an element of $\mathcal{T}_\Sigma^s(\mathcal{X})$ for each variable $x : \rightarrow s$.
- c is an element of $\mathcal{T}_\Sigma^s(\mathcal{X})$ for each (constant) function $c : \rightarrow s$.
- For each function $f : s_1 \# s_2 \# \dots \# s_n \rightarrow s$ with $n \geq 1$ and for all terms $t_1 \in \mathcal{T}_\Sigma^{s_1}(\mathcal{X})$, $t_2 \in \mathcal{T}_\Sigma^{s_2}(\mathcal{X})$, \dots , $t_n \in \mathcal{T}_\Sigma^{s_n}(\mathcal{X})$ $f(t_1, t_2, \dots, t_n)$ is defined as element of $\mathcal{T}_\Sigma^s(\mathcal{X})$.

The set of *closed terms* (terms without variables) of type s is denoted by \mathcal{T}_Σ^s . The set of terms $\mathcal{T}_\Sigma(\mathcal{X})$ over Σ is the union of $\mathcal{T}_\Sigma^s(\mathcal{X})$ for all $s \in S_\Sigma$.

An *unconditional equation* of type $s \in S_\Sigma$ over a given signature $\Sigma = \langle S_\Sigma, \mathcal{F}_\Sigma \rangle$ and a given set of variables \mathcal{X} is an element of $\mathcal{Eq}^s \equiv \mathcal{T}_\Sigma^s(\mathcal{X}) \times \mathcal{T}_\Sigma^s(\mathcal{X})$. It is denoted by $s = t$ where $s, t \in \mathcal{T}_\Sigma^s(\mathcal{X})$. The set of all (possibly conditional) equations \mathcal{E} in an algebraic specification $\langle \Sigma, \mathcal{E} \rangle$ is a subset of $\mathcal{Eq} \times \mathcal{Eq}^*$, where \mathcal{Eq} denotes the set of unconditional equations $\mathcal{Eq} \equiv \bigcup_{s \in S_\Sigma} \mathcal{Eq}^s$. Conditional equations with at least one

condition are denoted by

$$s = t \text{ when } s_1 = t_1, s_2 = t_2, \dots, s_n = t_n,$$

$$s_1 = t_1, s_2 = t_2, \dots, s_n = t_n \implies s = t, \text{ or}$$

$$\begin{array}{l}
s_1 = t_1, s_2 = t_2, \dots, s_n = t_n \\
\hline
s = t.
\end{array}$$

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

- For each (constant) function $c : \rightarrow s$ we define $\rho(c) \equiv c$.
- For all functions $f : s_1 \# s_2 \# \dots \# s_n \rightarrow s$ with $n \geq 1$ ρ is defined by

$$\rho(f(t_1, t_2, \dots, t_n)) \equiv f(\rho(t_1), \rho(t_2), \dots, \rho(t_n)).$$

If there is at least one closed term for each sort (see [MG85]), the axioms and rules of (conditional) equational logic can be given:

$$\frac{(s = t) \in \mathcal{E}}{\mathcal{E} \vdash s = t} \quad (\text{Eq1})$$

$$\mathcal{E} \vdash t = t \quad (\text{Eq2})$$

$$\frac{\mathcal{E} \vdash t_1 = t_2}{\mathcal{E} \vdash t_2 = t_1} \quad (\text{Eq3})$$

$$\frac{\mathcal{E} \vdash t_1 = t_2 \quad \mathcal{E} \vdash t_2 = t_3}{\mathcal{E} \vdash t_1 = t_3} \quad (\text{Eq4})$$

$$\frac{\mathcal{E} \vdash s_1 = t_1 \quad \mathcal{E} \vdash s_2 = t_2 \quad \dots \quad \mathcal{E} \vdash s_n = t_n}{\mathcal{E} \vdash f(s_1, s_2, \dots, s_n) = f(t_1, t_2, \dots, t_n)} \quad (\text{Eq5})$$

$$\frac{\mathcal{E} \vdash s = t}{\mathcal{E} \vdash \rho(s) = \rho(t)} \quad (\text{Eq6})$$

$$\frac{\mathcal{E} \vdash \rho(s_1) = \rho(t_1) \quad \dots \quad \mathcal{E} \vdash \rho(s_n) = \rho(t_n) \quad (s = t \text{ when } s_1 = t_1, \dots, s_n = t_n) \in \mathcal{E}}{\mathcal{E} \vdash \rho(s) = \rho(t)} \quad (\text{C-Eq})$$

which holds for all terms $s, t, s_1, s_2, \dots, s_n, t_1, t_2, \dots, t_n \in \mathcal{T}_{\Sigma}(\mathcal{X})$; for all functions $f \in \mathcal{F}_{\Sigma}$ and for all assignments $\rho : \mathcal{X} \rightarrow \mathcal{T}_{\Sigma}(\mathcal{X})$. Axioms and rules (Eq1) through (Eq6) together constitute equational logic. Rule (C-Eq) handles conditional equations.

3.2.2 Implementation in Prolog

In this section the algorithm of how to generate Prolog code for the implementation of an algebraic specification is described. Section 2.4.2 is referred to as an example of the generated code. In the next section specifications with unconditional equations are discussed first, followed by the description of the implementation of equations with conditions in Section 3.2.2.2.

3.2.2.1 Implementation of equations without conditions

How can the code for an algebraic specification such as that in Section 2.4.2 be generated? Each equation is typechecked before a Horn-clause is generated for it. During typechecking the use of variables in the equation is checked. It is impossible to generate code if the left-hand side of an equation is a variable or if the right-hand side contains variables that do not occur in the left-hand side. See the next section for conditional equations in which case the latter may not be necessary. At the same time, a list of variables and their corresponding Prolog variables which are to be used in the code is constructed.

The code generation process itself consists of two separate parts:

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

These steps are now described in somewhat more detail.

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

- $t \equiv x$:
The list of predicates to be generated is empty and the translated term is the Prolog variable that corresponds to the variable x .
- $t \equiv c$:
The translated term of a constant c is a “fresh” Prolog variable var (i.e., a Prolog variable that has not been assigned to any of the variables in the equation and which has not yet been used in the code generation process). The list of predicates contains just one element: the predicate $c(\text{var})$.
- $t \equiv f(t_1, t_2, \dots, t_n)$ with $n \geq 1$:
The translated term of t is, once again, a “fresh” Prolog variable var . Let L_1, L_2, \dots, L_n be the lists of predicates, respectively, generated for the subterms t_1, t_2, \dots, t_n , and, let T_1, T_2, \dots, T_n be the translated terms corresponding to these subterms. The list of predicates for t is a concatenation of the lists $L_1,$

L_2, \dots, L_n , with the predicate

$f(\text{Var}, T_1, T_2, \dots, T_n)$

added at the end of it.

In step 2 the conclusion of the Horn-clause is generated: if the left-hand side of the equation is of the form $f(t_1, t_2, \dots, t_n)$ with $n \geq 1$ the conclusion is

$f(\text{Res}, T_1, T_2, \dots, T_n)$,

where Res is the translated term from the right-hand side of the equation and the T_i are constructed from t_i by changing each variable into the corresponding Prolog variable. If the left-hand side of the equation is a constant c the conclusion is the predicate $c(\text{Res})$.

Finally, for each function from the specification the catch-all rule is added. It consists for each n -ary function symbol f with $n \geq 1$ of the Horn-clause

$f(f(x_1, x_2, \dots, x_n), x_1, x_2, \dots, x_n)$.

For constants c the catch-all rule is $c(c)$.

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

3.2.2.2 Implementation of conditions

As mentioned in Section 2.4.2 the evaluation of conditions is determined by the way variables are used in the corresponding equation. How is the appropriate code generated for a conditional equation? For each condition a list of Prolog predicates is generated and these lists are concatenated in the order in which the conditions are given in the equation. The list of predicates constructed in this way is added before the list constructed from the right-hand side of the conclusion of the equation, as described in the previous section.

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

1. Both sides of the condition $t_l = t_r$ will be decomposed in a list of predicates and a translated term as described in Section 3.2.2.1. Let L_l with T_l and L_r with T_r , respectively, be the lists of predicates and the translated terms for t_l and t_r . The code for this condition is a concatenation of L_l and L_r (in an arbitrary order) followed by testing the literal equality of T_l and T_r : $T_l == T_r$.

2. Suppose the condition is $t_c = t_n$, where t_c is the side of the condition which contains only variables which were known and t_n contains some new variables. Now t_c is decomposed into a list of predicates L_c and a translated term T_c as described in Section 3.2.2.1. All variables occurring in t_n are changed into their corresponding Prolog variable resulting in a Prolog term T_n . The code is the list L_c followed by a unification of T_c with T_n : $T_c = T_n$.

3.3 Algebraic specifications with associativity

3.3.1 Example

To illustrate associativity in an algebraic specification, the example given in Section 1.2 is changed by using the *assoc*-attribute to declare the associativity of the addition and multiplication on natural numbers and the union operator on sets:

```
...
plus : NAT # NAT -> NAT {assoc}
mult : NAT # NAT -> NAT {assoc}
...
union : SET # SET -> SET {assoc}
...
```

The equations of the specification are not changed.

3.3.2 Definitions

An *algebraic specification with associativity* $\langle \Sigma, \mathcal{E}, \text{assoc} \rangle$ consists of an algebraic specification $\langle \Sigma, \mathcal{E} \rangle$ and a predicate *assoc* defined on the set of function symbols \mathcal{F}_Σ . Only associativity for functions of the form

$$f : S \# S \rightarrow S$$

is described. For functions with other typings it is either impossible to give a semantics for associativity or it is unclear what its meaning should be as the following example shows. Given a function $f : S_1 \# S_2 \# S_1 \rightarrow S_1$ (with $S_1 \neq S_2$) associativity could very well stand for the equation

$$f(x_1, y_1, f(x_2, y_2, x_3)) = f(f(x_1, y_1, x_2), y_2, x_3)$$

where $x_1, x_2, x_3 : \rightarrow S_1$ and $y_1, y_2 : \rightarrow S_2$. But, for a function $g : S \# S \# S \rightarrow S$ it is questionable whether it would have to stand for the equations

$$\begin{aligned}
&g(z1, z2, g(z3, z4, z5)) = \\
&g(z1, g(z2, z3, z4), z5) = \\
&g(g(z1, z2, z3), z4, z5)
\end{aligned}$$

where $z1, z2, z3, z4, z5 : \rightarrow S$.

3.3.3 Semantics

The semantics of an algebraic specification with associativity $\langle \Sigma, \mathcal{E}, assoc \rangle$ is defined as the semantics of the algebraic specification $\langle \Sigma, \mathcal{E}' \rangle$, where \mathcal{E}' is constructed by adding the corresponding associative law to the set of equations \mathcal{E} for each associative function. Hence, for each function $f : S \# S \rightarrow S$ for which $assoc(f)$ holds, the equation

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

is added (where $x, y, z : \rightarrow S$).

3.3.4 Implementation in Prolog

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

$$f(x, f(y, z)) \rightarrow f(f(x, y), z)$$

and

$$f(f(x, y), z) \rightarrow f(x, f(y, z)).$$

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

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

When rewriting modulo associativity, the following three complications have to be considered:

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

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

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

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

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

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

assoc_all(_, [Term], Term)
:- !.

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

```

        assoc_arg(Name, Res, L2_New),
        split(L1, L2_New, L3, Input_New),
        assoc_all(Name, Input_New, Result).
assoc_all(Name, Input, Result)
    :- Result =.. [Name, Input].

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

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

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

```

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

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

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

The code generated for the specification of natural numbers and finite sets of natural numbers given in Section 3.3.1 is the following:

```

/* >>> Equations                                     <<< */

plus(N, [zero| N_List])                               /* [1] */
:- assoc_arg(plus, N, N_List).

plus(Res, [succ(N1)| N2_List])                         /* [2] */
:- assoc_arg(plus, N2, N2_List),
   assoc_flat(plus, [N1, N2], List),
   assoc_all(plus, List, Tmp),
   succ(Res, Tmp).

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

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

add(Res, I, add(I, S))                                /* [5] */
:- add(Res, I, S).

add(Res, I1, add(I2, S))                              /* [6] */
:- add(Tmp, I1, S),
   add(Res, I2, Tmp).

union(S, [empty| S_List])                             /* [7] */
:- assoc_arg(union, S, S_List).

union(Res, [add(I, S1)| S2_List])                     /* [8] */
:- assoc_arg(union, S2, S2_List),
   assoc_flat(union, [S1, S2], List),
   assoc_all(union, List, Tmp),
   add(Res, I, Tmp).

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

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

```

The generation of code for an algebraic specification with associativity uses an extension of the method described in Section 3.2.2 for standard algebraic specifications. The only difference as far as typechecking the specification is concerned is the flattening of terms which is done in this phase. Extensions of the two steps

defined in Section 3.2.2.1 give the code generation of one Horn-clause for each equation:

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

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

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

- $t \equiv f(t_1, t_2, \dots, t_n)$ with $n \geq 2$ and $\text{assoc}(f)$:
Let L_1, L_2, \dots, L_n be the lists of predicates generated for the subterms t_1, t_2, \dots, t_n , and let T_1, T_2, \dots, T_n be the corresponding translated terms. The list of predicates for t is a concatenation of the lists L_1, L_2, \dots, L_n , and the predicates

```
assoc_flat(f, [T1, T2, ..., Tn], List) and
assoc_all(f, List, Var)
```

added at the end of it. The variables `List` and `Var` are both fresh Prolog variables, and `Var` is the translated term for t .

The treatment of the left-hand side of the equation is not as easy as in Section 3.2.2.1. A clear distinction between the handling of the head symbol of the left-hand side and the handling of its arguments is necessary.

- 2a. A corresponding Prolog term has to be generated for each argument. This Prolog term is called the *matching term* of the argument. The variables in the arguments will be represented by their corresponding Prolog variables. Care must be taken that their value after using Prolog unification and resolution of generated `assoc_decomp` and `assoc_arg` predicates, is the term which the original variable would have had after matching modulo associativity.
- 2b. The conclusion of the Horn-clause is constructed from the head symbol of the left-hand side and the matching terms of its arguments. If the head symbol is an associative operator the matching terms of the arguments have to be put in a Prolog list as the second argument of the predicate.

In case 2a the matching term in the standard code generation process was simply created by changing all variables into their corresponding Prolog variable. Now the

matching term and a list of `assoc_decomp` and `assoc_arg` predicates is defined for each term using induction on the complexity of the term t :

- $t \equiv x$:
The list of predicates is empty and the matching term is the Prolog variable that corresponds to the variable x .
- $t \equiv c$:
The matching term of a constant c is c and the list of predicates is empty.
- $t \equiv f(t_1, t_2, \dots, t_n)$ with $n \geq 1$ and not `assoc(f)`:
The matching term of t is $f(T_1, T_2, \dots, T_n)$, where T_1, T_2, \dots, T_n are the matching terms of t_1, t_2, \dots, t_n . The list of predicates for t is simply a concatenation of the lists for t_1, t_2, \dots, t_n .
- $t \equiv f(t_1, t_2, \dots, t_n)$ with $n \geq 2$ and `assoc(f)`:
The matching term and the list of predicates for the arguments $a \equiv [t_1, t_2, \dots, t_n]$ and the associative operator f are created as follows:
 - $a \equiv [t_1, t_2, \dots, t_n]$ with $n \geq 2$:
Let the matching term of $[t_2, \dots, t_n]$ be Tr , and let the list of predicates be L_r .
 - If t_1 is a variable x and the Prolog variable which corresponds to x is x , then the list of predicates for a is L_r with

$$\text{assoc_decomp}(f, \text{Var}, x, Tr)$$
 added at the end. Here Var is a fresh Prolog variable which is also the matching term of a .
 - If t_1 is not a variable and the matching term for t_1 is T_1 and the list of predicates is L_1 , then the list of predicates for a is a concatenation of L_r and L_1 . The matching term for a is $[T_1 \mid Tr]$.
 - $a \equiv [t_1]$:
 - If t_1 is a variable x , the list of predicates for a is

$$\text{assoc_arg}(f, x, \text{Var})$$
 where x is the Prolog variable which corresponds to x . The matching term of a is a fresh Prolog variable Var .
 - If t_1 is not a variable and the matching term for t_1 is T_1 and the list of predicates is L_1 , then the list of predicates for a is L_1 and the matching term for a is $[T_1]$.

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

- $t \equiv c$:
If the left-hand side is a constant c the conclusion is $c(Res)$.
- $t \equiv f(t_1, t_2, \dots, t_n)$ with $n \geq 1$ and *not* $assoc(f)$:
The conclusion is $f(Res, T_1, T_2, \dots, T_n)$, where the T_i are the terms which correspond to the arguments t_i as defined in step 2a.
- $t \equiv f(t_1, t_2, \dots, t_n)$ with $n \geq 2$ and $assoc(f)$:
The conclusion is $f(Res, Tr)$, where the Tr is the matching term which corresponds to the list of arguments $[t_1, t_2, \dots, t_n]$ as defined in step 2a.

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

Finally, the decomposition of input terms to Prolog questions and the handling of conditional equations is similar to that in Section 3.2.2.

3.4 Algebraic specifications with lists

3.4.1 Example

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

```

module Natural-Numbers
begin
  exports
begin
  sorts DIGIT, NAT, SET
  functions
    0      :          -> DIGIT
    ...
    9      :          -> DIGIT
    nat    : DIGIT+    -> NAT
    succ   : NAT       -> NAT
    set    : NAT*      -> SET
    union  : SET # SET -> SET
end

```

```

variables
  k, k1, k2  : -> DIGIT+
  m          : -> DIGIT*
  n          : -> NAT
  x1, x2     : -> NAT+
  y1, y2, y3 : -> NAT*

equations

[1]  nat([0, k]) = nat([k])
[2]  succ(nat([m, 0])) = nat([m, 1])
    ...
[10] succ(nat([m, 8])) = nat([m, 9])
[11] succ(nat([9]))    = nat([1, 0])
[12] succ(nat([k1, 9])) = nat([k2, 0])
    when succ(nat(k1)) = nat(k2)

[13] set([y1, n, y2, n, y3]) = set([y1, n, y2, y3])
[14] set([x1, x2])           = set([x2, x1])
[15] union(set(y1), set(y2)) = set([y1, y2])

end Natural-Numbers

```

3.4.2 Definitions

An *algebraic specification with lists* $\langle \Sigma, \mathcal{E} \rangle$ consists of an extended signature Σ and a set of (possibly conditional) equations \mathcal{E} over Σ . An *extended signature* $\Sigma \equiv \langle \mathcal{S}_\Sigma, \mathcal{F}_\Sigma \rangle$ contains a set of sort symbols \mathcal{S}_Σ and a set of function symbols \mathcal{F}_Σ . Unlike the typing function in standard signatures, the typing function in extended signatures may also use “starred” and “plussed” sorts in its *input type*. Hence, the implicit typing function is now defined from \mathcal{F}_Σ to $\{s, s^*, s+ \mid s \in \mathcal{S}_\Sigma\}^* \times \mathcal{S}_\Sigma$. Sorts of the form s^* and $s+$ are called *iterated sorts*.

The user of the specification formalism is prevented from changing the semantics of iterated sorts. To this end, the use of these sorts as the output sort of functions, and also as the sort of any equation in a specification is forbidden. There are several reasons for these restrictions:

- The names s^* and $s+$ suggest that these sorts contain only iterations of elements of sort s and none of these lists can be identified. In the context of a modular algebraic specification formalism it may cause confusion if two lists are equal as a consequence of the import of a module in which those lists are identified.
- It is questionable whether an equation over terms of an iterated sort should be applicable to sublists of lists. Suppose for example the sort NAT was removed from the above specification and replaced by $\text{DIGIT}+$. The declaration of the

function `nat` would then disappear and equation [1] would become $[0, k] = [k]$. This equation is only correct if it is not applicable to sublists of lists for then it would also remove the zeros in $[1, 0, 0, 4]$. If an equation is applicable to sublists of lists, it is even necessary to introduce the extra sort `NAT` to express removal of leading zeros.

- It is easier to implement algebraic specifications with lists if the semantics of lists cannot be modified.

I am perfectly aware that none of the above reasons gives a strict argument for the choice to forbid changing the semantics of iterated sorts. It just gives a clearer formalism if the data types of iterated sorts are separated from those in which lists are identified. Meanwhile, these constraints do not restrict the expressive power of algebraic specifications with lists and their implementation by rewriting modulo lists. As in the above example of natural numbers as lists of digits, an extra sort in which the identification of lists of digits is expressed can always be introduced.

Given an extended signature $\Sigma = \langle \mathcal{S}_\Sigma, \mathcal{F}_\Sigma \rangle$ and a set of typed variables X , the set of terms over such a signature can be defined. As can be seen from the above example variables are allowed to be of an iterated sort. In the sequel, $S, S1, S2, \dots$ are used to denote the usual sorts of the specification (the elements of \mathcal{S}_Σ) and $T, T1, T2, \dots$ to denote possibly iterated sorts. The sets $\mathcal{T}_\Sigma^S(X)$, $\mathcal{T}_\Sigma^{S*}(X)$, $\mathcal{T}_\Sigma^{S+}(X)$ of terms of respectively sort S , starred sort S^* , and plussed sort S^+ are now defined. $\mathcal{T}_\Sigma^S(X)$ is the smallest set such that:

- x is an element of $\mathcal{T}_\Sigma^S(X)$ for each variable $x : \rightarrow S$.
- c is an element of $\mathcal{T}_\Sigma^S(X)$ for each (constant) function $c : \rightarrow S$.
- For each function $f : T1 \# T2 \# \dots \# Tn \rightarrow S$ with $n \geq 1$ and for all terms $t1 \in \mathcal{T}_\Sigma^{T1}(X)$, $t2 \in \mathcal{T}_\Sigma^{T2}(X)$, \dots , $tn \in \mathcal{T}_\Sigma^{Tn}(X)$ we define $f(t1, t2, \dots, tn)$ to be an element of $\mathcal{T}_\Sigma^S(X)$.

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

- x is an element of $\mathcal{T}_\Sigma^{S*}(X)$ for each variable $x : \rightarrow S^*$ or $x : \rightarrow S^+$.
- The list $[t1, t2, \dots, tn]$ where $n \geq 0$ is an element of $\mathcal{T}_\Sigma^{S*}(X)$ if for all $1 \leq i \leq n$ either $ti \in \mathcal{T}_\Sigma^S(X)$ holds or ti is a variable of type S^* or S^+ .

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

- x is an element of $\mathcal{T}_\Sigma^{S+}(X)$ for each variable $x : \rightarrow S^+$.
- $[t1, t2, \dots, tn]$ with $n \geq 1$ is an element of $\mathcal{T}_\Sigma^{S+}(X)$ if for all $1 \leq i \leq n$ either $ti \in \mathcal{T}_\Sigma^S(X)$ holds or ti is a variable of type S^* or S^+ . At least one of the ti should not be a variable of type S^* .

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

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

- identical sorts s appear at both positions, or
- a type s^+ in one position corresponds to s^* or s^+ at the other position, or
- a type s^* corresponds to s^+ or s^* or another starred sort s_1^* .

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

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

- For each (constant) function $c : \rightarrow s$ we define $\rho(c) \equiv c$.
- For all functions $f : t_1 \# t_2 \# \dots \# t_n \rightarrow s$ with $n \geq 1$ ρ is defined by

$$\rho(f(t_1, t_2, \dots, t_n)) \equiv f(\rho(t_1), \rho(t_2), \dots, \rho(t_n)).$$

- For the empty list we define: $\rho([]) \equiv []$.
- Finally, for non-empty lists $[t_1, t_2, \dots, t_n]$ with $n \geq 1$ which are an element of $\mathcal{T}_{\Sigma}^{s*}(\mathcal{X})$ or $\mathcal{T}_{\Sigma}^{s+}(\mathcal{X})$ suppose

$$\rho([t_2, \dots, t_n]) = [s_1, s_2, \dots, s_m]$$

with $m \geq 0$. There are two possibilities:

- If $\rho(t_1)$ is an element of $\mathcal{T}_{\Sigma}^s(X)$, or a variable of type S^* or S^+ then

$$\rho([t_1, t_2, \dots, t_n]) \equiv [\rho(t_1), s_1, s_2, \dots, s_m].$$

- If $\rho(t_1)$ is an element of $\mathcal{T}_{\Sigma}^{s*}(X)$ or $\mathcal{T}_{\Sigma}^{s+}(X)$ of the form $[u_1, u_2, \dots, u_k]$ with $k \geq 0$ we define

$$\rho([t_1, t_2, \dots, t_n]) \equiv [u_1, u_2, \dots, u_k, s_1, s_2, \dots, s_m].$$

3.4.3 Semantics

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

```

module Lists
begin

parameters
Sort begin
  sorts S
end Sort

exports
begin
  sorts S-plus, S-star
  functions
    inj      : S                -> S-plus
    c-pp     : S-plus # S-plus -> S-plus {assoc}
    empty-S  :                  -> S-star
    inj      : S-plus           -> S-star
    c-ps     : S-plus # S-star  -> S-plus
    c-sp     : S-star # S-plus  -> S-plus
    c-ss     : S-star # S-star  -> S-star {assoc}
  end
end

variables
  sp, sp1, sp2 : -> S-plus
  ss           : -> S-star

equations
[1]  c-ps(sp, empty-S)  = sp

```

```

[2]  c-ps(sp1, inj(sp2)) = c-pp(sp1, sp2)
[3]  c-sp(empty-S, sp)   = sp
[4]  c-sp(inj(sp1), sp2) = c-pp(sp1, sp2)
[5]  c-ss(ss, empty-S)   = ss
[6]  c-ss(empty-S, ss)   = ss
[7]  c-ss(inj(sp1), inj(sp2)) = inj(c-pp(sp1, sp2))

end Lists

```

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

- $\tau_s(x) \equiv x$ for each variable $x : \rightarrow S$.
- $\tau_s(c) \equiv c$ for each (constant) function $c : \rightarrow S$.
- $\tau_s(f(t_1, t_2, \dots, t_n)) \equiv f(\tau_{T_1}(t_1), \tau_{T_2}(t_2), \dots, \tau_{T_n}(t_n))$ for each function $f : T_1 \# T_2 \# \dots \# T_n \rightarrow S$ with $n \geq 1$.

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

- $\tau_{s^*}(x) \equiv x$ for each variable $x : \rightarrow S^*$.
- $\tau_{s^*}(x) \equiv \text{inj}(x)$ for each variable $x : \rightarrow S^+$.
- $\tau_{s^*}([\]) \equiv \text{empty-S}$.
- $\tau_{s^*}([t_1, t_2, \dots, t_n]) \equiv \text{inj}(\tau_{s^+}([t_1, t_2, \dots, t_n]))$ if at least one of the t_i ($1 \leq i \leq n$) is not a variable of type S^* .
- $\tau_{s^*}([x_1, x_2, \dots, x_n]) \equiv \text{c-ss}(x_1, \tau_{s^*}([x_2, \dots, x_n]))$ if all x_i ($1 \leq i \leq n$) are variables of type S^* .

For all terms $t \in \mathcal{T}_{\Sigma}^{S^+}(X)$ the translation τ_{s^+} is defined such that $\tau_{s^+}(t)$ is an element of $\mathcal{T}_{\Sigma'}^{S\text{-plus}}(X)$:

- $\tau_{s^+}(x) \equiv x$ for each variable $x : \rightarrow S^+$.
- $\tau_{s^+}([t_1, t_2, \dots, t_n]) \equiv \text{c-ps}(\text{inj}(\tau_{s^*}(t_1)), \tau_{s^*}([t_2, \dots, t_n]))$ if $t_1 \in \mathcal{T}_{\Sigma}^S(X)$ and $n \geq 1$.
- $\tau_{s^+}([x, t_2, \dots, t_n]) \equiv \text{c-ps}(x, \tau_{s^*}([t_2, \dots, t_n]))$ if $x : \rightarrow S^+$ and $n \geq 1$.
- $\tau_{s^+}([x, t_2, \dots, t_n]) \equiv \text{c-sp}(x, \tau_{s^+}([t_2, \dots, t_n]))$ if $x : \rightarrow S^*$ and $n \geq 2$.

3.4.4 Implementation in Prolog

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

$$[2] \quad \text{succ}(\text{nat}([m, 0])) = \text{nat}([m, 1])$$

of the example of Section 3.4.1 would give:

$$[2] \quad \text{succ}(\text{nat}(\text{c-sp}(m, \text{c-ps}(\text{inj}(0), \text{empty-DIGIT})))) \\ = \text{nat}(\text{c-sp}(m, \text{c-ps}(\text{inj}(1), \text{empty-DIGIT}))).$$

The translation of the term $\text{succ}(\text{nat}([0]))$ which is

$$\text{succ}(\text{nat}(\text{c-ps}(\text{inj}(0), \text{empty-DIGIT})))$$

cannot match the left-hand side of the translated equation. The same holds for the translation of $\text{succ}(\text{nat}([1, 0]))$ which is

$$\text{succ}(\text{nat}(\text{c-ps}(\text{inj}(1), \text{inj}(\text{c-ps}(\text{inj}(0), \text{empty-DIGIT}))))).$$

Even reducing both sides of the translated equation [2] using the equations of module *Lists* as given in Section 3.4.3 gives no solution:

$$[2] \quad \text{succ}(\text{nat}(\text{c-sp}(m, \text{inj}(0)))) = \text{nat}(\text{c-sp}(m, \text{inj}(1))).$$

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

$$[2a] \quad \text{succ}(\text{nat}([0])) = \text{nat}([1]) \\ [2b] \quad \text{succ}(\text{nat}([m, 0])) = \text{nat}([m, 1])$$

where $m : \rightarrow \text{DIGIT}^+$.

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

1. In the construction of legal terms given in Section 3.4.2 lists as arguments of a list are forbidden. As a consequence, care must be taken that no lists as arguments of lists occur during list construction. Hence, in the decomposition of the right-hand side *append* predicates to join lists must be generated.
2. To match a given list with the left-hand side of an equation it must be possible to split the given list in arbitrary parts. The *append* predicate is also used for this.

As an example the code generated for the example of Section 3.4.1 is presented:

[illegible]

```

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

```

In general, the code generation process is again an extension of the two steps described in Sections 3.2.2.1 and 3.3.4. So far, for each variable occurring in an equation a corresponding Prolog variable has been added in the typechecking phase. To prevent variables of a plussed sort from matching an empty list, an expression `[Head| Tail]` has been added to each such variable. Of course, the Prolog variables `Head` and `Tail` are different for each variable. So, instead of a list of variables with their corresponding Prolog variables, a list of corresponding Prolog expressions is generated.

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

- $t \equiv []$:
The list of predicates to be generated is empty and the translated term is the empty list `[]`.
- $t \equiv [t_1]$:
 - If t_1 is a variable x of an iterated sort then the translated term of t_1 is the expression which is associated to it in the typechecking phase. Hence, if x is of a starred sort it is the Prolog variable associated to x , and if x is of a plussed sort it is an expression of the form `[Head| Tail]`. In this case the generated list of predicates is empty.
 - If t_1 is not a variable of an iterated sort and the translated term for t_1 is T_1 and the list of predicates is L_1 then the list of predicates for `[t_1]` is L_1 and the translated term for `[t_1]` is `[T_1]`.
- $t \equiv [t_1, t_2, \dots, t_n]$ with $n \geq 1$:
Let the translated term of `[t_2, \dots, t_n]` be Tr , and let the list of predicates be L_r .
 - If t_1 is a variable x of an iterated sort and the Prolog expression which corresponds to x is T_x then the list of predicates for t is L_r with


```
append(Tx, Tr, Var)
```

 added at the end. Here `var` is a fresh Prolog variable which is also the translated term of t .
 - If t_1 is not a variable of an iterated sort and the translated term for t_1 is T_1 and the list of predicates is L_1 then the list of predicates for t is a concatenation of L_r and L_1 . The translated term for t is `[T_1 | Tr]`.

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

The handling of conditional equations is similar to what is done in Section 3.2.2.2. The only difference in the treatment of input is that terms in which variables of iterated sorts occur cannot be handled. These terms are simply forbidden in the input.

3.5 Conclusions

As mentioned in the introduction, lists and associative functions do not add expressive power to an algebraic specification formalism, however, especially the use of lists gives more elegant specifications which are easier to read. Both features have been added to the ASF system (see Chapter 2 and [Hen88b]) using the given algorithms. Rewriting modulo associativity as well as rewriting modulo lists give a more powerful implementation for specifications using these features. The implementation of lists is reasonably fast as long as head-tail-like decomposition of lists in Prolog can be used. From the specification point of view other decompositions of lists are desirable and it is very useful to have an implementation for them.

A Case Study in ASF+SDF: Typechecking Mini-ML

This chapter presents an algebraic specification of a typechecker for Mini-ML, a sublanguage of ML.

4.1 Introduction

The specification of a typechecker for the functional language ML is a challenge because ML allows polymorphism and typechecking ML programs requires type inference. For an extensive overview of typing schemes see [CW85]. Typechecking ML (or parts of the language) has been the subject of several papers. [DM82] describes an inference system which yields type schemes for expressions and also gives an algorithm for computing the most general type of an expression. The specification is based on this algorithm. It specifies not only which expressions are typeable but also gives *false* (or equivalently: one or more error messages) if an expression is not typeable. Each method which is solely based on the set of inference rules mentioned above can only show which expressions are typeable and a proof at the meta-level is needed to show that an expression cannot be typed. [Car84] describes a system of type equations, a type inference system to typecheck ML expressions, and an implementation of a typechecker in ML. The Mini-ML specification in TYPOL [CDDK86, Kah87] resembles the type inference system presented in [DM82] and [Car84]. TYPOL [Kah87] is a specification formalism developed to describe the static and dynamic semantics of programming languages.

The specification as given in this chapter differs from the earlier one in [Hen89b] in the following respects:

- The formalism used is the current version of ASF+SDF as described in Section 1.4.

An earlier version of this specification was given in Chapter 7 of [BHK89a]. An extended abstract was published in the proceedings of the SION conference CSN'87 [Hen87].

- *Inequalities* are used in conditions as described in [HK89a]. As a consequence, some uninteresting parts of the specification like the specification of equality for identifiers and natural numbers could be removed.
- The current specification can be executed in the system as described in Chapter 5 because the underlying term rewriting system is confluent and terminating.
- Apart from specifying which expressions are typeable and which are not, I also specify which error messages should be generated for incorrect expressions.

The typechecker is informally described in Section 4.2 and the specification itself is presented in Section 4.3. Section 4.4 describes how an implementation for a typechecker can be derived from the algebraic specification. Finally, Section 4.5 contains some remarks on questions related to the specification.

4.2 Mini-ML

Mini-ML is a small sublanguage of the Standard ML Core Language [HMM86, HMT87]. The version of Mini-ML used here is a slight modification of the language used in [CDDK86]. As far as typechecking is concerned, Mini-ML contains all essential elements of ML. Information on the static semantics of ML (and Mini-ML) is given in [CDDK86, Car84, DM82]. [DM82] describes a typechecking algorithm for a sublanguage of ML which is even smaller than Mini-ML. It determines the most general type for every expression of the language. This algorithm is essentially the one used in the algebraic specification presented in Section 4.3.

The syntax of Mini-ML and the Mini-ML typechecker are described in the following sections.

4.2.1 Concrete syntax of Mini-ML

Mini-ML expressions have the following syntax:

| | |
|-----------------------------------|-----------------------|
| <code><exp> ::= true</code> | Boolean constants |
| false | |
| <natural-number> | |
| <ident> | |
| (<exp> <exp>) | application |
| λ <ident> . <exp> | lambda-abstraction |
| let <ident> = <exp> in <exp> | declaration |
| letrec <ident> = <exp> in <exp> | recursive declaration |
| if <exp> then <exp> else <exp> fi | conditional |
| (<exp> , <exp>) | Cartesian product |

In this definition $\langle \text{idnt} \rangle$ and $\langle \text{natural-number} \rangle$ are predefined lexical notions for, respectively, identifiers and natural numbers.

4.2.2 Typechecking Mini-ML

First, the syntax and semantics of types and of generalized types are described, and some examples of typechecking will be given. Next, an informal description of the typecheck algorithm is presented in Section 4.2.2.2.

4.2.2.1 Syntax and semantics of types and generalized types

Each closed expression (i.e., expressions without free variables: all identifiers are bound by lambda-abstraction, a declaration or a recursive declaration) of Mini-ML denotes a basic notion (like Booleans or natural numbers), a function, or a Cartesian product. Hence, it is possible to attach a type to each expression defined by the following syntax:

| | |
|---|-------------------------------------|
| $\langle \text{type} \rangle ::= \langle \text{var} \rangle$ | |
| <i>bool</i> | type of the Booleans |
| <i>nat</i> | type of the natural numbers |
| $\langle \text{type} \rangle \rightarrow \langle \text{type} \rangle$ | function-type; right-associative |
| $\langle \text{type} \rangle \times \langle \text{type} \rangle$ | Cartesian product; left-associative |

Here, $\langle \text{var} \rangle$ is a non-terminal which produces type variables. In the sequel, the symbols $\sigma_0, \sigma_1, \sigma_2, \dots$ are used as type variables. $\sigma_0 \rightarrow \sigma_1$ is the type of an expression which is a function from expressions of type σ_0 to expressions of type σ_1 . $\sigma_0 \times \sigma_1$ is the Cartesian product of the types σ_0 and σ_1 . The Cartesian product \times binds more strongly than \rightarrow .

An expression can have several possible types. For instance, $\text{bool} \rightarrow \text{bool}$, $(\sigma_2 \times \sigma_2) \rightarrow (\sigma_2 \times \sigma_2)$ or $\sigma_1 \rightarrow \sigma_1$ are some of the possible types of the identity function " $\lambda x. x$ ". The typecheck algorithm always returns the *most general type* of an expression, i.e., the type from which all other possible types can be derived using a substitution which replaces type variables by types. The most general type of an expression is unique up to renaming of the type variables occurring in it. The most general type of " $\lambda x. x$ " is $\sigma_1 \rightarrow \sigma_1$.

An identifier defined in a (recursive) declaration is assumed to be polymorphic. If the identifier occurs more than once in the expression part (the in-part of the let- or letrec-construction) each occurrence may have another type. Consider the following examples:

$$\text{let } x = \lambda y. y \text{ in } (x \ x) \tag{1}$$

$$(\lambda x. (x x) \lambda y. y) \quad (2)$$

In the first expression x is declared to be the polymorphic identity function. Both occurrences of x in (1) should have a type of the form $\sigma_1 \rightarrow \sigma_1$. This is possible if we assign the type $(\sigma_2 \rightarrow \sigma_2) \rightarrow (\sigma_2 \rightarrow \sigma_2)$ to the first occurrence of x and $\sigma_2 \rightarrow \sigma_2$ to the second one. In expression (2) both occurrences of the identifier x should always have the same type. Hence, it is impossible to assign a correct type to this expression because the hierarchical type structure of Mini-ML forbids the application of x to itself. The fact that x will be bound to " $\lambda y. y$ " is irrelevant here.

It is not possible to describe polymorphic declarations by attaching types to identifiers. The notion of types has to be generalized in order to distinguish generic type variables (i.e., different occurrences may have different type values) and "normal" type variables (i.e., all occurrences have the same type value). Generalized types have the following syntax:

$$\begin{aligned} \text{<gen-type>} ::= & \text{<type>} \\ & | \text{<gen-var>} \\ & | \text{<gen-type>} \rightarrow \text{<gen-type>} \\ & | \text{<gen-type>} \times \text{<gen-type>} \end{aligned}$$

Here, <gen-var> represents *generic* type variables which will be written as $\beta_0, \beta_1, \beta_2, \dots$. Rules for priority and associativity of \rightarrow and \times are similar to the ones given earlier for types. The syntax of generalized types (or type schemes) is somewhat different from the one used in [DM82]. In [DM82] polymorphism is described by type schemes: types prefixed with universal quantifiers in order to bind some of the type variables in the type. This syntax could be handled in the specification, but for reasons of readability generic type variables are simply represented by $\beta_0, \beta_1, \beta_2, \dots$.

The typecheck algorithm associates a generalized type with each identifier in a Mini-ML expression. This information is kept in a type environment. If an *instance* of a generalized type is needed, all generic type variables are changed to "fresh" type variables (i.e., type variables which have not yet been used by the typecheck algorithm). The type computed for an identifier defined by a (recursive) declaration is *generalized*, i.e., each type variable that does not occur in the type environment is potentially polymorphic and is changed into a generic type variable.

A syntactically correct Mini-ML expression is only typeable if it satisfies the following constraints:

- The expression is closed. Otherwise it would contain identifiers not bound by a lambda-abstraction, a declaration or a recursive declaration. An example is: " $\lambda x. y$ ".

- The type structure of Mini-ML expressions is hierarchical, which means that an expression may not be applied to itself. Expressions like " $\lambda x. (x\ x)$ " and " $\lambda x. \lambda y. ((x\ y)\ x)$ " are forbidden.
- In a recursive declaration the identifier and all its occurrences in the declaration part must be typeable with the same type. An expression like " $\text{letrec } x = (x\ 4)\ \text{in } \dots$ " is not typeable.
- In an if-then-else-fi expression, the first expression should have type *bool* and the then- and else-part of the expression should have the same types. Examples of erroneous expressions are: " $\lambda x. \text{if } 4 \text{ then } x \text{ else } x\ \text{fi}$ ", and " $\lambda x. \text{if } x \text{ then } 2 \text{ else true fi}$ ".
- All subexpressions of an expression should be typeable using the information from the type environment in case subexpressions are not closed.

4.2.2.2 The typechecker

In this section the typecheck algorithm is described informally. Typechecking is defined recursively on the (abstract) syntactic structure of an expression E . The typechecker computes the most general type of E , provided this type exists. During typechecking, a type environment T is constructed, which contains the generalized types of the identifiers occurring in E . In this informal description the following details will not be considered:

- changes in the type environment as a consequence of typechecking of subexpressions;
- changes in types as a result of unification.

These details can be found in the algebraic specification of the algorithm. The typecheck algorithm distinguishes the following cases:

- $E \equiv \text{true or } E \equiv \text{false}$:
"true" and "false" are both of constant type *bool*.
- E is a natural number:
All natural numbers are of constant type *nat*.
- E is an identifier:
Each identifier in an expression must have been bound by lambda-abstraction, a declaration or a recursive declaration. We only need to examine the type environment T . If the identifier is present in T , the type of E is an instance of the generalized type found in T . Otherwise, E cannot be typed.
- $E \equiv (E_1\ E_2)$:
The main idea is to determine the types of both expressions E_1 and E_2 , which gives τ_1 and τ_2 respectively. These types must be such that τ_1 can "eat" τ_2 , i.e.,

we must unify τ_1 and $\tau_2 \rightarrow \sigma_1$, where σ_1 is a “fresh” variable. The type of the whole expression is σ_1 .

- $E \equiv \lambda x. E_1$:
First, identifier x is added to type environment T with type σ_1 , where σ_1 is a “fresh” type variable. Then the type τ_1 of E_1 is determined using the new type environment T_1 . The type of E is $\sigma_1 \rightarrow \tau_1$. Identifier x and its corresponding generalized type are removed from the type environment T_1 .
- $E \equiv \text{let } x = E_1 \text{ in } E_2$:
First, E_1 is typechecked in type environment T , and then the type of E_1 is generalized, resulting in a generalized type α_1 . Now the identifier x and its associated generalized type α_1 are added to the environment and the type of E_2 is determined using this type environment. The type of the entire expression is the type of E_2 . Finally, x and its generalized type are removed from the environment.
- $E \equiv \text{letrec } x = E_1 \text{ in } E_2$:
The identifier x is added to the type environment T with a “fresh” type variable σ_1 and then the type of E_1 is determined. The types of E_1 and σ_1 have to be equal, i.e., these types have to be unified (note that σ_1 could have been changed due to unification while typechecking E_1). Now, x and its associated generalized type are removed from the type environment and the type resulting from the unification is generalized, resulting in the generalized type α_1 . Next, x with the generalized type α_1 are added to the type environment and E_2 is typechecked. The type of E is the type of E_2 and, once again, x and its generalized type are removed from the type environment.
- $E \equiv \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \text{ fi}$:
Let τ_1 , τ_2 and τ_3 be the types of E_1 , E_2 and E_3 respectively. τ_1 has to be unified with *bool* and τ_2 with τ_3 resulting in the type τ_4 . The type of the expression E is τ_4 .
- $E \equiv (E_1, E_2)$:
The type of E is the Cartesian product $\tau_1 \times \tau_2$ of the types τ_1 and τ_2 of E_1 and E_2 respectively.

The typecheck algorithm is illustrated by typechecking the expression “ $\lambda x. \lambda y. (x (x y))$ ”. We start with an empty type environment T_0 , and perform the following steps:

- Add (x, σ_1) , the identifier “ x ” and a “fresh” type variable σ_1 to the environment resulting in $T_1 = [(x, \sigma_1)]$.
- Typecheck the subexpression “ $\lambda y. (x (x y))$ ” in the environment T_1 :

- Extend the environment T_1 to $T_2 = [(y, \sigma_2), (x, \sigma_1)]$ and typecheck " $(x (x y))$ ":
 - First, the type of the first part of the application has to be determined. Examining the environment T_2 shows that the type of " x " is σ_1 .
 - Next, the second part of the application, i.e., " $(x y)$ " is typechecked.
 - " x " has type σ_1 .
 - " y " has type σ_2 .
 - In order to unify σ_1 and $\sigma_2 \rightarrow \sigma_3$, (where σ_3 is a "fresh" type variable) σ_1 is simply changed into $\sigma_2 \rightarrow \sigma_3$. As a consequence, the environment T_2 changes into $T_3 = [(y, \sigma_2), (x, \sigma_2 \rightarrow \sigma_3)]$ and the type of " $(x y)$ " is σ_3 .
 - The expression " $(x (x y))$ " can now be typechecked. The type of the first part is $\sigma_2 \rightarrow \sigma_3$ and the type of the second part is σ_3 . Hence, we have to unify $\sigma_2 \rightarrow \sigma_3$ and $\sigma_3 \rightarrow \sigma_4$. The solution is to change σ_2 and σ_3 into σ_4 . The type of " $(x (x y))$ " therefore is σ_4 and the environment is changed into $[(y, \sigma_4), (x, \sigma_4 \rightarrow \sigma_4)]$.
- The type of " $\lambda y. (x (x y))$ " is $\sigma_4 \rightarrow \sigma_4$ and we can delete the information about " y " from the environment.
- The type of the expression is $(\sigma_4 \rightarrow \sigma_4) \rightarrow (\sigma_4 \rightarrow \sigma_4)$ and the environment becomes empty after the deletion of the identifier " x " and its generalized type.

4.3 Algebraic specification of Mini-ML typechecking

4.3.1 Basic notions

This section contains the algebraic specification of four basic notions (layout, Booleans, natural numbers, and tables), necessary to specify the typechecker.

4.3.1.1 Layout

Module `Layout` is identical to the module in Section 1.4. It is also (directly or indirectly) imported in all other modules of this specification.

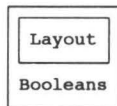
```
module Layout
  exports
    lexical syntax
    [ \t\n ] -> LAYOUT
```

**Fig. 4.1.** Structure diagram of Layout

4.3.1.2 Booleans

These are specified in module `Booleans`, in which sort `BOOL` with constants (functions without arguments) `true` and `false`, and Boolean operators `|` (or), `&` (and) and `~` (negation) are defined.

Note that sort `BOOL` cannot be used to define the Mini-ML-expressions `"true"` and `"false"` in Section 4.3.2. If sort `BOOL` would have been used for that purpose we would automatically define extra expressions in Mini-ML such as, e.g., `"~ true"`.

**Fig. 4.2.** Structure diagram of Booleans

```

module Booleans

imports Layout

exports
  sorts BOOL
  context-free syntax
    true          -> BOOL
    false         -> BOOL
    BOOL "&" BOOL -> BOOL assoc
    BOOL "&" BOOL -> BOOL assoc
    "~" BOOL      -> BOOL
    "(" BOOL ")"  -> BOOL bracket
  variables
    bool [0-9]* -> BOOL

priorities
  "&" < "&" < "~"

equations
[1] true | bool = true
[2] false | bool = bool
[3] true & bool = bool
[4] false & bool = false
[5] ~ true = false
[6] ~ false = true
  
```

4.3.1.3 Natural-Numbers

Natural numbers will be used in several ways throughout the specification:

- Natural numbers are expressions in Mini-ML and as such module `Natural-Numbers` is imported in module `Mini-ML-Expressions`.
- The variables and generic variables in types and generalized types are renamings of the natural numbers.

Natural numbers are written in ordinary decimal notation and all operations on numbers are defined in this decimal representation. Compare this with the “classical” algebraic definition in Section 1.2 which uses a unary representation: the constant 0 and the successor function generate all numbers. The specification of natural numbers as given in Section 1.4 cannot be reused as natural numbers with leading zeros and expressions containing the successor function are forbidden in the syntax of Mini-ML. For this reason, a sort `LEX-NAT` is defined which contains the lexical definition of the natural numbers as a sequence of one or more digits beginning with a non-zero digit.

In module `Natural-Numbers` sorts `LEX-NAT` and `NAT` are defined together with some functions:

- `[0-9] -> LEX-NAT` and `[1-9] [0-9]+ -> LEX-NAT` give the lexical definition of the natural numbers.
- `succ "(" NAT ")" -> NAT` defines the successor on natural numbers.

In equations [7] through [15], and [17] the function

```
lex-nat "(" CHAR* ")" -> LEX-NAT
```

is used to split the string of characters in the lexical item representing an element of sort `LEX-NAT`. These functions are generated automatically for each output sort of a function in a `lexical syntax` section.

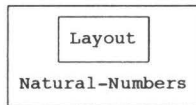


Fig. 4.3. Structure diagram of `Natural-Numbers`

```

module Natural-Numbers
  imports Layout
  exports
    sorts LEX-NAT NAT
    lexical syntax
  
```

```

[0-9]          -> LEX-NAT
[1-9] [0-9]+ -> LEX-NAT
context-free syntax
LEX-NAT        -> NAT
succ "(" NAT ")" -> NAT

hiddens
variables
m          -> CHAR*
k [0-9]* -> CHAR+

equations
[7] succ(lex-nat(m "0")) = lex-nat(m "1")
[8] succ(lex-nat(m "1")) = lex-nat(m "2")
[9] succ(lex-nat(m "2")) = lex-nat(m "3")
[10] succ(lex-nat(m "3")) = lex-nat(m "4")
[11] succ(lex-nat(m "4")) = lex-nat(m "5")
[12] succ(lex-nat(m "5")) = lex-nat(m "6")
[13] succ(lex-nat(m "6")) = lex-nat(m "7")
[14] succ(lex-nat(m "7")) = lex-nat(m "8")
[15] succ(lex-nat(m "8")) = lex-nat(m "9")
[16] succ(9) = 10
[17] succ(lex-nat(k1 "9")) = lex-nat(k2 "0")
      when succ(lex-nat(k1)) = lex-nat(k2)

```

4.3.1.4 Tables

Tables is a parameterized module defining lists of pairs consisting of a key and a corresponding entry. This module has *Keys* and *Entries* as parameters and is used to define type environments and substitutions on types (see Sections 4.3.3.2, 4.3.3.3, and 4.3.3.4).

The *parameters* section of module *Tables* describes the requirements on the export signatures of modules that are to be bound to parameters *Keys* and *Entries*. Parameter *Keys* only defines a sort *KEY*, and parameter *Entries* defines a sort *ENTRY* and a constant *error-entry*.

Some remarks:

- This module exports sorts *PAIR*, *TABLE*, and *LOOKUP-OUT*.
- `"(" KEY "," ENTRY ")" -> PAIR`, `"[" {PAIR ","}* "]" -> TABLE`, and `"<" BOOL "," ENTRY ">" -> LOOKUP-OUT` are the constructor functions of the elements of sorts *PAIR*, *TABLE*, and *LOOKUP-OUT* respectively.
- `PAIR "+" TABLE -> TABLE` is a function which adds a pair to a table.
- `lookup KEY in TABLE -> LOOKUP-OUT` is the lookup function. It gives a Boolean which is true when the given key is in the table, and the first entry

associated with that key. It returns `<false, error-entry>` if the key is missing.

The definitions are such that multiple occurrences of a key are not removed. This is essential when tables are used as type environments and nested declarations of identifiers have to be handled. The treatment of multiple occurrences is irrelevant in the other instances of `Tables`.

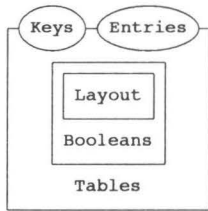


Fig. 4.4. Structure diagram of `Tables`

```

module Tables
  imports Booleans

  parameter Keys
    sorts KEY

  parameter Entries
    sorts ENTRY
    context-free syntax
    error-entry -> ENTRY

  exports
    sorts PAIR TABLE LOOKUP-OUT
    context-free syntax
    "(" KEY "," ENTRY ")" -> PAIR
    "[" {PAIR ","}* "]" -> TABLE
    PAIR "+" TABLE -> TABLE
    "(" TABLE ")" -> TABLE bracket
    "<" BOOL "," ENTRY ">" -> LOOKUP-OUT
    lookup KEY in TABLE -> LOOKUP-OUT

  hiddens
    variables
      key [0-9]* -> KEY
      entry -> ENTRY
      pair -> PAIR
      pairs -> {PAIR ","}*
  
```

```

equations
[18] pair + [pairs] = [pair, pairs]
[19] lookup key in [] = <false, error-entry>
[20] lookup key in [(key, entry), pairs]
    = <true, entry>
[21]     key1 != key2
    =====
    lookup key1 in [(key2, entry), pairs]
    = lookup key1 in [pairs]

```

4.3.2 The syntax of Mini-ML

The module `Mini-ML-Expressions` defines the syntax of expressions in Mini-ML. Identifiers are non-empty sequences of digits or letters preceded by a letter. A different syntax has to be chosen for the variables of sort `ID`. For this reason, the variable `Id` beginning with a capital `I` is defined in the `variables` section of this module.

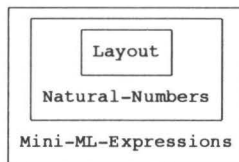


Fig. 4.5. Structure diagram of `Mini-ML-Expressions`

```

module Mini-ML-Expressions
imports Natural-Numbers

exports
  sorts ID EXP
  lexical syntax
    [a-z] [a-z0-9]* -> ID
  context-free syntax
    true                -> EXP
    false               -> EXP
    LEX-NAT             -> EXP
    ID                  -> EXP
    "(" EXP EXP ")"     -> EXP
    lambda ID "." EXP   -> EXP
    let ID "=" EXP in EXP -> EXP
    letrec ID "=" EXP in EXP -> EXP
    if EXP then EXP else EXP fi -> EXP
    "(" EXP "," EXP ")" -> EXP

```

```

variables
  Id      -> ID
  Exp [0-9]* -> EXP

```

4.3.3 Types and tools to handle types

This section contains the specification of the syntax of types and generalized types (4.3.3.1). In addition to this, several operations on types are defined. Section 4.3.3.2 describes type substitutions. These have two applications. The result of unification is a type substitution and during typechecking changes in the type environment are represented by a type substitution. Type environments are defined in Section 4.3.3.3 and the functions to generalize a type and to instantiate a generalized type can be found in module `Type-Instant-Generalize` (in 4.3.3.4). Finally, the unification algorithm is specified in Section 4.3.3.5.

4.3.3.1 Types

Module `Types` defines the syntax of types (sort `TYPE`) and generalized types (sort `GEN-TYPE`). The type variables $\sigma_0, \sigma_1, \sigma_2, \dots$ are represented by `s0, s1, s2, ...` and generic type variables $\beta_0, \beta_1, \beta_2, \dots$ by `b0, b1, b2, ...`. The symbols \rightarrow and \times are respectively replaced by `->` and `#`.

In the `imports` section the natural numbers are used to define type variables (sort `VAR`) and generic type variables (sort `GEN-VAR`). The left-hand side of `=>` gives the name of the sort (in the `sorts` section) or the syntax of the function (in the `lexical syntax` and `context-free syntax` section) which is renamed to the right-hand side. In the left-hand side of a renaming of a context-free syntax function abbreviation is allowed, the terminal skeleton must be given.

The `priorities` section gives the priorities as they were informally defined in Section 4.2.2.1. It is equivalent to the four inequalities:

$$\begin{aligned} \text{TYPE } \text{"->" } \text{TYPE } \text{"->" } \text{TYPE} &< \\ \text{TYPE } \text{"\#"} \text{TYPE } \text{"->" } \text{TYPE} &< \end{aligned} \quad (1)$$

$$\begin{aligned} \text{GEN-TYPE } \text{"->" } \text{GEN-TYPE } \text{"->" } \text{GEN-TYPE} &< \\ \text{GEN-TYPE } \text{"\#"} \text{GEN-TYPE } \text{"->" } \text{GEN-TYPE} &< \end{aligned} \quad (2)$$

$$\begin{aligned} \text{GEN-TYPE } \text{"->" } \text{GEN-TYPE } \text{"->" } \text{GEN-TYPE} &< \\ \text{TYPE } \text{"->" } \text{TYPE } \text{"->" } \text{TYPE} &< \end{aligned} \quad (3)$$

$$\begin{aligned} \text{GEN-TYPE } \text{"\#"} \text{GEN-TYPE } \text{"->" } \text{GEN-TYPE} &< \\ \text{TYPE } \text{"\#"} \text{TYPE } \text{"->" } \text{TYPE} &< \end{aligned} \quad (4)$$

Inequalities (1) and (2) express the fact that $\#$ binds more strongly than \rightarrow for elements of sort `TYPE` and `GEN-TYPE` respectively. Inequalities (3) and (4) state that expressions constructed with $\#$ and \rightarrow are interpreted as elements of `TYPE` whenever possible.

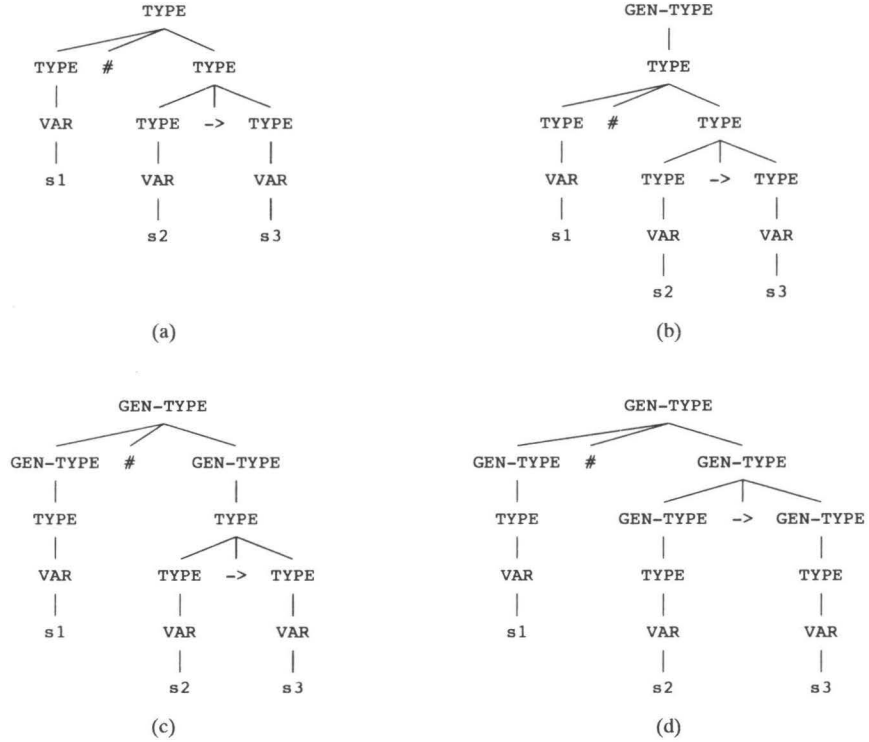


Fig. 4.6. Parse trees of `s1 # (s2 -> s3)`

As an example Figure 4.6 shows the four possible parse trees of the expression `s1 # (s2 -> s3)` if no priorities would have been defined. Parse tree (b) is selected if the expression is used in a context which forces it to be an element of `GEN-TYPE`. In other cases parse tree (a) is chosen.

In the three actualizations of module `Tables` as described in the following sections the parameter `Entries` is always bound to `Types`. In these parameter bindings the constant `error-entry` has to be bound to a constant in the actual sort to which `ENTRY` is bound. To serve this purpose the constants `error-type`, `error-gt`, and `error-var` are specified in module `Types`. Without equations [22], [23], and [24] these extra constants would have added extra elements like `error-type -> s3` and `bool # error-type` to `TYPE` and `GEN-TYPE`. The function `elm` tests whether a type variable occurs in a type or generalized type.

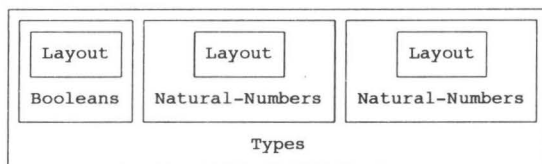


Fig. 4.7. Structure diagram of Types

```

module Types

imports Booleans
       Natural-Numbers
       renamed by
       sorts
           LEX-NAT => LEX-VAR
           NAT      => VAR
       lexical syntax
           [0-9] -> LEX-NAT => s [0-9] -> LEX-VAR
           [1-9] [0-9]+ -> LEX-NAT
               => s [1-9] [0-9]+ -> LEX-VAR
       context-free syntax
           succ "(" ")" => next "(" VAR ")" -> VAR

Natural-Numbers
       renamed by
       sorts
           LEX-NAT => LEX-GEN-VAR
           NAT      => GEN-VAR
       lexical syntax
           [0-9] -> LEX-NAT => b [0-9] -> LEX-GEN-VAR
           [1-9] [0-9]+ -> LEX-NAT
               => b [1-9] [0-9]+ -> LEX-GEN-VAR
       context-free syntax
           succ "(" ")" => next "(" GEN-VAR ")" -> GEN-VAR

exports
       sorts TYPE GEN-TYPE
       context-free syntax
           VAR -> TYPE
           bool -> TYPE
           nat -> TYPE
           TYPE "->" TYPE -> TYPE right
           TYPE "#" TYPE -> TYPE left
           "(" TYPE ")" -> TYPE bracket
           TYPE -> GEN-TYPE
           GEN-VAR -> GEN-TYPE
           GEN-TYPE "->" GEN-TYPE -> GEN-TYPE right

```

```

GEN-TYPE "#" GEN-TYPE -> GEN-TYPE left
"(" GEN-TYPE ")"      -> GEN-TYPE bracket
error-var              -> VAR
error-type             -> TYPE
error-gt               -> GEN-TYPE
VAR elm GEN-TYPE       -> BOOL

variables
var [0-9]* -> VAR
gv         -> GEN-VAR
type [0-9]* -> TYPE
gt [0-9]* -> GEN-TYPE

priorities
GEN-TYPE "->" GEN-TYPE -> GEN-TYPE <
{GEN-TYPE "#" GEN-TYPE -> GEN-TYPE,
 TYPE "->" TYPE -> TYPE} <
TYPE "#" TYPE -> TYPE

equations
[22] error-var = s0
[23] error-type = bool
[24] error-gt = bool
[25] var elm var = true
[26] var1 != var2 ==> var1 elm var2 = false
[27] var elm bool = false
[28] var elm nat = false
[29] var elm type1 -> type2 = var elm type1 | var elm type2
[30] var elm type1 # type2 = var elm type1 | var elm type2
[31] var elm gv = false
[32] var elm gt1 -> gt2 = var elm gt1 | var elm gt2
[33] var elm gt1 # gt2 = var elm gt1 | var elm gt2

```

4.3.3.2 Type-Substitutions

Type substitutions are used to express changes in types and type environments during typechecking. The result of the unification of a set of type equations is a type substitution, provided that the unification succeeds. Type substitutions are lists of pairs consisting of a type variable and its associated type. Module `Type-Substitutions` is defined as an instance of `Tables` in which parameter `Keys` is bound to `Types` (or more precisely, sort `KEY` is bound to sort `VAR`) and parameter `Entries` is also bound to `Types` (sort `ENTRY` is bound to `TYPE`).

After binding the parameters, sorts `PAIR`, `TABLE`, and `LOOKUP-OUT` are renamed as can be seen in the `imports` section of the specification of this module. Without this renaming, problems will occur when this module is combined with other

modules in which `Tables` is also imported without renaming these sorts. The origin rule of ASF (see [BHK89b]) will not forbid such imports as the origin of these sorts is always the module `Tables`. As a consequence, the sorts will be identified and it will be possible to construct tables in which different kinds of pairs exist. This is of course undesirable.

Functions `apply-type` and `apply-gt` define how a type substitution should be applied to an element of `TYPE` or `GEN-TYPE`, respectively. Applying a type substitution to a type containing a type variable that does not occur as a key in the substitution does not affect that variable. If a type variable occurs more than once as a key in a type substitution only the first occurrence is important. All other occurrences are ignored. Application of a type substitution is defined in such a way that the substitution is performed simultaneously on the whole type or generalized type. For instance, the result of

```
apply-type([(s1, s2 -> s3), (s2, bool), (s1, nat)], s1 -> s2)
```

is

```
(s2 -> s3) -> bool.
```

The composition of two type substitutions (the `o`-operator) is defined in such a way that all the above-mentioned properties of type substitutions also hold for the composition. The result of applying the composition of two type substitutions to a type or generalized type gives the same result as first applying the right-hand one and then applying the left-hand one to the result. For example: the expression

```
[(s2, s1 -> bool)] o
[(s1, s2 -> s3),
 (s2, bool),
 (s1, nat)]
```

is equal to

```
[(s1, (s1 -> bool) -> s3),
 (s2, bool),
 (s1, nat),
 (s2, s1 -> bool)].
```

Both substitutions give the same result when applied to, for instance, `s1` and `s2`.

The reason for the use of the suffixes `-type` and `-gt` in the definitions of `apply-type` and `apply-gt` is a subtle one. Both functions are needed in the rest of the specification: `apply-gt` is used in `Type-Environments`, and `apply-type` in module `Type-Equations`. In the latter use of the function `apply-type` it is important that the result type of it is `TYPE`, therefore, both functions are needed in the specification. Without the suffixes `-type` and `-gt` the grammar of this module

would be ambiguous and it would be impossible to express the equivalent of equation [40].

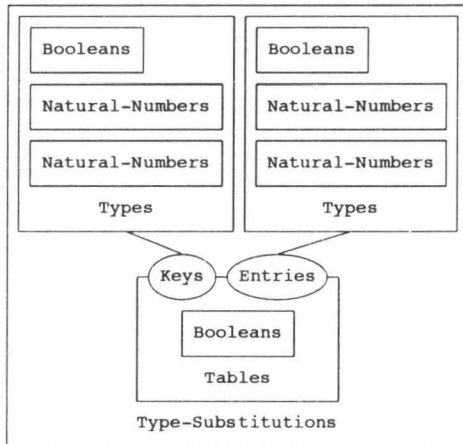


Fig. 4.8. Structure diagram of Type-Substitutions

```

module Type-Substitutions
imports
  Tables
  Keys bound by
    sorts KEY => VAR
  to Types
  Entries bound by
    sorts ENTRY => TYPE
    context-free syntax
    error-entry => error-type -> TYPE
  to Types
  renamed by
    sorts
      PAIR      => SUBS-PAIR
      TABLE    => TYPE-SUBS
      LOOKUP-OUT => TYPE-SUBS-LOOKUP-OUT

exports
  context-free syntax
    apply-type "(" TYPE-SUBS "," TYPE ")" -> TYPE
    apply-gt "(" TYPE-SUBS "," GEN-TYPE ")" -> GEN-TYPE
    TYPE-SUBS o TYPE-SUBS -> TYPE-SUBS
                                right
  variables
    subs [0-9]* -> TYPE-SUBS

```

```

u-subs [0-9]* -> TYPE-SUBS

hiddens
  variables
    pairs -> {SUBS-PAIR ", "}*

equations
[34] apply-type(subs, var) = var
      when lookup var in subs = <false, type>
[35] apply-type(subs, var) = type
      when lookup var in subs = <true, type>
[36] apply-type(subs, bool) = bool
[37] apply-type(subs, nat) = nat
[38] apply-type(subs, type1 -> type2)
    = apply-type(subs, type1) -> apply-type(subs, type2)
[39] apply-type(subs, type1 # type2)
    = apply-type(subs, type1) # apply-type(subs, type2)
[40] apply-gt(subs, type) = apply-type(subs, type)
[41] apply-gt(subs, gv) = gv
[42] apply-gt(subs, gt1 -> gt2)
    = apply-gt(subs, gt1) -> apply-gt(subs, gt2)
[43] apply-gt(subs, gt1 # gt2)
    = apply-gt(subs, gt1) # apply-gt(subs, gt2)

[44] subs o [] = subs
[45] subs o [(var, type), pairs]
    = (var, apply-type(subs, type)) + (subs o [pairs])

```

4.3.3.3 Type-Environments

Type-Environments is an instance of module Tables in which parameter Keys is bound to Mini-ML-Expressions and parameter Entries is bound to Types. It is important to allow multiple occurrences of an identifier as key in the type environment. This is needed for typechecking expressions in which an identifier occurs more than once in a nested fashion. Typechecking the expression " $\lambda x. (x (\lambda x. x \text{ true}))$ ", for instance, should give the same result as typechecking " $\lambda x. (x (\lambda y. y \text{ true}))$ ".

```

module Type-Environments

imports Booleans Type-Substitutions
  Tables
    Keys bound by
      sorts KEY => ID
    to Mini-ML-Expressions
    Entries bound by

```

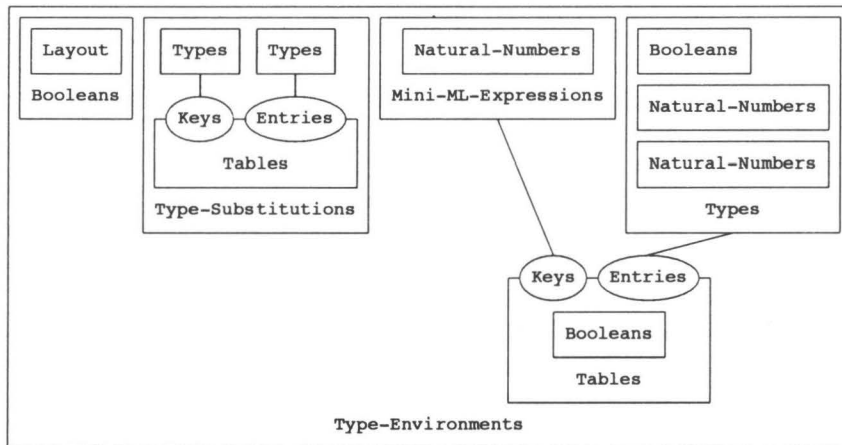


Fig. 4.9. Structure diagram of Type-Environments

```

sorts ENTRY => GEN-TYPE
context-free syntax
  error-entry => error-gt -> GEN-TYPE
to Types
renamed by
  sorts
    PAIR      => ENV-PAIR
    TABLE   => TYPE-ENV
    LOOKUP-OUT => TYPE-ENV-LOOKUP-OUT
exports
  context-free syntax .
    apply "(" TYPE-SUBS "," TYPE-ENV ")" -> TYPE-ENV
    VAR elm TYPE-ENV                      -> BOOL
  variables
    env [0-9]* -> TYPE-ENV
hiddens
  variables
    pairs -> {ENV-PAIR ","}*
equations
[46] apply(subs, []) = []
[47] apply(subs, [(Id, gt), pairs])
    = (Id, apply-gt(subs, gt)) + apply(subs, [pairs])
[48] var elm [] = false
[49] var elm [(Id, gt), pairs] = var elm gt | var elm [pairs]

```

4.3.3.4 Type-Instant-Generalize

Module `Type-Instant-Generalize` gives an algebraic specification of the functions to instantiate a generalized type and to generalize a type.

To *instantiate* a generalized type we simply change all generic type variables occurring in the type into “fresh” type variables which are created by means of the `next` function. The arguments of the function `instant` are a generalized type and a type variable. The latter should be the last type variable which has been used during typechecking. The output of the function is a tuple consisting of the result type and of the last type variable used by the instantiation process.

A type variable substitution is constructed during instantiation. When a generic type variable is encountered, it is looked up in the substitution. If it is not there, the generic type variable is changed into a “fresh” type variable and the pair of generic and “fresh” type variable is added to the substitution. If the generic type variable is already in the substitution, it is changed into the corresponding entry. This assures that all occurrences of the same generic type variable are instantiated to the same “fresh” type variable.

To *generalize* a type, we just have to change all type variables in the type that do not occur in the type environment into the corresponding generic type variable.

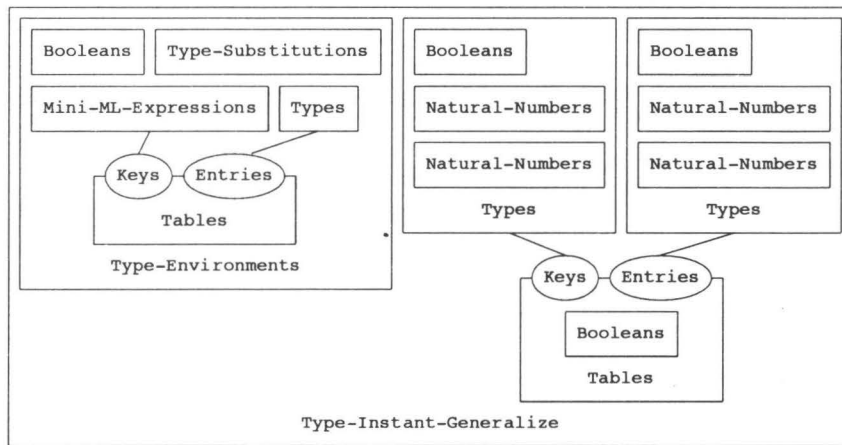


Fig. 4.10. Structure diagram of `Type-Instant-Generalize`

```

module Type-Instant-Generalize
  imports Type-Environments
    Tables
      Keys bound by
        sorts KEY => GEN-VAR
      to Types

```

```

    Entries bound by
      sorts ENTRY => VAR
      context-free syntax
        error-entry => error-var -> VAR
    to Types
    renamed by
      sorts
        PAIR          => VAR-PAIR
        TABLE        => VAR-SUBS
        LOOKUP-OUT    => VAR-SUBS-LOOKUP-OUT

exports
  sorts INSTANT-OUT
  context-free syntax
    "<" TYPE " ," VAR ">" -> INSTANT-OUT
    instant "(" GEN-TYPE " ," VAR ")" -> INSTANT-OUT
    generalize "(" TYPE " ," TYPE-ENV ")" -> GEN-TYPE

hiddens
  sorts INS-OUT
  context-free syntax
    "<" TYPE " ," VAR " ," VAR-SUBS ">" -> INS-OUT
    ins-subs "(" GEN-TYPE " ," VAR " ," VAR-SUBS ")" -> INS-OUT
  variables
    n -> CHAR+
    subs [0-9]* -> VAR-SUBS

equations
[50] instant(gt, var) = <type, var1>
      when ins-subs(gt, var, []) = <type, var1, subs>
[51] generalize(var, env) = var
      when var elm env = true
[52] generalize(lex-var("s" n), env) = lex-gen-var("b" n)
      when lex-var("s" n) elm env = false
[53] generalize(bool, env) = bool
[54] generalize(nat, env) = nat
[55] generalize(type1 -> type2, env)
      = generalize(type1, env) -> generalize(type2, env)
[56] generalize(type1 # type2, env)
      = generalize(type1, env) # generalize(type2, env)
[57] ins-subs(type, var, subs) = <type, var, subs>
[58] lookup gv in subs = <false, var1>
      =====
      ins-subs(gv, var, subs)
      = <next(var), next(var), (gv, next(var)) + subs>
[59] lookup gv in subs = <true, var1>

```

```

=====
ins-subs(gv, var, subs) = <var1, var, subs>
[60]   ins-subs(gt1, var, subs) = <type1, var1, subs1>,
       ins-subs(gt2, var1, subs1) = <type2, var2, subs2>
=====

ins-subs(gt1 -> gt2, var, subs)
= <type1 -> type2, var2, subs2>
[61]   ins-subs(gt1, var, subs) = <type1, var1, subs1>,
       ins-subs(gt2, var1, subs1) = <type2, var2, subs2>
=====

ins-subs(gt1 # gt2, var, subs)
= <type1 # type2, var2, subs2>

```

4.3.3.5 Type-Equations

This module defines type equations (sort `TYPE-EQ`) and lists of type equations (sort `TYPE-EQS`). One or more type equations can be unified. The result of the unification of a type equation or a list of type equations is a Boolean and a type substitution. The Boolean is `true` if the unification succeeds and `false` otherwise. The type substitution is the minimal substitution which has to be made in the equation(s) in order to solve them. In this module a hidden function is needed that applies a type substitution to a list of type equations.

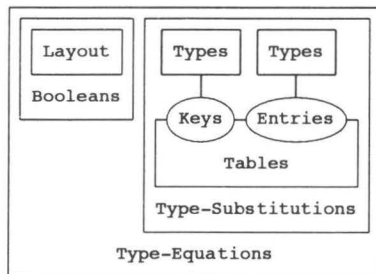


Fig. 4.11. Structure diagram of Type-Equations

```

module Type-Equations
imports Booleans Type-Substitutions
exports
  sorts TYPE-EQ TYPE-EQS UNIFY-OUT
  context-free syntax
    TYPE is TYPE                -> TYPE-EQ
    "{" {TYPE-EQ " ," }* "}"    -> TYPE-EQS
    "<" BOOL " ," TYPE-SUBS ">" -> UNIFY-OUT
    unify "(" TYPE-EQ ")"       -> UNIFY-OUT

```

```

    unify "(" TYPE-EQS ")"      -> UNIFY-OUT

hiddens
  context-free syntax
    apply "(" TYPE-SUBS "," TYPE-EQS ")" -> TYPE-EQS
  variables
    teq          -> TYPE-EQ
    pairs [0-9]* -> {TYPE-EQ ", "}*

equations

[62] unify(var1 is var2) = <true, [(var1, var2)]>
[63] unify(var is bool)  = <true, [(var, bool)]>
[64] unify(var is nat)   = <true, [(var, nat)]>
[65] unify(var is type1 -> type2) = <false, []>
      when var elm type1 -> type2 = true
[66] unify(var is type1 -> type2)
      = <true, [(var, type1 -> type2)]>
      when var elm type1 -> type2 = false
[67] unify(var is type1 # type2) = <false, []>
      when var elm type1 # type2 = true
[68] unify(var is type1 # type2)
      = <true, [(var, type1 # type2)]>
      when var elm type1 # type2 = false
[69] unify(bool is var)          = <true, [(var, bool)]>
[70] unify(bool is bool)         = <true, []>
[71] unify(bool is nat)          = <false, []>
[72] unify(bool is type1 -> type2) = <false, []>
[73] unify(bool is type1 # type2) = <false, []>
[74] unify(nat is var)           = <true, [(var, nat)]>
[75] unify(nat is bool)          = <false, []>
[76] unify(nat is nat)           = <true, []>
[77] unify(nat is type1 -> type2) = <false, []>
[78] unify(nat is type1 # type2)  = <false, []>
[79] unify(type1 -> type2 is var)
      = unify(var is type1 -> type2)
[80] unify(type1 -> type2 is bool) = <false, []>
[81] unify(type1 -> type2 is nat)  = <false, []>
[82] unify(type1 -> type2 is type3 -> type4)
      = unify({type1 is type3, type2 is type4})
[83] unify(type1 -> type2 is type3 # type4)
      = <false, []>
[84] unify(type1 # type2 is var)
      = unify(var is type1 # type2)
[85] unify(type1 # type2 is bool) = <false, []>
[86] unify(type1 # type2 is nat)  = <false, []>
[87] unify(type1 # type2 is type3 -> type4)

```

```

= <false, []>
[88] unify(type1 # type2 is type3 # type4)
    = unify({type1 is type3, type2 is type4})
[89] unify({}) = <true, []>
[90]   unify(teq) = <bool1, subs1>,
      unify(apply(subs1, {pairs})) = <bool2, subs2>
=====
      unify({teq, pairs}) = <bool1 & bool2, subs2 o subs1>
[91] apply(subs, {}) = {}
[92]   apply(subs, {pairs1}) = {pairs2}
=====
      apply(subs, {type1 is type2, pairs1})
      = {apply-type(subs, type1) is apply-type(subs, type2),
         pairs2}

```

4.3.4 The typechecker of Mini-ML

Before the specification of the typechecker is finally given in Section 4.3.4.2, the syntax of type errors is first specified in 4.3.4.1.

4.3.4.1 Type-Errors

The following module defines the syntax of the error messages which are given when respectively:

- An identifier is not bound by a lambda-abstraction, a declaration or a recursive declaration.
- An application cannot be typed because the type of the first expression is such that it cannot “eat” the type of the expression to which it is applied.
- It is not possible to unify the type of the identifier and its occurrences in the declaration-part of a recursive declaration.
- The test in an if-then-else-fi expression is not of type *bool*.
- The types of the then- and else-part in an if-then-else-fi expression cannot be unified.

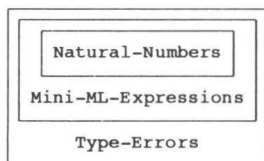


Fig. 4.12. Structure diagram of Type-Errors

```

module Type-Errors
imports Mini-ML-Expressions
exports
  sorts ERROR ERRORS
  context-free syntax
    unbound identifier ID          -> ERROR
    incorrect application EXP      -> ERROR
    incorrect recursive declaration of ID -> ERROR
    incorrect test EXP             -> ERROR
    then EXP incompatible with else EXP -> ERROR
    "[" {ERROR " , "}* "]"        -> ERRORS
  variables
    ers [0-9]* -> {ERROR " , "}*
    errors      -> ERRORS

```

4.3.4.2 Mini-ML-Typecheck

Module Mini-ML-Typecheck exports the function `check`, which returns `[]` if a given Mini-ML expression is typeable and a non-empty list of errors if it is not. However, the hidden function `check` is the most important function of this module. The arguments of the latter are:

- A syntactically correct Mini-ML expression.
- A type environment in which the expression has to be checked. This environment should at least contain the type of the identifiers in the expression which are not bound by a lambda-, a let- or a letrec-construction.
- A type variable, which is the last type variable used in typechecking.

The output of `check` is:

- the most general type of the expression;
- a type substitution containing the changes in the type environment due to unification;
- the list of errors found in the expression;
- the last type variable used in the typechecking process.

```

module Mini-ML-Typecheck
imports
  Mini-ML-Expressions
  Types Type-Errors Type-Substitutions Type-Environments
  Type-Instant-Generalize Type-Equations
exports
  context-free syntax

```

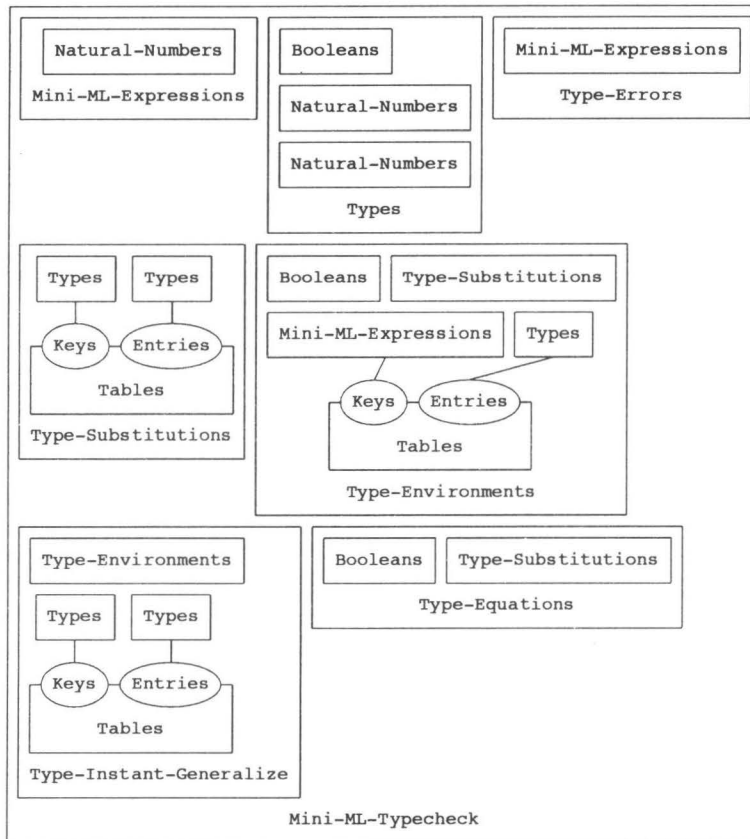


Fig. 4.13. Structure diagram of Mini-ML-Typecheck

```

check "(" EXP ")" -> ERRORS

hiddens
  sorts CHECK-OUT
  context-free syntax
    "<" TYPE "," TYPE-SUBS "," ERRORS "," VAR ">"
    check "(" EXP "," TYPE-ENV "," VAR ")" -> CHECK-OUT
    if BOOL then CHECK-OUT else CHECK-OUT fi -> CHECK-OUT
  variables
    found -> BOOL
    unifiable [0-9]* -> BOOL
    Nat -> LEX-NAT
    check-out [0-9]* -> CHECK-OUT

```

```

equations

[93]  check(Exp) = errors
      when check(Exp, [], s0) = <type, subs, errors, var>

[94]  check(true, env, var) = <bool, [], [], var>
[95]  check(false, env, var) = <bool, [], [], var>
[96]  check(Nat, env, var) = <nat, [], [], var>
[97]  lookup Id in env = <found, gt>,
      instant(gt, var) = <type, var1>
=====
check(Id, env, var)
= if found
  then <type, [], [], var1>
  else <next(var1),
      [],
      [unbound identifier Id],
      next(var1)>
fi
[98]  check(Exp1, env, var)
      = <type1, subs1, [ers1], var1>,
      check(Exp2, apply(subs1, env), var1)
      = <type2, subs2, [ers2], var2>,
      unify(apply-type(subs2, type1) is
            type2 -> next(var2))
      = <unifiable, u-subs>
=====
check((Exp1 Exp2), env, var)
= if unifiable
  then <apply-type(u-subs, next(var2)),
      u-subs o subs2 o subs1,
      [ers1, ers2],
      next(var2)>
  else <next(var2),
      [],
      [ers1,
      ers2,
      incorrect application (Exp1 Exp2)],
      next(var2)>
fi
[99]  check(Exp, (Id, next(var)) + env, next(var))
      = <type, subs, errors, var1>
=====
check(lambda Id . Exp, env, var)
= <apply-type(subs, next(var)) -> type,
  subs,
  errors,

```

```

    var1>
[100]  check(Exp1, env, var)
      = <type1, subs1, [ers1], var1>,
      apply(subs1, env) = env1,
      generalize(type1, env1) = gt,
      check(Exp2, (Id, gt) + env1, var1)
      = <type2, subs2, [ers2], var2>
=====
      check(let Id = Exp1 in Exp2, env, var)
      = <type2, subs2 o subs1, [ers1, ers2], var2>
[101]  check(Exp1, (Id, next(var)) + env, next(var))
      = <type1, subs1, [ers1], var1>,
      unify(type1 is apply-type(subs1, next(var)))
      = <unifiable, u-subs>,
      apply(u-subs o subs1, env) = env1,
      generalize(apply-type(u-subs, type1), env1) = gt,
      check(Exp2, (Id, gt) + env1, var1)
      = <type2, subs2, [ers2], var2>
=====
      check(letrec Id = Exp1 in Exp2, env, var)
      = if unifiable
        then <type2,
              subs2 o u-subs o subs1,
              [ers1, ers2],
              var2>
        else <next(var2),
              [],
              [ers1,
               incorrect recursive declaration of Id,
               ers2],
              next(var2)>
        fi
[102]  check(Exp1, env, var)
      = <type1, subs1, [ers1], var1>,
      unify(type1 is bool)
      = <unifiable1, u-subs1>,
      check(Exp2, apply(u-subs1 o subs1, env), var1)
      = <type2, subs2, [ers2], var2>,
      check(Exp3,
            apply(subs2 o u-subs1 o subs1, env),
            var2)
      = <type3, subs3, [ers3], var3>,
      unify(apply-type(subs3, type2) is type3)
      = <unifiable2, u-subs2>
=====
      check(if Exp1 then Exp2 else Exp3 fi, env, var)

```

```

= if unifiable1
  then if unifiable2
    then <apply-type(u-subs2, type3),
      u-subs2 o subs3 o subs2 o u-subs1 o subs1,
      [ers1, ers2, ers3],
      var3>
    else <next(var3),
      [],
      [ers1, ers2, ers3,
        then Exp2 incompatible with else Exp3],
      next(var3)>
  fi
else if unifiable2
  then <next(var3),
    [],
    [ers1, incorrect test Exp1, ers2, ers3],
    next(var3)>
  else <next(var3),
    [],
    [ers1, incorrect test Exp1, ers2, ers3,
      then Exp2 incompatible with else Exp3],
    next(var3)>
  fi
fi
[103] check(Exp1, env, var)
      = <type1, subs1, [ers1], var1>,
      check(Exp2, apply(subs1, env), var1)
      = <type2, subs2, [ers2], var2>
=====
check((Exp1, Exp2), env, var)
= <apply-type(subs2, type1) # type2,
  subs2 o subs1,
  [ers1, ers2],
  var2>

[104] if true then check-out1 else check-out2 fi = check-out1
[105] if false then check-out1 else check-out2 fi = check-out2

```

The specification is illustrated using the example given in Section 4.2.2.2:

1. $\text{check}(\lambda x. \lambda y. (x (x y))) = ???$
 1. $\text{check}(\lambda x. \lambda y. (x (x y)), [], s0) = ???$
 1. $\text{check}(\lambda y. (x (x y)), [(x, s1)], s1) = ???$
 1. $\text{check}((x (x y)), [(y, s2), (x, s1)], s2) = ???$
 1. $\text{check}(x, [(y, s2), (x, s1)], s2) =$
 $\langle s1, [], [], s2 \rangle$

```

2. check((x y), apply([], [(y, s2), (x, s1)]), s2) =
   check((x y), [(y, s2), (x, s1)], s2) = ???
1. check(x, [(y, s2), (x, s1)], s2) =
   <s1, [], [], s2>
2. check(y, apply([], [(y, s2), (x, s1)]), s2) =
   check(y, [(y, s2), (x, s1)], s2) =
   <s2, [], [], s2>
3. unify(apply-type([], s1) is s2 -> next(s2)) =
   unify(s1 is s2 -> next(s2)) =
   <true, [(s1, s2 -> s3)]>
2. check((x y), [(y, s2), (x, s1)], s2) =
   <apply-type([(s1, s2 -> s3)], next(s2)),
   [(s1, s2 -> s3)] o [] o [],
   [],
   next(s2)> =
   <s3, [(s1, s2 -> s3)], [], s3>
3. unify(apply-type([(s1, s2 -> s3)], s1) is
   s3 -> next(s3)) =
   unify(s2 -> s3 is s3 -> s4) =
   <true, [(s2, s4), (s3, s4)]>
1. check((x (x y)), [(y, s2), (x, s1)], s2) =
   <apply-type([(s2, s4), (s3, s4)], next(s3)),
   [(s2, s4), (s3, s4)] o [(s1, s2 -> s3)] o [],
   [],
   next(s3)> =
   <s4, [(s1, s4 -> s4), (s2, s4), (s3, s4)], [], s4>
1. check(lambda y. (x (x y)), [(x, s1)], s1) =
   <s4 -> s4, [(s1, s4 -> s4), (s2, s4), (s3, s4)], [], s4>
1. check(lambda x. lambda y. (x (x y)), [], s0) =
   <(s4 -> s4) -> (s4 -> s4),
   [(s1, s4 -> s4), (s2, s4), (s3, s4)],
   [],
   s4>
1. check(lambda x. lambda y. (x (x y))) =
   []

```

4.4 Implementations of the Mini-ML specification

From the above Mini-ML specification and several of its predecessors, implementations have been derived. To this end, the algebraic specification is viewed as a term rewriting system by interpreting the conclusion of each (conditional) equation as a rewrite rule from left to right. A generated implementation can rewrite a closed expression (an expression without variables) into its normal form, where the normal forms are implicitly defined by the term rewriting system. Such implementations are not general equation solvers and some of them cannot reduce open expressions in an appropriate way.

In the process of generating an implementation from a specification written ASF+SDF we can distinguish the following steps:

- *Remove syntax:*

First, we remove the user-defined syntax from the specification in order to create a specification written in “pure” ASF. This amounts to changing all SDF descriptions into corresponding signatures and parsing the equations such that only equations with terms over these signatures remain. The result of this step is a modular ASF specification. In the specification enough syntax is added to ensure that each equation can be parsed unambiguously. In general, the syntax defined by the user may be ambiguous and in this case some of the equations may even be multi-interpretable.

- *Normalize:*

Next, we normalize this ASF-specification, i.e., we remove all modular structure from module `Mini-ML-Typecheck` until a module without imports remains. This normalization strategy is described in [BHK89b]. This step results in an algebraic specification (a signature with conditional equations) without modular structure.

- *Transform to a term rewriting system:*

The normalized ASF-specification of the module has to be transformed into a (conditional) term rewriting system. We have to give a direction to each of the equations. Depending on the strategy chosen to implement the term rewriting system, the resulting rewrite rules have to satisfy some constraints. Consequently, it might be necessary to modify the specification such that it obeys these constraints. The specification in this chapter leads to a confluent, and terminating term rewriting system when all conclusions of equations are interpreted as rewrite rules from left to right.

- *Implement the term rewriting system:*

Finally, the resulting term rewriting system can be implemented.

Until now, I have experimented with the following implementations:

- *Equation Interpreter*

The Equation Interpreter [HOD82, ODo85] (see also [BW89]) was used to generate the first implementation of the typechecker for Mini-ML. Its input is a single-sorted algebraic specification which is interpreted as a rewriting system. The specification had to be rewritten considerably because the Equation Interpreter cannot handle conditional equations and overloading of function symbols is forbidden. An advantage of the system is that a specification accepted by the system is guaranteed to be confluent.

- *C-Prolog*

Faster code was obtained by translating a normalized version of the specification to C-Prolog [PWBBP85]. First, the method described by Drosten and Ehrich in [DE84] which implements a leftmost innermost reduction strategy was used. This version was improved by implementing a parallel outermost strategy as described in ([BW89] (pages 219-221) and [Die86]). The latter version terminates even if the term rewriting system is only weakly terminating (each term has a normal form). In both cases, the C-Prolog code was created by hand and overloaded function symbols needed to be disambiguated.

- *ASF system*

The above-mentioned experiments in translating a term rewriting system into C-Prolog code resulted in the ASF system (see Chapter 2). This system generates C-Prolog code for an algebraic specification written in ASF. If we want to implement the Mini-ML specification given above, the syntax has to be removed from it. The system checks the specification and generates code for all its modules. The main drawback of this system is the fact that it is a batch-oriented system. A lot of work has to be (re)done after each modification of a specification.

- *ASF+SDF system*

Finally, the Mini-ML specification was implemented in the ASF+SDF system (see Chapter 5). This is an interactive system to manipulate specifications written in ASF+SDF. The current version of the system cannot yet handle renamings and parameter bindings in imports. We have to work around this limitation of the system by working out these kind of imports by hand. The system provides a nice environment in which the specification can easily be modified and tested by rewriting terms to their normal form.

4.5 Further research

The work described here raised several questions that are not addressed in this thesis and require further research:

- While creating the specification for the typechecker experience was gained with a combination of the formalisms ASF and SDF. Research has to be done to further improve both formalisms and to design the combination of them carefully to achieve a convenient specification formalism with a solid theoretical base.
- There may be profit in making other (algebraic) specifications of typecheckers for ML (or Mini-ML).

These questions will now be discussed in somewhat more detail.

- The hiding mechanism should be improved:
 - In ASF, the exported signature of each imported module is automatically included in the export signature of the importing module (see [BHK89b]). As a consequence, module `Mini-ML-Typecheck` not only exports sorts like `EXP` and `ERRORS`, but also unintended sorts like `TYPE-EQ` and `UNIFY-OUT`. It is not possible to hide the specification of type variable substitutions in module `Type-Instant-Generalize` if we want to reuse module `Tables` in its definition.
 - Sort `PAIR` in module `Tables` is an example of a sort which is needed to define the grammar of the module. We do not want to change the internal structure of such a sort and nor do we want to write equations over it. It is not possible to hide these sorts because in that case all functions in the `exports` section of the module which use these sorts must be hidden too. The hiding mechanism of the formalism should be augmented or extended to give the user of the specification formalism the possibility to express these kinds of properties of sorts.
- It is not possible to give a generic specification of a unification algorithm in ASF that has an arbitrary signature as parameter. This would require the introduction of some kind of higher-order signatures.
- In combining ASF and SDF the question has to be answered what is to be done with ambiguous syntax definitions (see also Section 4.3.3.2). One of the major problems of ambiguities is that it is in general undecidable whether a syntax definition is ambiguous [HU79, p. 200]. Ambiguities of terms of different sorts should be permitted by the formalism because it would be unpleasant if the user of the formalism would, for instance, have to come up with extra syntax for zero as a constant of type natural number, integer, rational number or real.

Ambiguity of terms within a sort is more hazardous because it is not clear which interpretation of the term is intended by the user. Restricting the formalism to unambiguous interpretations of each term forces the addition of extra syntax to the specification (see also the discussion on the suffixes `-type` and `-gt` in Section 4.3.3.2).

- Is it possible to generate an *incremental* typechecker from the specification? In the meantime, it has turned out that this question has an affirmative answer. The specification of Mini-ML given here clearly falls in the realm of extensions of primitive recursive schemes for which an incremental implementation can be derived (see [Meu90]).
- The specification includes not only all information on *what* a typechecker for Mini-ML should do but it also states *how* it should be done in great detail. It would be nice if the specification would only define what typechecking is and leave the algorithmic details to the implementor of the specification.
- The specification would be considerably shorter if the typechecker was only partially specified. We could, as is done in TYPOL [CDDK86, Kah87], specify which expressions of Mini-ML are correct. In [HK89a] something similar is done for the typechecker of the toy programming language PICO by using inequalities in conditions. It is, however, necessary to specify which expressions are incorrect if we want to specify the error messages as is done in the specification. An alternative approach that eliminates the need to specify which expressions are incorrect could be to add a mechanism to ASF+SDF that generates an error message when the typechecking of an expression does not yield true. Such error messages could either be given explicitly in the specification itself, or they could be derived automatically from it.

The ASF+SDF System

The ASF+SDF system, an interactive system to manipulate specifications written in ASF+SDF, is described.

5.1 Introduction

The user-interface and global architecture of the ASF+SDF system are described. This system supports development and testing of specifications written in the combination of ASF [BHK89a] and SDF [HK89b, HHKR89] (see also Section 1.4). The system is highly incremental: the generated implementation is updated after each editing operation on the specification. The specified syntax can be tested immediately in a syntax-directed editor. The abstract syntax tree corresponding to the text in such an editor can be evaluated using the equations from the specification as rewrite rules. The user is warned whenever typechecking errors occur in the specification. Tracing facilities are provided which help to debug specifications.

The ASF+SDF system has the following functionality:

- *Creation:*
A specification is entered in the system either by reading in an already existing specification or by creating the first module of a new one (addition of a module).
- *Modification:*
Modifications to a specification can be made by editing the text of individual modules, or by adding or deleting modules.
- *Testing:*
For each module in the specification one or more syntax-directed editors can be invoked to create and evaluate terms. These editors use the syntax of the module for parsing the textual representation of terms and converting them into abstract syntax trees. The equations of the module are then used to rewrite these abstract syntax trees into normal form.
- *Saving:*
After modifying a specification it can be saved in files.

The main qualities of the ASF+SDF system which distinguish it from other systems are:

- *Uniform interface:*
To modify either the syntax or the semantics of a module or to test the module, instances of the generic syntax-directed editor GSE are used. This improves the uniformity of the user-interface of the ASF+SDF system.
- *Incrementality:*
After each modification of the specification, the generated implementation is updated. If, for example, a rule in the syntax part of a module is deleted, all editors which may be influenced by its deletion are warned. This concerns not only the syntax-directed editor in which terms for that module are edited, but also the editor for the equations of that module, and all editors of modules in which the changed module is imported.

The implementation of the system is not yet completely finished, but the current system has already been used to write and test several specifications:

- the typechecker for Mini-ML (see Chapter 4),
- modularization (in Chapter 6),
- the syntax and static semantics of the formal specification language LOTOS [JJWW90],
- a program generator which generates database-oriented application programs written in Fortran [Laa90], and
- the static semantics of Pascal [Deu91].

The user interface of the system is described in more detail in the following section. Next, the definition of the syntax of the formalism is given in Section 5.3 and the internal structure of the system is presented in Section 5.4. The chapter is concluded with Section 5.5 containing a discussion on further development of the system.

5.2 User interface

In the following sections a description is given of how to start the system (in Section 5.2.1), and how to use the generic syntax-directed editor GSE (Section 5.2.2). Some miscellaneous remarks are captured in Section 5.2.3.

5.2.1 Starting the system

The ASF+SDF system is a part of the CENTAUR system [CENT89], which runs in an X-window environment. A specialized version of CENTAUR has to be started

by typing the command `ctasdf`. As a result, LeLisp [Lisp87] is started and a small menu is created.

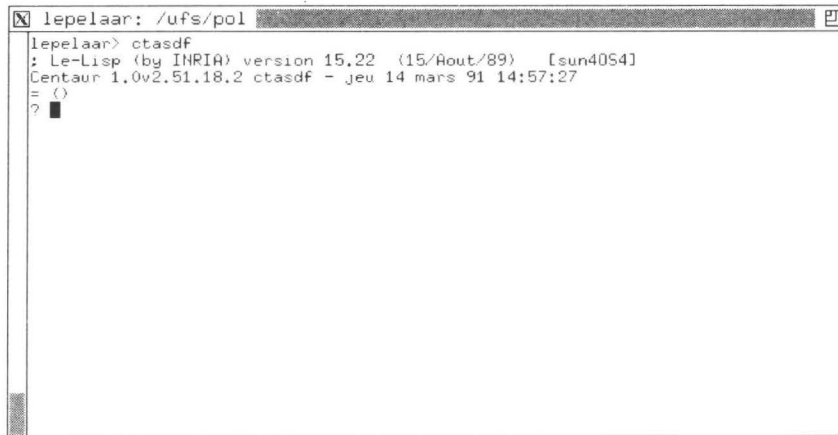


Fig. 5.1. LeLisp window after starting the CENTAUR system

The LeLisp window (see Figure 5.1) can be used to communicate with LeLisp but this will be unnecessary in most cases. It is also used to print messages and to show the results of evaluation.

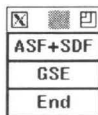


Fig. 5.2. Menu created after starting the CENTAUR system

The small menu (see Figure 5.2) contains three buttons which indicate different possibilities:

- **ASF+SDF:**
After pressing this button, the ASF+SDF system itself is started and the user can start developing specifications in ASF+SDF.
- **GSE:**
Pressing this button will result in starting a stand-alone GSE.
- **End:**
To leave LeLisp this button should be pressed. A dialog will ask for confirmation.

Any of these buttons may be chosen and it is, for instance, possible to invoke the ASF+SDF system more than once. It is, however, not advisable to do this as it will be difficult to find out which editors of terms or modules belong to which system.

After choosing the ASF+SDF button in the menu a new window (see Figure 5.3) is created which contains a menu bar and an area in which error messages will be printed.

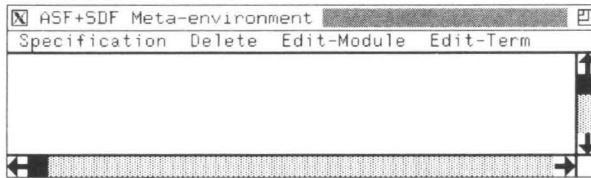


Fig. 5.3. Window of the ASF+SDF System

The first menu called *Specification* contains four buttons, which can be used to add a module to the specification (*add*), to clear all information in the system and re-initialize it (*clear*), to save all texts (*save*), or to quit the system (*quit*). The precise meaning of the buttons in the *Specification* menu is:

- **add:**
After pressing this button, the user is asked for the name of a directory and of a module (see Figure 5.4). If the module name is correct (see the syntax definition of *Id* in Section 5.3.1) and if no module with that name exists in the system, the files belonging to it are searched for in the given directory. If those files can be found, the module and all modules in its import graph which are not already in the system will be loaded. If those files cannot be found, the user is asked whether a new module is to be created. An editor containing a template of the specification of a module (see also Section 5.2.2 and Figure 5.5) is started. The files belonging to the new module will be created in the given directory when the user saves its text.
- **clear:**
The system returns to the initial situation. If the system contains modules whose changes have not yet been saved, the user is asked whether they should be saved or not.
- **save:**
All modules in the system are saved on file.
- **quit:**
Before leaving the system, the user is asked whether changes in modules should be saved.

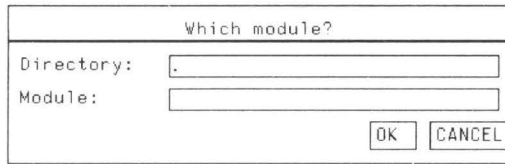


Fig. 5.4. Dialog box asking for directory and module name

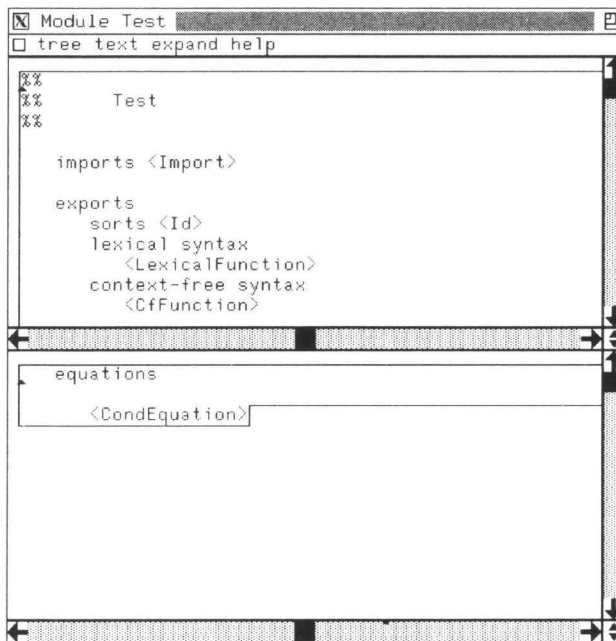


Fig. 5.5. Editor with template of just created module named Test

The three other menus (**Delete**, **Edit-Module**, and **Edit-Term**) always contain the names of all modules currently in the system. These names are presented in alphabetic order. After pressing a button in these menus, the corresponding module is deleted from the system (**Delete**), an editor is started containing the text of the specification (**Edit-Module**), or an editor is started containing a term for the module (**Edit-Term**).

Deleting a module means that it is removed from the specification in the ASF+SDF system. Its corresponding files, however, are not removed from the file system. Therefore, the user is asked whether changes should be saved that have been made to either the module itself or to the terms belonging to it when deleting a module.

For each module in the specification an editor containing the text of the module can be created. Such an editor is called a *module editor*. When first selecting an entry in the menu `Edit-Module`, the module editor (see, for example, Figures 5.5 and 5.10) is created. Subsequent selections of that entry will only pop up the window of it.

To test the specification of a module, a so-called *term editor* can be used. The text of such an editor is parsed using the syntax as defined in the selected module. Each term editor contains a button named `reduce` (see also Figure 5.6) which reduces the abstract syntax tree of the text to normal form using the equations of the module. Each selection of an entry in the `Edit-Term` menu will give a new term editor for that module. Hence, several term editors are allowed per module. The user is always asked to provide the filename and the name of the directory in which the text of the term to be reduced can be found. If the given file cannot be found, a term editor containing an empty text is started.

5.2.2 Editing

All editing of text in the ASF+SDF system is done with the generic syntax-directed editor GSE (see also Section 5.4.4). It is an editor in which textual and structural editing are highly integrated. It does not do any prettyprinting of text as is done in many other syntax-directed editors like the editor in the Synthesizer Generator [RT89] and *ctedit* in CENTAUR [CI89a].

The structure of the edited text is shown using the *focus*. It is a region of text corresponding to the smallest structure (abstract syntax tree) containing a certain character when pointing at that character in a syntactically correct text. The focus is displayed by drawing lines around it (see Figure 5.6). Textual modifications can only be done inside the focus. The focus can be moved by pointing at a character in the text, or by structural movements like `go up in tree`, `go down`, `go to next child`, and `go to previous child`. In all these circumstances, the text in the focus is parsed. If it is syntactically correct, the focus is moved as intended. If it cannot be parsed, the user is warned (see also Figure 5.7) and the new position of the focus depends on how the parse of text was initiated:

- If the parse was the result of pointing at a character in the text, the focus is put on the smallest tree whose corresponding text encloses the old focus as well as the character pointed at.
- When giving commands to move up in the tree, the focus is indeed put on the father of the old focus.
- In all other cases (going down, to next child, or to previous child), the focus is left unchanged.

The consequence of this strategy is that all text *outside* the focus is always syntactically correct and the text *in* the focus might be incorrect.

The edited text may contain *meta-variables*. These are strings of the form `<Sort>` which are defined for each sort `Sort` in the specification of the used syntax. If the focus is positioned at a meta-variable, the `expand` menu in the editor provides all possible expansions of that meta-variable. If any of these possibilities is chosen, the meta-variable is replaced by the selected alternative. Any non-terminals appearing in the alternative are again represented by meta-variables. In this way, the user who is not familiar with the syntax of the used language is helped. This also provides an elegant way to construct programs in a top-down fashion. If the user starts typing if the focus is positioned at a meta-variable, it disappears automatically.

In Figure 5.6, an instance of GSE is shown. It is an example of a term editor of the module `Booleans` as given in Section 4.3.1.2.

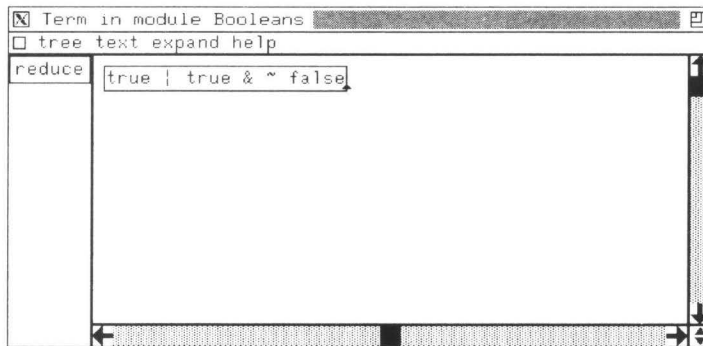


Fig. 5.6. Term editor of module `Booleans`

In the window, the text and focus are displayed. Initially, the focus is placed on the complete text. The top of the window is a menu bar containing the entries `□`, `tree`, `text`, `expand`, and `help`. On the bottom and the right of the text, two scroll bars are shown. To the left of the text, an area is displayed which contains the `reduce` button. The functionality of the different parts of the window will now be described.

When clicking in the window containing the text and focus, GSE tries to put the focus on the smallest tree containing the character at which the user pointed. As described before, the latter is augmented with the text of the old focus if it could not be parsed. All characters the user types are inserted before the cursor which will always be inside the focus. Movements of the cursor in the focus can be achieved by using the arrows in the right panel of the keyboard or by pointing at the wanted position as usual. A piece of text can be selected by dragging with the mouse from

one position to another. The selected text is shown in inverse video and can be used to cut, or paste (using entries from the text menu).

The menu bar at the top of the window provides the following functionality:

- ☐:
To leave the editor, this button should be pressed. If the edited text is changed, the user is asked whether it should be saved on file.
- tree:
This menu contains entries which deal with structural manipulations. When choosing any of these entries, the text in the focus will be parsed before carrying out the selected operation.
 - zoom in <cntrl F>:
After a successful parse of the text in focus, the focus is put on the first child of the resulting tree if it has children at all. If no children are present, the focus is left unchanged.
 - zoom out <cntrl U>:
This is more or less the inverse of the previous one: the focus is placed on the father of the tree which was in focus. The focus is unchanged if the tree has got no father, i.e., if the focus is on the complete text.
 - next child <cntrl N>:
The focus is moved to the next child with the same father. It is left unchanged if either the text of the focus cannot be parsed, the current tree is the last child of its father, or if it has got no father.
 - previous child <cntrl P>:
The focus is moved to the previous child with the same father, provided that this is possible.
 - insert hole after <cntrl A>:
If the focus is an element of a list, a meta-variable for a new element of this list is inserted next to it. If required by the syntax of the language, a separator is inserted in the text. The position of the meta-variable and possible separator in the text are determined using heuristics on the positions of the other elements of the list. If the focus is not an element of a list, the first such ancestor is searched. A meta-variable and possible separator are then inserted next to this ancestor. Nothing happens if either the text in focus cannot be parsed or if such an ancestor cannot be found.
 - insert hole before <cntrl B>:
This operation is identical to the previous one except that the meta-variable and the possible separator are inserted before the current tree.

- **text:**

In this menu, entries dealing with textual manipulations are grouped:

- **cut <ctrl C>:**

If part of the text is selected (by dragging with the mouse from one position to another), it is removed. If no text is selected, the text of the current focus is removed.

- **paste <ctrl V>:**

The last selected text from this window or any other X-window is pasted in the text at the current position of the cursor.

- **expand:**

The expand menu contains varying entries. If the focus is positioned at a meta-variable, it contains all possible context-free syntax functions, and output sorts of lexical functions which may be substituted for that meta-variable. If a context-free syntax function is chosen, the meta-variable is replaced by its terminals and non-terminals. If the output sort of a lexical function is chosen, the meta-variable disappears and the user can enter the desired lexeme. If the focus is not positioned at a meta-variable, the expand menu contains the message **Non expandable sort: Sort** where **Sort** is the sort of the tree in the focus.

- **help:**

Some miscellaneous functions of GSE are gathered in this menu.

- **undo:**

As much as possible of the last user action is undone.

- **cursor to error:**

The cursor is moved to the last position where a parse error occurred.

- **show cursor:**

The cursor is shown at its current position in the middle of the window. This operation is particularly useful if the cursor is lost due to the scrolling of text.

In some entries of the above menus an indication **<ctrl x>** where **x** is some capital letter is given. It means that holding the **Control** key and typing the announced letter **x** or **x** is an abbreviation for choosing that entry of the menu. This gives a shortcut for advanced users of the editor.

The two scroll bars at the bottom and the right of the window have the following functionality. The black square in these bars indicates the position of the envisaged part of the text relative to the complete text. If the square is dragged to another position in the gray area of the scroll bar, the corresponding part of the text is visualized in the window. Clicking in the gray area has the effect of scrolling one page in the desired direction. When clicking in one of the squares at the end of the scroll bar which contain an arrow, the text is moved either one character to the left or right (the horizontal scroll bar) or one line up or down (the vertical scroll bar). These

actions are repeated as long as the mouse is pressed in such a square. The square in the corner of the two scroll bars which is filled with two little, black triangles can be used to resize the window. When clicked in the lower triangle, the window is enlarged with one extra line. A click in the upper triangle results in diminishing the window with one line of text.

When clicking at the `reduce` button in the column left of the text, the edited text is parsed and the abstract syntax tree is evaluated. The equations of the module to which the term editor belongs are used as rewrite rules to rewrite the term (the abstract syntax tree) to normal form. The result of evaluation is currently displayed in the LeLisp window (see Figure 5.1) which is shown when creating the system. Other buttons, whose functionality can be described in the configuration file (see Section 5.2.3.2), can be added to this default `reduce` button.

If the text cannot be parsed an error window as shown in Figure 5.7 pops up.

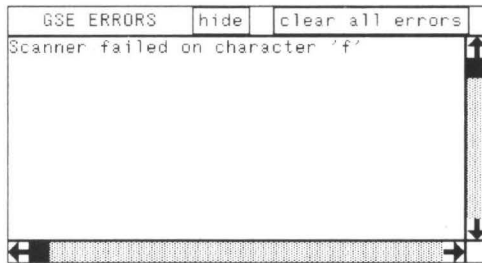


Fig. 5.7. Error window shown if parse fails

This window, for example, appears if the `s` in `false` in the term editor of Figure 5.6 is removed and if parsing of the text is forced by pointing at some character. The cursor to error entry in the help menu helps the user to locate the place where the error is detected. The error window disappears as soon as the user resumes entering text. It also disappears if the `hide` button in the menu bar of the window is clicked. Clicking the `clear all errors` button results in the removal of the message(s) displayed in the error window.

If the parsed text turns out to be ambiguous, a window as shown in Figure 5.8 appears containing a menu to disambiguate the text.

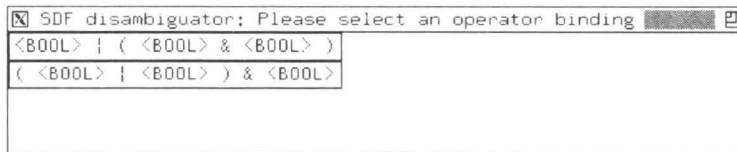


Fig. 5.8. Window containing disambiguation menu

This window, for example, appears if the priority declaration in the specification of module `Booleans` is removed and if the whole text is parsed. The two possible parse trees of the text are visualized in Figure 5.9.

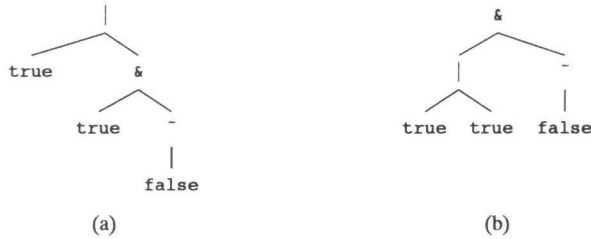


Fig. 5.9. Parse trees of `true | true & ~ false`

If the first entry

`<BOOL> | (<BOOL> & <BOOL>)`

in the disambiguation menu is chosen, parse tree (a) is preferred. Parse tree (b) results from choosing the second entry

`(<BOOL> | <BOOL>) & <BOOL>.`

In Figure 5.10 the module editor of module `Booleans` as presented in Section 4.3.1.2 is shown. The functionality of a module editor is a combination of two generic syntax-directed editors. Each module of an ASF+SDF specification is split in two parts:

- *Syntax part:*
It contains the specification of the syntax and the imports. These can be edited in the upper part of the module editor. For a module named `Mod`, this part is stored in a file named `Mod.syn`.
- *Equations part:*
This part contains the equations defined in this module and can be modified in the lower part of the module editor. It is stored in a file named `Mod.eqs` if the name of the module is `Mod`.

The syntax of the syntax part and the equations part of a module are described in Sections 5.3.1 and 5.3.2, respectively.

The functionality of the menu bar in a module editor is identical to that of a term editor as described before. The only difference is that every choice of a menu in the menu bar applies to the last part in which a mouse click was registered. So, if the user clicks in the equations part of a module editor and chooses the undo from the help menu, the last action from the equations part is undone.



Fig. 5.10. Module editor of module `Booleans`

If the syntax part of a module is changed and the text of the modification is parsed successfully, that change is processed by the system immediately. As a consequence, all editors whose syntax has been influenced by that modification are warned. Normally, this means that all equations parts in module editors, and all term editors of modules which import the module in which the modification was made are warned. Only if the modification just influenced the hiddens of the module, the equations part in the module editor and the term editors of that module are warned. The current version of the editor responds to such a warning by zooming out until the whole text fits in the focus. It assumes that this text is possibly syntactically incorrect.

When text in the equations part of a module editor or in a term editor has to be parsed, all syntax parts of modules which are imported in that module are automatically forced to parse their text. All modifications which might result from these parses are immediately incorporated in the parse which initiated the process.

5.2.3 Miscellaneous features

This section contains the description of some features of the current system whose functionality and/or appearance will change drastically in the near future. They are, however, already very useful in their current form. The debugger provided to trace the evaluation of terms is described in Section 5.2.3.1, and the configuration file with which the system can be adapted to the user's wishes is described in Section 5.2.3.2.

5.2.3.1 Debugger

The current version of the ASF+SDF system provides a simple debugger (see [Tip91]) to trace the evaluation of terms. The current version of the debugger has the following two modes:

- *Step mode:*
The user communicates with the debugger just before an equation is applied to a term or before a condition of an equation has to be decided.
- *Go mode:*
The rewriting process is only interrupted at a breakpoint or it stops if the result of the evaluation is found.

As soon as the user gains control, the mode of debugging can be switched, breakpoints can be added or deleted, the amount of printed information can be changed, execution can be stopped, and the current term or condition can be inspected.

Debugging is activated by giving the LeLisp variable `#:EQM:debug` the value `true`. This is done by typing the command

```
(setq #:EQM:debug t)
```

in the LeLisp window (see Figure 5.1). Now, all reductions of terms to normal form will be done in debug mode. To leave debug mode, the value `false` has to be given to `#:EQM:debug` by typing

```
(setq #:EQM:debug ( ))
```

in the LeLisp window.

The debugger uses the LeLisp window to interact with the user. It starts in step mode. In this mode, the standard prompt of LeLisp (usually `?`) is changed to `<S:n/l>`, where *n* is the number of the reduction step and *l* is the level of the current condition. In go mode, the prompt is `<G:n/l>`, where *n* and *l* have the same meaning as in step mode. When inspecting the current term or condition, the prompt is `<TREE:n/l>`.

The commands which the debugger supports are listed in Figure 5.11.

| commands | | | |
|----------------|------|-------|--|
| step | s | | On typing step, s, or Return, the debugger does one step in step mode, or it reduces in go mode. |
| go [<nr>] | g | <nr>] | Reduce <nr> steps in go mode. If <nr> is omitted, the debugger reduces in go mode. |
| skip | x | | Reduce in go mode until the current level of conditions is finished. |
| goto [<nr>] | | | Goto step <nr> in go mode. If <nr> is smaller than current step number, the debugger will not return to that step. It reduces in go mode. |
| result [<nr>] | | | Reduce in go mode until the result of the current step (no <nr>) or step <nr> is found. |
| quit | q | | Reduction of the term to normal form is aborted. |
| help | h | ? | Online help is printed. |
| + <tagid> | | | A breakpoint is placed on all equations having the tag identifier <tagid> (the contents of the tag preceding each equation of the specification, see Section 5.3.2). |
| - <tagid> | | | All breakpoints of equations with tag identifier <tagid> are removed. |
| breaks | b | | All breakpoints are shown. |
| status | stat | | The stack of current conditions and their status is shown. |
| tree | t | | The current tree or condition can be inspected. |
| set +<options> | | | Add <options> to the current print options. |
| set -<options> | | | Remove <options> from the current print options. |
| set | | | The current print options are shown. |
| lisp | ll | | The user is asked to provide a LeLisp command which is then executed. |
| ! <command> | | | The Unix command <command> is executed. |

Fig. 5.11. Commands of the debugger

The commands which can be given when inspecting a term or condition (command tree) are listed in Figure 5.12.

| commands | | | |
|----------|---|---|--|
| left | l | | The previous child with the same father is printed. |
| right | r | | The next child with the same father is printed. |
| up | u | | The father is printed. |
| down | d | | The first child is printed. |
| root | | | The current term or condition is printed. |
| redex | | | The current redex is printed. |
| quit | q | | Inspecting the term or condition is ended and the debugger returns to the previous debugging mode. |
| help | h | ? | Online help is printed. |

Fig. 5.12. Commands of the debugger when inspecting terms or conditions

In Figure 5.13 the possible options in the commands `set +<options>` and `set -<options>` are listed.

| option | |
|--------|--|
| t | Print term. |
| e | Print equation. |
| b | Print variables in the equation and their binding. |
| r | Print result of the applied equation. |
| c | In step mode, the user is given control before deciding a condition. |
| i | Print conditions with instantiated variables. If this option is removed, conditions are printed as in the specification and the binding of variables is shown. |
| s | Print tag identifiers of equations when reducing in go mode. |
| z | Stop at breakpoints when reducing in go mode. |

Fig. 5.13. Options of the debugger when printing information

Several options can be added or deleted at once. The command `set +bte` adds the options `b`, `t`, and `e` at once to the printing options.

5.2.3.2 Configuration file

The configuration file gives the possibility to adapt the ASF+SDF system to the wishes of the user. The current version provides the following:

- *Search path:*

A search path can be defined in the configuration file. All files in the system are first searched in the directory provided by the user in the dialog boxes like the one in Figure 5.4. If a file cannot be found in that directory, it is successively looked for in the directories indicated in the search path. In this way it is possible to distribute a specification over several directories. Often used modules (like the specifications of the Booleans, natural numbers, sets, etc) can be placed in a library of modules located in a separate directory.

- *Initial modules:*

If a module should always be present in the system, this can be stated in the configuration file.

- *Buttons:*

The functionality of term editors can be changed by defining buttons which will be placed below the `reduce` button in the column left of the text (see Figure 5.6). The configuration file describes the name of the button, the term editors to which the button is attached, and the functionality of the button.

The configuration file is read in after starting an ASF+SDF system (when pushing the button `ASF+SDF` in the menu of Figure 5.2), or after clearing the system (choose `clear` in the `Specification` menu). At that moment, the file named `ctasdf.conf` from the current working directory of the LeLisp process is processed. Normally, this is the directory from which the initial command `ctasdf` was given.

An example of a configuration file in case of the typechecker for Mini-ML (See chapter 4) is the following:

```
(search-path "ML")
(read-always "Booleans" "Mini-ML-Expressions")
(term-button "check" "Mini-ML-Expressions" "Mini-ML-Typecheck"
  "check( <EXP> )" )
(lisp-button "print" "Booleans"
  (lambda (gse) (print "Tree " (#:GSE:tree gse))) )
```

The configuration file contains a list of LeLisp expressions. Each expression begins with a keyword indicating what is done with the arguments following the keyword.

The following four keywords are supported:

- **search-path:**
The arguments following this keyword are the directories added to the search path of the system. In the above example the directory named `ML` is added.
- **read-always:**
The strings after this keyword are names of modules which are loaded in the system. In the above example, the modules named `Booleans` and `Mini-ML-Expressions`, and all modules which these two import are automatically loaded into the system.
- **term-button:**
A **term-button** defines a button whose functionality is defined by a function in the specification. The respective arguments of **term-button** are its name (`check`), the module to whose term editors the button is attached (`Mini-ML-Expressions`), the name of the module used for evaluation (`Mini-ML-Typecheck`), and the text describing what has to be evaluated (`check(<EXP>)`). If this button is pushed, the text in the editor is parsed and if the result of parsing can be substituted for the meta-variable appearing in the given text, the entire text is evaluated using the equations of the named module. If, in the above example, the button named `check` is pushed and if the editor contains a syntactically correct `Mini-ML` expression, that expression is typechecked. Currently, the result of this evaluation is displayed in the `LeLisp` window (see Figure 5.1).
- **lisp-button:**
A **lisp-button** defines a button whose functionality is defined by a `Lisp` function. The arguments of a **lisp-button** are its name (`print` in the example), the name of the module to whose term editors it is attached (`Booleans`), and a `Lisp` function describing what should happen if it is pushed. The only argument of the latter function is the instance of `GSE` to which the button belongs. In the above example, the string `Tree` is printed followed by a print of the internal structure of the abstract syntax tree corresponding to the text in the editor. The function `#:GSE:tree` fetches that abstract syntax tree.

5.3 Syntax of ASF+SDF

This section contains the specification of the syntax of `ASF+SDF` written in `ASF+SDF` itself. This specification is created as a combination of the following components:

- The specification of the syntax of SDF written in SDF given in Appendix I of the SDF reference manual [HHKR89].
- The specification of the syntax of ASF written in a BNF-like notation in Section 1.2 of [BHK89a].
- The syntactic modifications which follow from the modifications described in Section 1.4.

In the following Section 5.3.1, the syntax of the syntax part of an ASF+SDF module is described, and in Section 5.3.2 the syntax of the equations part is described.

5.3.1 Syntax of the syntax part of an ASF+SDF module

All spaces, tabs (`\t`), and newlines (`\n`) are layout-characters. Comments are either defined as two percent signs (`%%`) followed by the rest of the line including the new-line character, or as a piece of text surrounded by percent signs (`%`). This is defined in the following module `Layout`.



Fig. 5.14. Structure diagram of `Layout`

```

module Layout
  exports
    lexical syntax
      [ \t\n]          -> LAYOUT
      "%%" ~[\n]* "\n" -> LAYOUT
      "%" ~[\n%]+ "%"  -> LAYOUT
  
```

Identifiers are used as names of modules, parameters, and sorts. They begin with a capital letter and consist of a non-empty sequence of letters, or digits, possibly with embedded hyphens (`-`) or underscores (`_`). Identifiers are defined in the following module `Identifiers`.

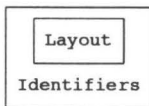


Fig. 5.15. Structure diagram of `Identifiers`

```

module Identifiers
  imports Layout
  
```

```

exports
  sorts Id
  lexical syntax
    [A-Z] -> Id
    [A-Z] [A-Za-z0-9\-\_]* [A-Za-z0-9] -> Id

```

Literals are used in the definitions of lexical functions (see module `LexFunctions`), context-free functions (`CfFunctions`), and variables (`Variables`). They normally consist of a possibly empty list of literal characters and are surrounded by double quotes ("). A *literal character* is any of the following:

- An arbitrary character with the exception of the non-printable characters whose three digit (octal) character code ranges from 000 through 037, the double quote ("), and the backslash (\).
- A backslash followed by an arbitrary character.
- A backslash followed by a three digit (octal) character code.

The double quotes surrounding a literal may be omitted if it begins with a lower case letter and consists of a non-empty sequence of letters, or digits, possibly with embedded hyphens or underscores. The double quotes are, however, obligatory if the literal coincides with any of the keywords of the formalism. These keywords are in alphabetic order:

| | | | |
|--------------|---------|------------|-----------|
| assoc | exports | module | right |
| bound | hiddens | non-assoc | sorts |
| bracket | imports | parameter | syntax |
| by | LAYOUT | priorities | to |
| CHAR | left | renamed | variables |
| context-free | lexical | | |

The following module `Literals` contains the definition of the syntax of literals.

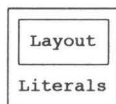


Fig. 5.16. Structure diagram of `Literals`

```

module Literals
  imports Layout
  exports
    sorts Literal Literal-Char
    lexical syntax

```

```

"\\"" Literal-Char* "\"      -> Literal
[a-z]      -> Literal
[a-z] [A-Za-z0-9\-\_]* [A-Za-z0-9] -> Literal
~[\000-\037"\\] -> Literal-Char
"\" ~[] -> Literal-Char
"\" [01] [0-7] [0-7] -> Literal-Char

```

The iterators + and * are respectively used to denote non-empty and possibly empty lists:

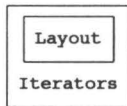


Fig. 5.17. Structure diagram of Iterators

```

module Iterators

imports Layout

exports
  sorts Iterator
  lexical syntax
  [+*] -> Iterator

```

The declaration of a *lexical function* (sort `LexFunction`) consists of one or more lexical elements followed by an arrow (`->`) and its result sort. Lexical functions whose result sort is `LAYOUT` define layout. *Lexical elements* (sort `LexElem`) are either basic lexical elements or basic lexical elements followed by an iterator. A *basic lexical element* (`BasicLexElem`) is either

- an identifier which denotes a non-terminal,
- a literal to denote a terminal,
- a character class, or
- a negation (`~`) of a character class which represents all characters except those listed in the character class.

The elements of the `lexical syntax` section are defined in the module `LexFunctions`.

```

module LexFunctions

imports Identifiers Literals Iterators

exports
  sorts
    CharClass CharRange CharRange-Char

```

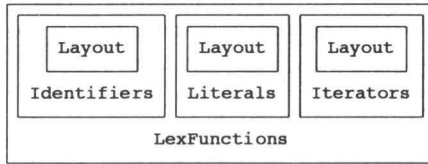


Fig. 5.18. Structure diagram of LexFunctions

```

LexFunction LexElem BasicLexElem
lexical syntax
"[" CharRange* "]" -> CharClass
CharRange-Char -> CharRange
CharRange-Char "-" CharRange-Char -> CharRange
~["\000-\037\-\[\]\\" -> CharRange-Char
"\" ~[" -> CharRange-Char
"\" [01] [0-7] [0-7] -> CharRange-Char
context-free syntax
LexElem+ "->" Id -> LexFunction
LexElem+ "->" "LAYOUT" -> LexFunction
BasicLexElem Iterator -> LexElem
BasicLexElem -> LexElem
Id -> BasicLexElem
Literal -> BasicLexElem
CharClass -> BasicLexElem
"~" CharClass -> BasicLexElem

```

The *context-free functions* (sort CfFunction) consist of an possibly empty list of context-free elements followed by an arrow (\rightarrow), a result sort, and optionally some attributes. The *context-free elements* (CfElem) are either

- an identifier (denoting a non-terminal of the grammar),
- a literal to denote the terminals of the grammar,
- an identifier followed by an iterator which denotes an iteration of the elements of the non-terminal, or
- an identifier and a literal surrounded by curly brackets and followed by an iterator denoting repetitions of the non-terminal, indicated by the identifier, separated by the terminal, given in the literal.

The abbreviations of context-free functions are defined as sort BareFunction. These are used in renamings (see module Imports) and priorities (module Priorities). They are either the context-free function without its attributes, or its terminal skeleton.

The attributes (*Attributes*) of a context-free function are either empty, a single attribute, or a list of one or more attributes separated by commas and surrounded by curly brackets (a set of attributes). The allowed attributes are *bracket*, *assoc*, *non-assoc*, *left*, and *right*.

The syntax of the elements of the context-free syntax section is specified in the following module *CfFunctions*.

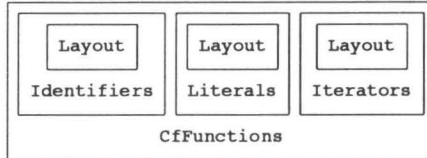


Fig. 5.19. Structure diagram of *CfFunctions*

```

module CfFunctions

imports Identifiers Literals Iterators

exports
  sorts
    CfFunction BareFunction
    CfElem Attributes Attribute
  context-free syntax
    CfElem* "->" Id Attributes -> CfFunction
    CfElem* "->" Id -> BareFunction
    Literal+ -> BareFunction

    Id -> CfElem
    Literal -> CfElem
    Id Iterator -> CfElem
    "{" Id Literal "}" Iterator -> CfElem

    % empty % -> Attributes
    Attribute -> Attributes
    "{" {Attribute ","}+ "}" -> Attributes

    "bracket" -> Attribute
    "assoc" -> Attribute
    "non-assoc" -> Attribute
    "left" -> Attribute
    "right" -> Attribute
  
```

The syntax of variables (module *Variables*) resembles the syntax of lexical functions (see the module *LexFunctions* above). The difference is that where a lexical function can only have an identifier as its result sort, for a variable more possibilities exist:

- an identifier,
- an identifier followed by an iterator,
- an identifier and literal surrounded by curly brackets, followed by an iterator,
- the keyword CHAR, and
- the keyword CHAR followed by an iterator.

This is defined in the following module `Variables`.

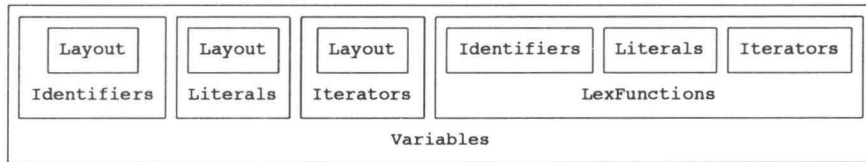


Fig. 5.20. Structure diagram of `Variables`

```

module Variables

imports Identifiers Literals Iterators LexFunctions

exports
  sorts Variable VarSort
  context-free syntax
    LexElem+ "->" VarSort -> Variable
    Id
    Id Iterator
    "{" Id Literal "}" Iterator
    "CHAR"
    "CHAR" Iterator
    Id -> VarSort
    Id Iterator -> VarSort
    "{" Id Literal "}" Iterator -> VarSort
    "CHAR" -> VarSort
    "CHAR" Iterator -> VarSort

```

Imports are defined in ASF+SDF as an identifier (representing the name of the module to be imported) followed by an optionally empty list of modifiers. *Modifiers* are either renamings or parameter bindings. A *renaming* consists of a list of one or more renaming sections which are labeled with a keyword (`sorts`, `lexical syntax`, `context-free syntax`, or `variables`) indicating the items to be renamed. There are two kinds of *parameter bindings*: those which contain a renaming to indicate which items in the parameter have to be renamed to match items in the actual module

```
Id "bound" "by" RenSection+ "to" Id -> Modifier,
```

and parameter bindings without renaming

```
Id "bound" "to" Id -> Modifier.
```

The two identifiers in parameter bindings respectively indicate the name of the

parameter which is bound, and the name of the actual module to which it is bound. The syntax of imports is defined as follows:

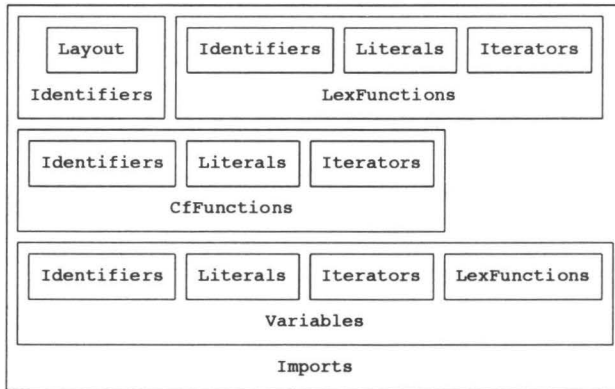


Fig. 5.21. Structure diagram of Imports

```

module Imports
imports Identifiers LexFunctions CfFunctions Variables
exports
  sorts
    Import Modifier RenSection
    SortRen LexFunctionRen CfFunctionRen VarRen
  context-free syntax
    Id Modifier* -> Import
    "renamed" "by" RenSection+          -> Modifier
    Id "bound" "by" RenSection+ "to" Id -> Modifier
    Id "bound" "to" Id                  -> Modifier
    "sorts" SortRen+                    -> RenSection
    "lexical" "syntax" LexFunctionRen+  -> RenSection
    "context-free" "syntax" CfFunctionRen+ -> RenSection
    "variables" VarRen+                  -> RenSection
    Id "=>" Id -> SortRen
    LexFunction "=>" LexFunction -> LexFunctionRen
    BareFunction "=>" CfFunction -> CfFunctionRen
    Variable "=>" Variable -> VarRen

```

In priorities sections the relative priority of context-free functions, and the associativity of groups of these functions is given.

The *relative priority* of two context-free functions is established by a declaration of the form $f > g$ (or alternatively: $g < f$), where f and g are either context-free

functions without their attributes or terminal skeletons. The following two types of abbreviations in declarations of relative priorities are allowed:

- $f_1 > f_2 > f_3$ stands for $f_1 > f_2$ and $f_2 > f_3$, and
- $\{f_1, f_2\} > f_3$ stands for $f_1 > f_3$ and $f_2 > f_3$.

The *associativity of groups* of context-free functions is given by a declaration of the set of context-free functions whose opening curly bracket is followed by an attribute (GroupAttribute) and a colon (:).

Summarizing, a priority declaration is either a declaration of the associativity of a group, or an ascending or descending list of elements of FunctionList. Each element of FunctionList is either a single function, a set of functions, or a set of functions containing an attribute. The following module Priorities gives the syntax of these elements of priorities sections.

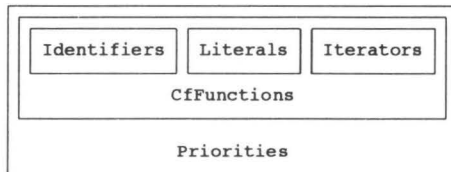


Fig. 5.22. Structure diagram of Priorities

```

module Priorities
imports CfFunctions
exports
  sorts PriorChain FunctionList GroupAttribute
  context-free syntax
    "{" GroupAttribute ":" {BareFunction ","}+ "}"
                                     -> PriorChain
    FunctionList ">" {FunctionList ">"}+ -> PriorChain
    FunctionList "<" {FunctionList "<"}+ -> PriorChain
    BareFunction
                                     -> FunctionList
    "{" {BareFunction ","}+ "}"
                                     -> FunctionList
    "{" GroupAttribute ":" {BareFunction ","}+ "}"
                                     -> FunctionList

    "left" -> GroupAttribute
    "right" -> GroupAttribute
    "non-assoc" -> GroupAttribute
  
```

Each module in an ASF+SDF specification is defined as the keyword module followed by an identifier (the name of the module), and a list of zero or more sections.

These sections are either imports, a single parameter, exports, hidden, or priorities. A parameter, the exports, and the hidden contain lists of sections in which respectively the sorts, lexical functions, context-free functions, and variables are specified. This is defined in the final module ASF-and-SDF:

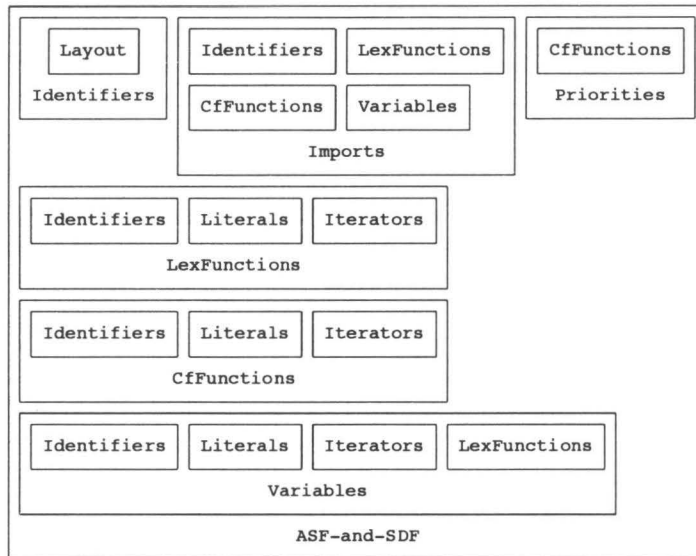


Fig. 5.23. Structure diagram of ASF-and-SDF

```

module ASF-and-SDF

imports
  Identifiers Imports Priorities
  LexFunctions CfFunctions Variables

exports
  sorts Module Section SDFsection
  context-free syntax
    "module" Id Section* -> Module
    "imports" Import+      -> Section
    "parameter" Id SDFsection* -> Section
    "exports" SDFsection+   -> Section
    "hidden" SDFsection+   -> Section
    "priorities" PriorChain -> Section
    "sorts" Id+             -> SDFsection
    "lexical" "syntax" LexFunction+ -> SDFsection
    "context-free" "syntax" CfFunction+ -> SDFsection
    "variables" Variable+   -> SDFsection
  
```

5.3.2 Syntax of the equations part of an ASF+SDF module

The equations part of an ASF+SDF module is either completely empty or it contains a non-empty list of (possibly conditional) equations (sort `CondEquation`) preceded by the keyword `equations`.

Each (possibly conditional) equation begins with a tag. These *tags* (sort `Tag`) are defined as tag identifiers surrounded by square brackets. Each *tag identifier* is a non-empty sequence of letters, digits, or quotes ('), possibly with embedded hyphens (-).

An *unconditional equation* is denoted by $s = t$ where s and t are terms belonging to the same user-defined sort. *Conditional equations* with at least one condition can be denoted in three different ways:

$$s = t \text{ when } Cond_1, Cond_2, \dots, Cond_n,$$

$$Cond_1, Cond_2, \dots, Cond_n \implies s = t, \text{ or}$$

$$\begin{array}{l} Cond_1, Cond_2, \dots, Cond_n \\ \hline s = t. \end{array}$$

Where s and t are again terms belonging to the same user-defined sort, and $Cond_1, Cond_2, \dots, Cond_n$ are conditions (elements of sort `Condition`). *Conditions* are either positive or negative which is denoted by, respectively, $s = t$ and $s \neq t$.

The grammar of the equations part of a module is based on the following definition:

```

module Equations
  exports
    sorts
      TagId Implies
      Equations CondEquation
      Tag Condition Equation NegEquation
    lexical syntax
      [A-Za-z0-9']                                -> TagId
      [A-Za-z0-9'] [A-Za-z0-9'\-]* [A-Za-z0-9'] -> TagId
      "==" "*" ">" -> Implies
      "==" "*"      -> Implies
    context-free syntax
      % empty %                                -> Equations
      equations CondEquation+ -> Equations
      Tag Equation                             -> CondEquation
      Tag {Condition "," }+ Implies Equation -> CondEquation

```

```

Tag Equation when {Condition ","}+    -> CondEquation
"[" TagId "]" -> Tag
Equation    -> Condition
NegEquation -> Condition

```

This specification is automatically extended as follows:

- All definitions of sorts, lexical functions, context-free functions, priorities, and variables of the *normalized* module (i.e., the module obtained by textually replacing imported modules by their text) under consideration are added. It is inadequate to add the name of the module to the `imports` section of the above module, because then the hiddens of the module would be lost.
- For each sort `SORT` in the normal form of the module the rules

```

SORT "=" SORT -> Equation, and
SORT "!=" SORT -> NegEquation

```

are added to the context-free syntax of the grammar.

- For each *chain rule*, i.e., a function from the context-free syntax section of the form

```

SORT1                      -> SORT2,
SORT1 Iterator             -> SORT2, or
{SORT1 Literal} Iterator -> SORT2

```

the following priority declarations are added:

```

SORT2 "=" SORT2 -> Equation
< SORT1 "=" SORT1 -> Equation, and
SORT2 "!=" SORT2 -> NegEquation
< SORT1 "!=" SORT1 -> NegEquation.

```

5.4 Internal structure

The ASF+SDF system contains the following components:

- a syntax manager SM which handles the syntactic parts of the specification,
- an equation manager EQM to handle the semantics defined in (possibly conditional) equations,
- a module manager MM which manages the modular aspects,
- a generic syntax-directed editor GSE of which several instances are used to edit the specification or its input, and

- a supervisor SV which drives the other components of the system and interprets the commands of the user.

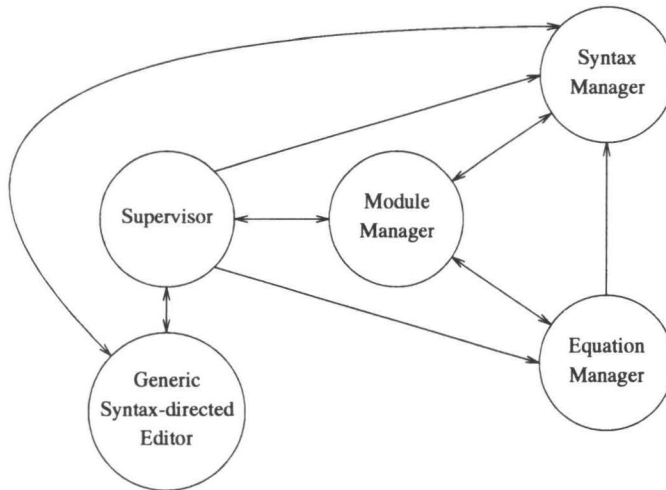


Fig. 5.24. Global architecture of the ASF+SDF system

The global architecture of the system is shown in Figure 5.24. The arrows in this figure indicate which components in the system use functions provided by other components.

The implementation of the ASF+SDF system is completely written in LeLisp [Lisp87]. Its architecture as described in this chapter reflects the current status of the implementation and was first described by Paul Klint in [Kli90]. The syntax manager SM [HKR90, Rek89b] was written by Jan Rekers, and the scanner generator [HKR87, Kli91] which is the part of it which handles lexical analysis was written by Paul Klint. The current version of the equation manager EQM was made by Casper Dik and Pum Walters. The debugger [Tip91] was written by Frank Tip. The author of this thesis wrote the module manager MM and the supervisor SV (see Chapter 7). The generic syntax-directed editor GSE [Log88, DK89, DK90] was the combined work of Monique Logger, Hans van Dijk, and Wilco Koorn. Furthermore, the system uses the VTP [CIL89] to manipulate abstract syntax trees, and the graphical objects [CI89b] for the man-machine interface. These were created at INRIA Sophia Antipolis and INRIA Rocquencourt by Dominique Clément, Janet Incerpi, Gilles Kahn, Bernard Lang, and co-workers.

In the following sections a short description of the functionality of each component and its interface with other components is given. The syntax manager SM is described in Section 5.4.1, the equation manager EQM in Section 5.4.2, the module

manager MM in Section 5.4.3, the generic syntax-directed editor GSE in Section 5.4.4, and the supervisor SV in Section 5.4.5.

5.4.1 Syntax manager - SM

The syntax manager SM is an enhanced version of the implementation of SDF. It consists of a lazy, incremental, and modular parser generator (MPG [Rek89b]) which can handle arbitrary context-free grammars. It generates a table-driven parser based on Tomita's algorithm [Tom85, Rek89a] which returns all possible parse trees of a given text.

In this component a lazy, incremental, and modular scanner generator (MSG [Kli91]) is used to generate a finite automaton from the lexical part of an SDF specification. This automaton is also capable of handling ambiguous regular expressions as it returns all possible interpretations of a given string.

Modularity has been implemented using selections (see Section 7.2.1). In all these components (including EQM discussed in the next section) code is generated lazily for the complete specification: a parser, a scanner, and a term rewriting machine are generated for the corresponding parts of the entire specification. If any of these components is needed, the appropriate part of the specification is selected and used in respectively parsing, scanning, and term rewriting.

In the system, SM is initiated by the module manager. All modifications in the syntax part of a module are translated into appropriate calls to functions of SM which add or delete

- a sort,
- a lexical function,
- a context-free function,
- a variable,
- a relative priority, or
- the associativity of a group.

For each module in the specification, two selections are created by MM. One needed to parse the equations part of a module, and the other one to parse terms of it. MM enables and disables (see Section 7.2.1) the elements of the syntax which constitute these selections. If GSE wants to parse text, it first asks MM to give the appropriate selection, and next SM is called to parse the text using that selection. GSE asks SM for information on the used syntax, for example, to fill the entries of the expand menu (see Section 5.2.2).

5.4.2 Equation manager - EQM

The equation manager EQM is an incremental and modular tool which interprets the equations of a specification as rewrite rules (conditions are interpreted as described in Section 2.4.2). Upon evaluation it uses leftmost innermost rewriting modulo lists (see Section 3.4) to rewrite a given term (the tree of the text in a term editor) to its normal form. Rewriting modulo associativity (see Section 3.3) is not yet implemented. As the evaluation strategy of EQM is similar to the strategy which is used by the ASF system, identical remarks on correctness and completeness of EQM can be made as in Section 2.5.

Currently, a simple prettyprinter is provided which prints the resulting normal form in the LeLisp window (see Figure 5.1 in Section 5.2.1). A simple debugger (see Section 5.2.3.1) to trace the reduction of a term to its normal form is provided.

As in case of SM, EQM is initiated by the module manager. EQM responds to additions and deletions of equations. For each module in the specification, MM asks EQM to create a selection. Next, equations are enabled or disabled whenever needed. To prettyprint the result of evaluation, EQM asks SM for information on the grammar.

5.4.3 Module manager - MM

The module manager MM is an incremental tool which manages the modular structure of a specification. The construction of this component originated from the idea that it would be useless if the information on the modular structure was spread over different components of the system.

In the ASF system (see Chapter 2) each module is normalized (its modular structure is removed as much as possible) and code is generated for it independently of the code generated for other modules. In MM the information of each module is kept in a database without constructing the normal form of each module explicitly. The information of the whole specification is given to the appropriate components like SM and EQM. Selections (two per module in SM, and one in EQM) are constructed which correspond to the information from the normal form of the modules.

The initialization of MM is done by the supervisor. All modifications on the specification are translated into additions or deletions of parts of it using the difference analysis algorithm as described in Section 7.3.3. MM provides functions to add or delete

- a module,
- a section (imports, parameter, exports, hidden, or priorities),
- an import,
- a syntax section (sorts, lexical syntax, context-free syntax, or variables),

- a sort,
- a lexical function,
- a context-free function,
- a variable,
- a relative priority,
- the associativity of a group, or
- an equation.

5.4.4 Generic syntax-directed editor - GSE

The generic syntax-directed editor GSE is a generic editing tool parameterized with a parser which in turn needs a specification of the syntax of a language. The user interface of GSE is described in Section 5.2.2 as part of the interface of the ASF+SDF system.

In general, the functionality of the editor can be extended in two different ways:

- addition of buttons, and
- reaction to changes made in the editor.

Upon initialization of an editor buttons can be added to it. If a user pushes such a button, the action attached to it is executed. In the system this functionality is used to provide for an evaluation button in term editors (see Section 5.2.2). If this button is pushed and the text in the editor is syntactically correct, the equations of the module are used to evaluate the text. The buttons described in the configuration file (see Section 5.2.3.2) are also added to term editors using this method.

After each syntactically correct modification of the text, the editor executes a function which can be used by the environment of the editor to cope with changes. This facility is used in the system to propagate modifications made in the syntax or the semantics of a module to MM, SM, and EQM (see also Section 7.3.3).

5.4.5 Supervisor - SV

The tasks of the supervisor SV are threefold:

- it interprets the commands of the user,
- it propagates changes made in editing (parts of) the specification to the appropriate components of the system, and
- it communicates errors found by the system to the user.

The main task of SV is to handle the communication with the user of the system (apart from the interaction via editing operations in GSE).

5.5 Assessment

As mentioned already in the introduction, the implementation of the system is not yet finished. In several areas further research has to be done to enhance the system and improve its implementation. A number of topics are discussed below, it is expected that these will be handled in the near future.

- *Renamings and parameter bindings:*

Renamings and parameter bindings are part of the formalism, but these features are not yet supported by the system.

- *Errors:*

If parse errors occur, the error window appears (see Figure 5.7). Static semantic errors like the use of a sort that is not declared, or cycles in the import graph, are printed in the window of the ASF+SDF system (see Figure 5.3). Messages of LeLisp like

```
** eval : ***** Erreur fatale : pile pleine. : ()
```

or messages which are printed whenever a bug in the system is found, are printed in the LeLisp window (see Figure 5.1). This non-uniform handling of errors is confusing, uniform handling is necessary.

- *Disambiguation tool:*

The current disambiguation tool (see Figure 5.8) gives no information on where an ambiguity was detected.

- *Hybrid implementations:*

Reduction of terms to normal form could be improved considerably if standard modules like Booleans, natural numbers, reals or sets did not use term rewriting but the corresponding data types of LeLisp in their implementation. In [Wal89, Wal90] a formal framework in which implementations based on term rewriting can be combined transparently with implementations based on conventional programming languages is given.

- *Incremental rewriting:*

The reduction of terms to normal form is not yet incremental. Whenever a term is reduced, EQM does not use information from previous reductions. In this context, the generation of incremental implementations from algebraic specifications is studied (see [Meu90]).

- *Prettyprinter:*

Prettyprinting (also called unparsing) consists of translating an abstract syntax tree into a corresponding text. This is needed when printing the result of evaluation or in debugging. The current prettyprinter does not always give a correct result as it does not insert brackets if needed. Prettyprinting results are printed

in the LeLisp window. It would be nice, however, if it could be handled by GSE.

- *Debugger:*

The interface of the current debugger (see Section 5.2.3.1) will be improved. In the future, it will also use GSE to show terms or conditions that are evaluated. Dialog boxes and other graphical objects will be used such that the user no longer needs to interact with the debugger via the LeLisp window.

- *Configuration file:*

The configuration file (see Section 5.2.3.2) is only the first attempt to provide a possibility to adapt the system to the user's wishes. It may be necessary to develop a formalism and a corresponding implementation which handles these configuration descriptions in a more user-friendly way.

Despite the system's deficits and the above-mentioned list of possible improvements, it has already been used to develop and test several specifications such as the ones mentioned in the introduction to this chapter. In particular, the interactive character of the ASF+SDF system and the user-defined syntax of the formalism are great improvements with respect to the ASF system (see Chapter 2).

Textual Modularization and its Semantic Consequences

For the sake of comprehensibility and reusability, large specifications should be split into several fragments. Hence, it is practical to split a specification into different modules each with its own name. If the text of a module is needed in another module, the name of the former is simply added to the imports of the latter. This simple modularization technique, which we call *textual modularization*, is discussed in this chapter. A formal definition and an algebraic specification of it are given. Its applicability is described by discussing the semantic consequences of adding textual modularization to non-modular specification formalisms.

6.1 Introduction

6.1.1 Modularization

In this chapter, textual modularization is studied independently of any particular formalism. This means that a specification can be split into one or more modules each with a name. If we want to (re)use a module in another one, we simply put the name of the former in the list of imports of the latter. The semantics of such an import is given by replacing the import in the importing module by the text of the imported module. The process by which the modular structure of a specification is eliminated is called *normalization*.

In contrast to studying modularization for a given specification formalism, our starting point is a given modularization technique. Its mathematical definition is given in Section 6.2 and the corresponding algebraic specification is given in Section 6.4. Both are independent of any particular specification formalism and can, henceforth, be used to describe a family of modular specification formalisms sharing the same modular constructs. In Section 6.3 the semantic consequences of adding textual modularization to a given non-modular specification formalism are studied.

We may think of the following examples as applications of textual modularization:

- *Modular grammars:*

Each module consists of a set of start symbols, terminals, and non-terminals, and a set of syntax rules. Examples are regular grammars defining the lexical syntax of a language, or context-free grammars.

- *Modular logics:*

Each module consists of a set of declarations of non-logical symbols (such as constants, and relation and function symbols), and a set of axioms. For many-sorted logics the module also contains a set of sorts, and the non-logical symbols will be typed. The allowed formulae are derived from the non-logical symbols in combination with the logical connectives (like \wedge , \vee , and \exists). The set of axioms is a subset of this set of allowed formulae. Some examples are:

- *Algebraic specification formalisms (modular, many-sorted, equational logic):* Each module consists of a set of sorts and functions (constituting a signature), and a set of (possibly conditional) equations. These formalisms are used to define abstract data types.
- *Modular, many-sorted, first-order (predicate or propositional) logic:* Each module consists of a set of sorts, constants, relation and function symbols, and a set of axioms.

In these examples each module contains a *set* of elements such as syntax rules, sorts, functions, and axioms. In the sequel, we call such elements *items*.

Why are only specification formalisms all of whose specifications consist of an (unordered) *set* of items discussed? When adding modularization to such formalisms, the imports of a module can be described as a *set* of module names. The order in which the module names occur and the possibility that a module name occurs more than once are irrelevant after textual expansion of a module. Conversely, if imports are described using *sets* of module names, each specification has to consist of a set of items. If we considered *lists* of items instead of sets, the order in which a module is normalized would be important. When considering *multisets* of items care must be taken when handling modules that are imported in another module via different routes.

6.1.2 Related work

The model of modularization described above is used in both ASF [BHK89a] and the combination of ASF and SDF [HK89b, HHKR89] (see also Sections 1.4 and 5.3). Here the analogue of the imports-mechanism of ASF is discussed. The other module operations present in these formalisms like renaming, export, and actualization of parameters could be handled in a similar way.

The last mentioned module operators (except actualization) are also studied in [BHK90], where an algebraic definition of them is given. This definition, which is

called *BMA* (for basic module algebra) is tailored towards formal systems involving terms defined by many-sorted signatures, and in particular towards many-sorted first-order logic with equality. My approach is more general in that it does not make these assumptions. Another difference between both approaches is the use of *names* of modules. In [BHK90] modules do not have names. A modular specification is a module expression whose modular structure is indicated by the occurrence of export operators rather than imports. It is shown that, by applying the axioms of *BMA*, each such specification can be brought into a normal form containing at most one export operator. In my approach, modules do have names and, as I want to pay attention to the typechecking of modular specification formalisms, *explicit* functions which return the normal form of a module in the context of a specification are defined.

In [BHK90] four different semantics of *BMA* are discussed and compared. Three of these, in which a module is interpreted as, respectively, the class of all its models, the class of all its countable models, and its *theory* (the set of formulae derivable from the given axioms), are also proved to be models for textual modularization (see Theorems 6.2 and 6.4). The other model is the initial algebra of *BMA* itself. In Section 6.3 I have tried to establish the circumstances in which the semantics of a non-modular specification formalism is suitable for textual modularization.

The above-mentioned module operators are already known from other specification formalisms. In Clear [BG80] the semantics of a specification is its theory. The module operators of Clear correspond to semantic operators on theories. In OBJ [GMP83] and its successors OBJ2 [FGJM85] and OBJ3 [GKKMMW88, KKM88] a specification has a formal semantics based on the initial algebra semantics, and an operational semantics based on rewrite rules. Their modular operations are also based mainly on textual expansion.

In [GV89] the use of modularization for describing process algebra is proposed. As semantics of modules the theory and the class of all models is studied. The following module operators occur naturally in this context: the union operator $+$ to combine two modules, the export operator \square which allows one to forget some operators in a module, the operator H which changes the semantics by taking an homomorphic image, and the operator S which constructs a subalgebra. These operations on the semantic domain of the formalism are then transformed into module operators in the formalism.

The title of [DC90] suggests that the authors discuss how to combine attribute grammars. They, however, define modules in which a set of patterns and associated templates is specified. These patterns can be applied to a context-free grammar, and for those that match an attribute computation is generated from the associated template. The union of all sets of patterns with their associated template is applied to a

context-free grammar. Unfortunately, the semantics of composite modules in terms of the semantics of the components is not discussed.

6.2 Definitions

6.2.1 Non-modular specification formalisms

As described in the introduction to this chapter, specification formalisms are only considered such that specifications written in it are sets of items. Such formalisms are characterized primarily by the set of items allowed in specifications. Each specification is a finite subset of this set of allowed items. The definition of a specification formalism is as follows:

Definition 6.1: A *non-modular specification formalism* $\langle Items, ScSpec, SD, int \rangle$ is a structure consisting of:

- The set of allowed items $Items$. Each specification in the formalism is a finite subset of $Items$. In other words: it is an element of the finite power set of $Items$, which we call $Spec \equiv FP(Items)$.
- A *static correctness property* $ScSpec \subset Spec$, which determines which specifications in the formalism are statically correct. In most practical examples this property will be decidable and its implementation yields the typechecker of the formalism.
- A *semantic domain* SD .
- A (possibly partial) function $int: Spec \rightarrow SD$, which defines the semantics of the formalism (its interpretation).

If we are not particularly interested in the semantics of the formalism, a non-modular specification formalism is denoted by $\langle Items, ScSpec \rangle$. If we are not even interested in the static correctness property, it is simply denoted by $Items$.

6.2.2 Modular specification formalisms

In this section formal definitions of the simplest form of textual modularization are given. For each non-modular specification formalism as described in Definition 6.1, a *module* is defined as a specification labeled with a name, and augmented with *imports*. Imports are just sets of module names. The modules whose names are an element of the imports of a module are to be incorporated in it. All items are *exported* from a module, i.e., all of them are available in modules in which the module is imported (either directly or indirectly). The meaning of a module can only be given after *normalization* of it in the context of a given specification, which

is just a finite set of modules. Normalization is the process of textual expansion of a module.

Definition 6.2: Let $\langle Items, ScSpec, SD, int \rangle$ be a given non-modular specification formalism, then we define:

- A *module* $\mathcal{M} \equiv \langle Name(\mathcal{M}), Imp(\mathcal{M}), Items(\mathcal{M}) \rangle$ is a structure containing:
 - a module name $Name(\mathcal{M})$,
 - a set of module names $Imp(\mathcal{M})$ which the module imports, and
 - a finite set of items $Items(\mathcal{M}) \in Spec$, which is a (possibly statically incorrect) specification of the non-modular formalism.
- A *modular specification* S consists of a *finite* set of modules.
- The *import graph* of a given modular specification S is a binary relation imp_S defined on S by

$$\mathcal{M}_1 imp_S \mathcal{M}_2 \Leftrightarrow Name(\mathcal{M}_2) \in Imp(\mathcal{M}_1).$$

The transitive closure of the import graph imp_S of S is denoted as imp_S^+ , and the reflexive and transitive closure as imp_S^* .

- Let S be a given modular specification, then we define the *normal form* of a module $\mathcal{M} \in S$ in the context of S as the specification $NF(\mathcal{M}, S) \in Spec$ which is the smallest set of items such that:

$$NF(\mathcal{M}, S) \equiv Items(\mathcal{M}) \cup \{i \in NF(\mathcal{N}, S) \mid \mathcal{M} imp_S \mathcal{N}\}.$$

It is essential to define the normal form as the *smallest* set because the smallest fixed point of the above definition is needed in case the module \mathcal{M} is either directly or indirectly imported in itself.

- The *semantics* of a module \mathcal{M} in the context of a modular specification S is defined as the semantics of its normal form $int(NF(\mathcal{M}, S))$.

In the proof of the following Theorem 6.1 an alternative definition of the normal form of a module is used. This definition and the proof of its equivalence to the above definition are stated in the following lemma.

Lemma 6.1: Let S be a given modular specification, then we have for all $\mathcal{M} \in S$ that

$$NF(\mathcal{M}, S) = \{i \in Items(\mathcal{N}) \mid \mathcal{M} imp_S^* \mathcal{N}\} = \bigcup_{\{\mathcal{N} \in S \mid \mathcal{M} imp_S^* \mathcal{N}\}} Items(\mathcal{N}).$$

Proof: By induction on the complexity of both definitions of normal forms. □

The following theorem shows that the above definitions of normal forms reflect the intuition of textual modularization. It states that

- A name can be removed from a list of imports if the specification does not contain modules with that name.
- A name can also be removed from the imports if it is identical to the name of the module itself and if no other module with the same name exists.
- The constituents of a module whose name occurs in the imports of another module can be added to that importing module.

Theorem 6.1: Let S be a modular specification and let $\mathcal{M} = \langle \text{Name}(\mathcal{M}), \text{Imp}(\mathcal{M}), \text{Items}(\mathcal{M}) \rangle$ be a module in S .

1. If for some name $N \in \text{Imp}(\mathcal{M})$ there is no module with name N in S , then $NF(\mathcal{M}, S) = NF(\mathcal{M}', S')$ with

$$\mathcal{M}' \equiv \langle \text{Name}(\mathcal{M}), \text{Imp}(\mathcal{M}) - \{N\}, \text{Items}(\mathcal{M}) \rangle \text{ and}$$

$$S' \equiv (S - \{\mathcal{M}\}) \cup \{\mathcal{M}'\}.$$

2. If $\mathcal{M} \text{ imp}_S \mathcal{M}$ and if there is no other module in S with name $\text{Name}(\mathcal{M})$, then $NF(\mathcal{M}, S) = NF(\mathcal{M}', S')$ with

$$\mathcal{M}' \equiv \langle \text{Name}(\mathcal{M}), \text{Imp}(\mathcal{M}) - \{\text{Name}(\mathcal{M})\}, \text{Items}(\mathcal{M}) \rangle \text{ and}$$

$$S' \equiv (S - \{\mathcal{M}\}) \cup \{\mathcal{M}'\}.$$

3. If $\mathcal{M} \text{ imp}_S \mathcal{N}$ with $\mathcal{N} \equiv \langle \text{Name}(\mathcal{N}), \text{Imp}(\mathcal{N}), \text{Items}(\mathcal{N}) \rangle$, then $NF(\mathcal{M}, S) = NF(\mathcal{M}', S')$ with

$$\mathcal{M}' \equiv \langle \text{Name}(\mathcal{M}), \text{Imp}(\mathcal{M}) \cup \text{Imp}(\mathcal{N}), \text{Items}(\mathcal{M}) \cup \text{Items}(\mathcal{N}) \rangle \text{ and}$$

$$S' \equiv (S - \{\mathcal{M}, \mathcal{N}\}) \cup \{\mathcal{M}'\}.$$

Proof: From the above Lemma 6.1 it follows immediately, that in cases (1) and (2), it suffices to show that the modules in the import graph of S which are imported in \mathcal{M} and \mathcal{M}' are equivalent in the sense that \mathcal{M} is replaced by \mathcal{M}' . For, we then have

$$NF(\mathcal{M}, S) = \text{Items}(\mathcal{M}) \cup \bigcup_{\{\mathcal{N} \in S \mid \mathcal{M} \text{ imp}_S^* \mathcal{N}\} - \{\mathcal{M}\}} \text{Items}(\mathcal{N}) =$$

$$\text{Items}(\mathcal{M}') \cup \bigcup_{\{\mathcal{N} \in S' \mid \mathcal{M}' \text{ imp}_{S'}^* \mathcal{N}\} - \{\mathcal{M}'\}} \text{Items}(\mathcal{N}) = NF(\mathcal{M}', S').$$

The equality

$$\{\mathcal{N} \in S \mid \mathcal{M} \text{ imp}_S^* \mathcal{N}\} - \{\mathcal{M}\} = \{\mathcal{N} \in S' \mid \mathcal{M}' \text{ imp}_{S'}^* \mathcal{N}\} - \{\mathcal{M}'\}$$

follows by transforming a chain of importing modules from one specification into a corresponding chain in the other specification. In case (1) this is done by

replacing each occurrence of \mathcal{M} in a chain from \mathcal{S} by an occurrence of \mathcal{M}' and vice versa. In case (2) we replace a list of consecutive occurrences of \mathcal{M} by one occurrence of \mathcal{M}' in the transformation from \mathcal{S} to \mathcal{S}' , and in the converse each occurrence of \mathcal{M}' is replaced by \mathcal{M} . Note that in both cases these transformations are not each others inverse.

The proof of case (3) is similar. If \mathcal{M} is followed by \mathcal{N} in a chain of importing modules from \mathcal{S} , we replace it by \mathcal{M}' to obtain a chain of importing modules from \mathcal{S}' . An occurrence of \mathcal{M}' in a chain of importing modules from \mathcal{S}' is replaced by \mathcal{M} followed by \mathcal{N} whenever the name of the module following \mathcal{M}' is an element of $\text{Imp}(\mathcal{N})$. In all other cases \mathcal{M}' is simply replaced by \mathcal{M} . Hence,

$$\{\mathcal{N}' \in \mathcal{S} \mid \mathcal{M} \text{ imp}_{\mathcal{S}}^* \mathcal{N}'\} - \{\mathcal{M}, \mathcal{N}\} = \{\mathcal{N}' \in \mathcal{S}' \mid \mathcal{M}' \text{ imp}_{\mathcal{S}'}^* \mathcal{N}'\} - \{\mathcal{M}'\}.$$

As $\text{Items}(\mathcal{M}') = \text{Items}(\mathcal{M}) \cup \text{Items}(\mathcal{N})$, this completes the proof. \square

The above theorem shows that normalization of a module \mathcal{M} in the context of a given modular specification \mathcal{S} can be done by stepwise replacement of the imports of \mathcal{M} . In cases (1) and (2) the number of imports of \mathcal{M} decreases. In case (3), the number of modules in the transitive closure of the import graph of \mathcal{M} (i.e., $\{\mathcal{N} \in \mathcal{S} \mid \mathcal{M} \text{ imp}_{\mathcal{S}}^* \mathcal{N}\}$) usually decreases. Only if \mathcal{M} imports itself directly, and if we choose $\mathcal{N} = \mathcal{M}$, it does not decrease.

In Definition 6.2 the semantics of a module in the context of a modular specification is defined without imposing any constraints on the specification in question. This shows that such constraints are superfluous from a theoretical point of view. In practice, however, the user of a modular specification formalism would like to be warned whenever, for example, an import contains a name for which no module exists in the specification. In the following definition the most plausible static constraints for modular specifications are given.

One of these static constraints on modular specifications is the *origin rule*, which was introduced in ASF [BHK89b]. It forbids identification of identical items from different modules if they are imported in one module. Violations of the origin rule can be eliminated by creating a new module containing the item(s) that caused the conflict. This new module should be imported into the modules in which the item(s) originally occurred.

Definition 6.3: Let $\langle \text{Items}, \text{ScSpec} \rangle$ be a given specification formalism, then a modular specification \mathcal{S} is defined to be *statically correct* if it meets the following requirements:

- The names of all modules in the specification should be unique, i.e.,

$$\text{Name}(\mathcal{M}_1) = \text{Name}(\mathcal{M}_2) \Rightarrow \mathcal{M}_1 = \mathcal{M}_2.$$

- All module names occurring in imports should be present in the specification, i.e.,

$$\text{Imp}(\mathcal{M}) \subset \{\text{Name}(\mathcal{N}) \mid \mathcal{N} \in S\}.$$

- Cycles in the import graph are forbidden. In other words: the transitive closure of the import relation should be irreflexive, i.e.,

$$\neg \mathcal{M} \text{ imp}_S^+ \mathcal{M}.$$

- The *origin rule* should hold for all items of the specification, i.e.,

$$\left[i \in \text{Items}(\mathcal{M}_1) \wedge i \in \text{Items}(\mathcal{M}_2) \wedge \mathcal{M} \text{ imp}_S^* \mathcal{M}_1 \wedge \mathcal{M} \text{ imp}_S^* \mathcal{M}_2 \right] \Rightarrow \mathcal{M}_1 = \mathcal{M}_2.$$

- The normal forms of all modules in the specification should be statically correct, i.e.,

$$\text{NF}(\mathcal{M}, S) \in \text{ScSpec}.$$

Remark that the operations on modular specifications as described in Theorem 6.1 do not preserve static correctness. In fact, each statically incorrect modular specifications can be transformed in a statically correct one using these operations.

6.3 Semantic consequences

Some possible semantics belonging to the examples of specification formalisms given in the introduction (Section 6.1) are:

- *Grammars:*

The standard semantics of a grammar is the language (the set of all strings) produced by it.

- *Algebraic specifications:*

Some of the well-known semantics are:

Theory semantics: This semantics is the theory of an algebraic specification. It is the set of all equations without conditions that can be proved from the given equations using (conditional) equational logic (see Section 3.2.1). It is possible to consider either the closed theory (the set of derivable equations without variables) or the open theory (the set of derivable equations with or

without variables). It is also interesting to consider the conditional equational theory, i.e., the set of all non-conditional as well as conditional equations derivable from the specification.

(Countable) model class semantics: A second possibility is to consider the set of algebras in which the given (conditional) equations are true as the semantics of an algebraic specification. Likewise, one can choose the set of countable algebras that satisfy the given equations as semantics.

Initial/final algebra semantics: Furthermore, one can take the initial algebra or one of the final algebras (if one exists) as semantics of an algebraic specification.

- *First order (propositional or predicate) logic:*

We can choose semantics equivalent to those of algebraic specifications:

Theory semantics: In this case the theory semantics is the set of all (either closed or open) formulae derivable from the given axioms using first-order proposition or predicate calculus.

(Countable) model class semantics: Either the set of all models or the set of all countable models can be chosen as semantics.

Least Herbrand model: Finally, specific models like the least Herbrand model (if it exists and is unique) of a first-order theory can be chosen.

Which of the above-mentioned semantics is appropriate for textual modularization as defined in the previous section? As described in Section 6.2.1, in general it is possible to define the semantics of a formalism as a function *int* which assigns to each specification an interpretation in some semantic domain \mathcal{SD} . As the normal form of a compound module is defined in terms of the union of the component parts (see Definition 6.2), it would be meaningful to require that an operator \oplus exists such that for all specifications $I_1, I_2 \in \text{Spec}$

$$\text{int}(I_1 \cup I_2) = \text{int}(I_1) \oplus \text{int}(I_2) \quad (\oplus)$$

up to isomorphism. In other words: an operator \oplus should exist such that the diagram in Figure 6.1 commutes.

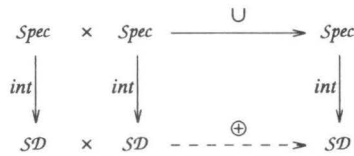


Fig. 6.1. Commutative diagram for semantics

A semantics of a specification formalism for which the diagram commutes is called *compositional*.

Hence, the question can be rephrased as: Which of the above-mentioned semantics are compositional? In the next section examples of non-compositional semantics are given. Then, general definitions of the theory semantics and the model class semantics of a specification formalism are given in respectively Sections 6.3.2 and 6.3.3. Both kind of semantics will be proved compositional under certain conditions.

6.3.1 Non-compositional semantics

If a semantics of a specification formalism is compositional, it follows immediately from (\oplus) that for all specifications $I_1, I_2, I \in \text{Spec}$

$$\text{int}(I_1) = \text{int}(I_2) \Rightarrow \text{int}(I_1 \cup I) = \text{int}(I_2 \cup I).$$

Consequently, we can prove that a semantics is not compositional by showing the existence of three specifications $I_1, I_2, I \in \text{Spec}$ such that $\text{int}(I_1) = \text{int}(I_2)$ and $\text{int}(I_1 \cup I) \neq \text{int}(I_2 \cup I)$. In other words: a semantics is non-compositional if by adding an item to two different specifications with identical semantics the latter can become different.

If we choose the language (set of all strings) produced by a grammar as the semantics of a formalism in which grammars can be specified, then it is not compositional. Consider the grammars

$$\begin{aligned} I_1 &\equiv \{S ::= A \mid B \mid C, A ::= "1", B ::= "2"\}, \\ I_2 &\equiv \{S ::= A \mid B \mid C, A ::= "2", B ::= "1"\}, \text{ and} \\ I &\equiv \{C ::= A B\}. \end{aligned}$$

Now, we have that $\text{int}(I_1) = \text{int}(I_2) = \{"1", "2"\}$ whereas $\text{int}(I_1 \cup I)$ equals $\{"1", "2", "12"\}$ and $\text{int}(I_2 \cup I)$ is $\{"1", "2", "21"\}$. In the following section a semantics for grammars in the style of the theory semantics is given and its compositionality is proved.

The initial algebra semantics of an algebraic specification is not compositional either. Examine the specifications

$$\begin{aligned} I_1 &\equiv \{a : \rightarrow S, f : S \rightarrow S, x \in \text{Var}(S), f(x) = x\}, \\ I_2 &\equiv \{a : \rightarrow S, f : S \rightarrow S, f(a) = a\}, \text{ and} \\ I &\equiv \{b : \rightarrow S\}. \end{aligned}$$

The initial algebras of both I_1 and I_2 have a single carrier S with one element a . The function f is the identity function on S . Hence, these algebras are isomorphic.

The initial algebra of $I_1 \cup I$ has a single carrier with two elements whereas the carrier of the initial algebra of $I_2 \cup I$ has an infinite (but countable) set of elements $a, b, f(b), f(f(b)), \dots$. Obviously, the latter two algebras are not isomorphic.

As initial algebra semantics is closely related to closed theory semantics, it is not surprising that the same specifications can be used to show that the closed theory of an algebraic specification does not lead to a compositional semantics. The closed theory is the set of non-conditional equations without variables that can be proved from the equations in the specification using (conditional) equational logic (see also Section 3.2.1). If we abbreviate $f(f(\dots f(a)\dots))$ with m repetitions of f to $f^m(a)$, it gives

$$\text{int}(I_1) = \text{int}(I_2) = \{f^m(a) = f^n(a) \mid m, n \geq 0\},$$

whereas

$$\text{int}(I_1 \cup I) = \{f^m(a) = f^n(a) \mid m, n \geq 0\} \cup \{f^m(b) = f^n(b) \mid m, n \geq 0\}$$

and

$$\text{int}(I_2 \cup I) = \{f^m(a) = f^n(a) \mid m, n \geq 0\} \cup \{f^m(b) = f^n(b) \mid m \geq 0\}.$$

Even the usual (open) theory semantics for conditional equational logic is not compositional if it is defined as the set of *non-conditional* equations that can be derived from the given (possibly conditional) equations in the specification. Consider the specifications

$$I_1 \equiv \{a, b, c, d : \rightarrow S\},$$

$$I_2 \equiv \{a, b, c, d : \rightarrow S, a = b \text{ when } c = d\}, \text{ and}$$

$$I \equiv \{a, b, c, d : \rightarrow S, c = d\}.$$

The set of non-conditional equations which can be derived from I_1 as well as I_2 is the set of all *tautologies*. This is the set of equations derivable from the empty set of axioms, i.e., the set of equations $a = a, b = b, c = c, d = d$, and $x = x$ for all possible variables x . In $\text{int}(I_1 \cup I)$ the equations $c = d$ and $d = c$ are added to the set of tautologies whereas in $\text{int}(I_2 \cup I)$ the equations $a = b, b = a, c = d$, and $d = c$ are added.

In the next section, the full open theory semantics (i.e., the set of derivable conditional as well as unconditional equations with and without variables) is shown to be compositional.

6.3.2 Theory semantics

In this section, the compositionality of theory semantics is handled in two stages. First, the simple case in which the axioms of the theory are exactly the formulae

given in the specification are handled. Next, a more general case in which the specification contains declarations of items which indicate which formulae are axioms is treated.

A general definition of the simplest form of theory semantics is the following:

Definition 6.4:

- A *theory semantics* $\langle \mathcal{Form}, \vdash \rangle$ for a given specification formalism *Items* is characterized by:
 - a given set of *formulae* \mathcal{Form} which is a subset of *Items*, and
 - a *proof relation* \vdash which is a relation on $P(\mathcal{Form}) \times \mathcal{Form}$ stating which formulae are derivable from a given set of formulae.

The latter is denoted by $\Gamma \vdash \phi$ for all formulae $\phi \in \mathcal{Form}$ and sets of formulae $\Gamma \subset \mathcal{Form}$.

- A proof relation \vdash for a given set of formulae \mathcal{Form} is called *correct* if it satisfies the following three constraints for all formulae $\phi \in \mathcal{Form}$ and for all sets of formulae $\Gamma, \Gamma_1, \Gamma_2 \subset \mathcal{Form}$:

$$\phi \in \Gamma \Rightarrow \Gamma \vdash \phi, \quad (|-1)$$

$$\Gamma_1 \vdash \phi \Rightarrow \Gamma_1 \cup \Gamma_2 \vdash \phi, \text{ and} \quad (|-2)$$

$$\Gamma_1 \vdash \Gamma_2 \wedge \Gamma_2 \vdash \phi \Rightarrow \Gamma_1 \vdash \phi, \quad (|-3)$$

where $\Gamma_1 \vdash \Gamma_2$ means $\Gamma_1 \vdash \psi$ for all $\psi \in \Gamma_2$.

- The semantic domain \mathcal{SD} of a theory semantics $\langle \mathcal{Form}, \vdash \rangle$ is the set of all theories $\mathcal{Th}(\mathcal{Form}, \vdash)$. A *theory* $\mathcal{Th} \subset \mathcal{Form}$ is a set of formulae which is closed under provability of a given proof relation, i.e.:

$$\mathcal{Th} \vdash \phi \Leftrightarrow \phi \in \mathcal{Th}$$

for all formulae $\phi \in \mathcal{Form}$.

- The interpretation $int: \mathcal{Spec} \rightarrow \mathcal{Th}(\mathcal{Form}, \vdash)$ corresponding to a theory semantics is defined by

$$int(I) \equiv \{\phi \in \mathcal{Form} \mid I \cap \mathcal{Form} \vdash \phi\}.$$

Note that the interpretation is defined correctly if the proof relation of the theory semantics is correct. In that case, $int(I)$ is indeed a theory for all specifications $I \in \mathcal{Spec}$. For if $int(I) \vdash \phi$, we can use the transitivity of the proof relation (|-3) and the fact that $I \cap \mathcal{Form} \vdash int(I)$ to conclude that $I \cap \mathcal{Form} \vdash \phi$ or in other words $\phi \in int(I)$. Conversely, if $\phi \in int(I)$ then property (|-1) immediately gives that $int(I) \vdash \phi$.

In the proofs of the following theorems we will use the following properties of correct proof relations:

Lemma 6.2: Let \vdash be a correct proof relation for a given set of formulae \mathcal{Form} , then the following properties hold for all sets of formulae $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \subset \mathcal{Form}$:

$$\Gamma_1 \subset \Gamma_2 \Rightarrow \Gamma_2 \vdash \Gamma_1 \quad (a)$$

$$\Gamma_1 \vdash \Gamma_2 \wedge \Gamma_3 \vdash \Gamma_4 \Rightarrow \Gamma_1 \cup \Gamma_3 \vdash \Gamma_2 \cup \Gamma_4 \quad (b)$$

Proof: Property (a) follows immediately from property (\vdash 1) of the proof relation.

If $\Gamma_1 \vdash \Gamma_2$, then by property (\vdash 2) we have $\Gamma_1 \cup \Gamma_3 \vdash \Gamma_2$. Likewise, we can prove $\Gamma_1 \cup \Gamma_3 \vdash \Gamma_4$ which terminates the proof of property (b). \square

We are now ready to prove that a theory semantics of a given specification formalism is compositional.

Theorem 6.2: Let $\langle \mathcal{Form}, \vdash \rangle$ be a theory semantics for a given specification formalism \mathcal{Items} , then it is compositional if its proof relation is correct. In these circumstances the operator \oplus which completes the commutative diagram of Figure 6.1 is defined by:

$$\mathcal{Th}_1 \oplus \mathcal{Th}_2 \equiv \{\phi \in \mathcal{Form} \mid \mathcal{Th}_1 \cup \mathcal{Th}_2 \vdash \phi\}$$

for all theories $\mathcal{Th}_1, \mathcal{Th}_2 \in \mathcal{Th}(\mathcal{Form}, \vdash)$.

Proof: Before we can start the actual proof of the compositionality of theory semantics, we first have to show that the operator \oplus is well defined, i.e., we have to show that $\mathcal{Th}_1 \oplus \mathcal{Th}_2$ is a theory for all theories $\mathcal{Th}_1, \mathcal{Th}_2 \in \mathcal{Th}(\mathcal{Form}, \vdash)$. This proof is similar to the above proof of the fact that the interpretation for theory semantics is well defined.

To prove the compositionality, we have to show that for all $I_1, I_2 \in \mathcal{Spec}$ condition (\oplus) holds, i.e.,

$$\{\phi \in \mathcal{Form} \mid \text{int}(I_1) \cup \text{int}(I_2) \vdash \phi\} = \{\phi \in \mathcal{Form} \mid (I_1 \cup I_2) \cap \mathcal{Form} \vdash \phi\}.$$

The definition of int gives $I \cap \mathcal{Form} \vdash \text{int}(I)$ for an arbitrary specification $I \in \mathcal{Spec}$. Consequently, we have for $i = 1, 2$ $I_i \cap \mathcal{Form} \vdash \text{int}(I_i)$. Property (b) of Lemma 6.2 gives

$$(I_1 \cap \mathcal{Form}) \cup (I_2 \cap \mathcal{Form}) \vdash \text{int}(I_1) \cup \text{int}(I_2)$$

which when combined with the transitivity of the proof relation (\vdash 3) results in

$$\begin{aligned}
& \{\phi \in \mathcal{Form} \mid \text{int}(I_1) \cup \text{int}(I_2) \vdash \phi\} \subset \\
& \quad \{\phi \in \mathcal{Form} \mid (I_1 \cap \mathcal{Form}) \cup (I_2 \cap \mathcal{Form}) \vdash \phi\} = \\
& \quad \{\phi \in \mathcal{Form} \mid (I_1 \cup I_2) \cap \mathcal{Form} \vdash \phi\}.
\end{aligned}$$

Conversely, we start by showing $\text{int}(I) \vdash I \cap \mathcal{Form}$ for all specifications $I \in \text{Spec}$. This is easy as $I \cap \mathcal{Form} \subset \text{int}(I)$ (use property ($\vdash 1$)). Consequently, we have now proved for $i = 1, 2$ $\text{int}(I_i) \vdash I_i \cap \mathcal{Form}$ which results in

$$\text{int}(I_1) \cup \text{int}(I_2) \vdash (I_1 \cap \mathcal{Form}) \cup (I_2 \cap \mathcal{Form}).$$

Combining this result with the transitivity of the proof relation ($\vdash 3$) gives

$$\begin{aligned}
& \{\phi \in \mathcal{Form} \mid (I_1 \cup I_2) \cap \mathcal{Form} \vdash \phi\} = \\
& \quad \{\phi \in \mathcal{Form} \mid (I_1 \cap \mathcal{Form}) \cup (I_2 \cap \mathcal{Form}) \vdash \phi\} \subset \\
& \quad \{\phi \in \mathcal{Form} \mid \text{int}(I_1) \cup \text{int}(I_2) \vdash \phi\}.
\end{aligned}$$

□

The commutative diagram corresponding to the above theorem is shown in Figure 6.2.

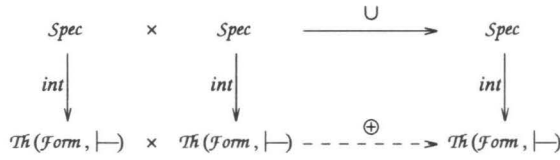


Fig. 6.2. Commutative diagram for theory semantics

Corollaries of Theorem 6.2 are the compositionality of the theory semantics for all kinds of logics like algebraic specifications with (conditional) equational logic (see also Section 3.2.1), extensions of algebraic specifications with associativity and/or lists as described in Sections 3.3.2 and 3.4.2, and first-order logic. In all these examples the *full* theory semantics is meant, meaning that *all* formulae in the specification are part of the theory.

Another corollary of the theorem is that it is possible to give a compositional semantics to grammars. To this end, we consider sets of grammar rules, which are formulae of the form

$$N ::= s_1 s_2 \cdots s_n$$

where N is a non-terminal and s_1, s_2, \dots, s_n are either terminals or non-terminals. With the following proof relation we can describe which grammar rules are derivable from a given grammar. It is defined by the following inference rules:

$$\frac{(N ::= l) \in \mathcal{G}}{\mathcal{G} \vdash N ::= l} \quad (\text{Gr1})$$

$$\frac{\mathcal{G} \vdash N ::= l_1 \ M \ l_3 \quad \mathcal{G} \vdash M ::= l_2}{\mathcal{G} \vdash N ::= l_1 \ l_2 \ l_3} \quad (\text{Gr2})$$

for all grammars \mathcal{G} , non-terminals M and N , and all (possible empty) lists of terminals and non-terminals l, l_1, l_2 and l_3 . The set of grammar rules derivable from a given grammar \mathcal{G} constitute a compositional semantics for grammars.

In the previous Section 6.3.1 we showed that the closed theory semantics of an algebraic specification and the usual open theory semantics of conditional equational logic are not compositional. In both cases, the specification may contain formulae that are not part of the corresponding theory. In the case of the closed theory semantics of an algebraic specification, open equations are allowed in the specification whereas the theory only consists of closed equations. Obviously, these kind of theory semantics are not captured by Definition 6.4 where the set of formulae \mathcal{Form} is assumed to be a subset of \mathcal{Items} . A more general definition of theory semantics is now given and the conditions under which this semantics is compositional are examined.

Definition 6.5:

- An *extended theory semantics* $\langle \mathcal{Form}, Ax, \vdash \rangle$ for a given specification formalism \mathcal{Items} is characterized by:
 - a given set of formulae \mathcal{Form} not necessarily part of \mathcal{Items} as in Definition 6.4,
 - an *axiom operator* $Ax : \mathcal{Spec} \rightarrow P(\mathcal{Form})$ indicating which formulae are the axioms defined in a specification, and
 - A *proof relation* \vdash (see Definition 6.4).
- The semantic domain \mathcal{SD} of an extended theory semantics $\langle \mathcal{Form}, Ax, \vdash \rangle$ is again the set of all theories $\mathcal{Th}(\mathcal{Form}, \vdash)$.
- The interpretation $int : \mathcal{Spec} \rightarrow \mathcal{Th}(\mathcal{Form}, \vdash)$ of an extended theory semantics is defined by

$$int(I) \equiv \{\phi \in \mathcal{Form} \mid Ax(I) \vdash \phi\}.$$

The above definition is indeed an extension of the one in Definition 6.4. If we choose $Ax(I) \equiv I \cap \mathcal{Form}$ for all specifications $I \in \mathcal{Spec}$, we regain the old definition.

The following theorem proves that an extended theory semantics of a given specification formalism is compositional under certain conditions.

Theorem 6.3: Let $\langle \mathcal{Form}, Ax, \vdash \rangle$ be an extended theory semantics for a given specification formalism *Items*, then it is compositional if its proof relation is correct (see Definition 6.4) and its axiom operator satisfies the conditions

$$\phi \in Ax(I_1) \Rightarrow Ax(I_1 \cup I_2) \vdash \phi \text{ and} \quad (Ax1)$$

$$\phi \in Ax(I_1 \cup I_2) \Rightarrow Ax(I_1) \cup Ax(I_2) \vdash \phi \quad (Ax2)$$

for all specifications $I_1, I_2 \in Spec$ and formulae $\phi \in \mathcal{Form}$. Under these circumstances the operator \oplus is again defined as the deductive closure of the union of both theories, i.e.,

$$\mathcal{Th}_1 \oplus \mathcal{Th}_2 \equiv \{\phi \in \mathcal{Form} \mid \mathcal{Th}_1 \cup \mathcal{Th}_2 \vdash \phi\}.$$

for all theories $\mathcal{Th}_1, \mathcal{Th}_2 \in \mathcal{Th}(\mathcal{Form}, \vdash)$.

Proof: To prove compositionality of an extended theory semantics for a given specification formalism *Items*, we have to show that

$$\{\phi \in \mathcal{Form} \mid \text{int}(I_1) \cup \text{int}(I_2) \vdash \phi\} = \{\phi \in \mathcal{Form} \mid Ax(I_1 \cup I_2) \vdash \phi\}$$

holds for all specifications $I_1, I_2 \in Spec$.

First,

$$Ax(I_1 \cup I_2) \vdash Ax(I_1) \cup Ax(I_2), \quad (a)$$

for let $\psi \in Ax(I_1) \cup Ax(I_2)$, then $\psi \in Ax(I_1)$ or $\psi \in Ax(I_2)$, and hence $Ax(I_1 \cup I_2) \vdash \psi$ by (Ax1). Furthermore, it follows from the definition of *int* that $Ax(I_i) \vdash \text{int}(I_i)$ ($i = 1, 2$). Now, by Lemma 6.2(b)

$$Ax(I_1) \cup Ax(I_2) \vdash \text{int}(I_1) \cup \text{int}(I_2). \quad (b)$$

The transitivity of the proof system (\vdash -3) combined with (a) and (b) gives

$$\{\phi \in \mathcal{Form} \mid \text{int}(I_1) \cup \text{int}(I_2) \vdash \phi\} \subset \{\phi \in \mathcal{Form} \mid Ax(I_1 \cup I_2) \vdash \phi\}.$$

Conversely, we start by noting that

$$Ax(I_1) \cup Ax(I_2) \vdash Ax(I_1 \cup I_2) \quad (c)$$

follows immediately from condition (Ax2). Furthermore, $\text{int}(I) \vdash Ax(I)$ for all sets of formulae $I \in Spec$ because $Ax(I) \subset \text{int}(I)$ and (\vdash -1). Consequently, by Lemma 6.2(b)

$$\text{int}(I_1) \cup \text{int}(I_2) \vdash Ax(I_1) \cup Ax(I_2). \quad (d)$$

From the transitivity of the proof system (\vdash -3) combined with (c) and (d), we conclude that

$$\{\phi \in \text{Form} \mid \text{Ax}(I_1 \cup I_2) \vdash \phi\} \subset \{\phi \in \text{Form} \mid \text{int}(I_1) \cup \text{int}(I_2) \vdash \phi\}$$

which terminates the proof. \square

The commutative diagram corresponding to the above theorem is shown in Figure 6.3, where the function $\text{close} : P(\text{Form}) \rightarrow \mathcal{Th}(\text{Form}, \vdash)$ returns the closure of a set of formulae with respect to \vdash .

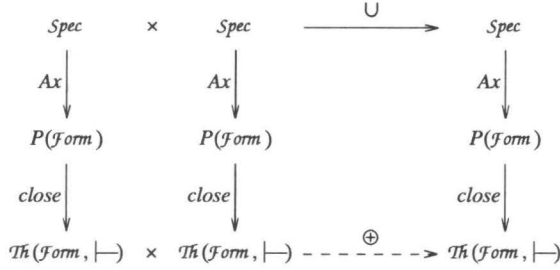


Fig. 6.3. Commutative diagram for extended theory semantics

Both conditions (Ax1) and (Ax2) of the axiom operator in Theorem 6.3 are needed to prove compositionality of an extended theory semantics with \oplus defined by

$$\mathcal{Th}_1 \oplus \mathcal{Th}_2 \equiv \{\phi \in \text{Form} \mid \mathcal{Th}_1 \cup \mathcal{Th}_2 \vdash \phi\}.$$

If either of the conditions is not fulfilled, the extended theory semantics of a specification could still be compositional. But the natural definition of \oplus defined as the closure of the union of both theories will not work. If condition (Ax1) is not fulfilled, there are specifications $I_1, I_2 \in \text{Spec}$, and a formula $\phi \in \text{Ax}(I_1)$ such that $\text{Ax}(I_1 \cup I_2) \vdash \phi$ does not hold. As $\phi \in \text{Ax}(I_1)$, we have $\text{int}(I_1) \vdash \phi$. The latter implies that $\text{int}(I_1) \cup \text{int}(I_2) \vdash \phi$, so $\phi \in \text{int}(I_1) \oplus \text{int}(I_2)$. On the other hand, the negation of $\text{Ax}(I_1 \cup I_2) \vdash \phi$ implies immediately that $\phi \notin \text{int}(I_1 \cup I_2)$. An analogous proof shows that condition (Ax2) is essential to ensure compositionality of the extended theory semantics with respect to \oplus .

The first condition on the axiom operator (Ax1) is a natural requirement, in contrast to the second one (Ax2). Suppose, for example, that we would like to define the semantics of a specification to be the trivial theory of all tautologies if it is semantically incorrect, i.e., $\text{int}(I) \equiv \{\phi \in \text{Form} \mid \emptyset \vdash \phi\}$ or in other words $\text{Ax}(I) \equiv \emptyset$. Suppose, furthermore, that we can find two semantically incorrect specifications $I_1, I_2 \in \text{Spec}$ which make up a correct specification with a non-trivial theory. Hence, there is a non-trivial formula ϕ such that $\phi \in \text{Ax}(I_1 \cup I_2)$, and not $\text{Ax}(I_1) \cup \text{Ax}(I_2) \vdash \phi$.

The previous Section 6.3.1 showed, among other things, that the closed theory semantics of an algebraic specification is not compositional. This fact may seem to contradict the following instance of Theorem 6.3. First, choose as set of formulae \mathcal{Form} the set of all closed equations (i.e., the set of all equations without variables). Next, restrict the proof relation of (conditional) equational logic (see Section 3.2.1) to closed equations. Finally, define the axiom operator for an algebraic specification S as the set of closed instances of equations in S . As it is true in equational logic that each closed equation which is derivable from a set of equations is also derivable from a suitable set of closed instances of these equations, the above definition does result in the closed theory semantics for algebraic specifications; but Theorem 6.3 is not applicable because condition (Ax2) does not hold for this definition of the axiom operator. If we examine the specifications

$$I_1 \equiv \{a : \rightarrow S, f : S \rightarrow S, x \in \text{Var}(S), f(x) = x\} \text{ and}$$

$$I_2 \equiv \{b : \rightarrow S\}$$

then we have $f(b) = b \in \text{Ax}(I_1 \cup I_2)$, but it is not possible to prove $f(b) = b$ from $\text{Ax}(I_1) \cup \text{Ax}(I_2)$ which is equal to $\{f(a) = a\}$.

6.3.3 Model class semantics

Now, a definition of the model class semantics of a specification formalism and the proof of its compositionality will be given. These are similar to the definitions and proofs of the compositionality of theory semantics.

Definition 6.6:

- A *model class semantics* $\langle \mathcal{Form}; \text{Ax}, \mathcal{Mod}, \models \rangle$ for a given specification formalism Items consists of:
 - a given set of formulae \mathcal{Form} not necessarily part of Items as in Definition 6.4,
 - an *axiom operator* $\text{Ax} : \text{Spec} \rightarrow P(\mathcal{Form})$ indicating which formulae are the axioms defined in a specification,
 - a class of models \mathcal{Mod} , and
 - a *truth relation* \models which is a relation on $\mathcal{Mod} \times \mathcal{Form}$ stating which formulae are true in a model.

As usual, if ϕ is true in M we write $M \models \phi$. $M \models \Gamma$ with $\Gamma \subset \mathcal{Form}$ a set of formulae is an abbreviation for $M \models \psi$ for all $\psi \in \Gamma$.

- The semantic domain \mathcal{SD} of a model class semantics $\langle \mathcal{Form}, \text{Ax}, \mathcal{Mod}, \models \rangle$ is the power set of the set of models $P(\mathcal{Mod})$.

- The interpretation $int : Spec \rightarrow P(Mod)$ of a model class semantics is defined by

$$int(I) \equiv \{M \in Mod \mid M \models Ax(I)\}.$$

Theorem 6.4: Let $\langle Form, Ax, Mod, \models \rangle$ be a model class semantics for a given specification formalism *Items*. It is compositional with respect to the intersection operator on classes of models if its axiom operator Ax fulfills the condition

$$M \models Ax(I_1 \cup I_2) \Leftrightarrow M \models Ax(I_1) \cup Ax(I_2). \quad (Ax3)$$

Proof: Let a specification formalism with a model class semantics be given which fulfills the conditions stated in the theorem, then we have to prove for all $I_1, I_2 \in Spec$

$$\begin{aligned} \{M \in Mod \mid M \models Ax(I_1)\} \cap \{M \in Mod \mid M \models Ax(I_2)\} = \\ \{M \in Mod \mid M \models Ax(I_1 \cup I_2)\}. \end{aligned}$$

This is trivial as it follows immediately from (Ax3). \square

In Figure 6.4 the commutative diagram for model class semantics is shown.

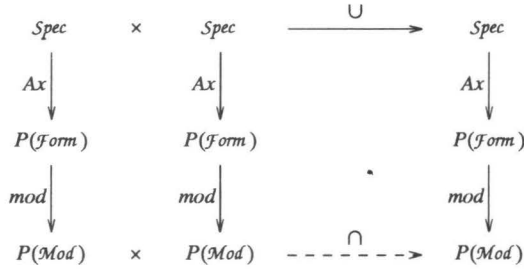


Fig. 6.4. Commutative diagram for model class semantics

The function $mod : P(Form) \rightarrow P(Mod)$ assigns to each set of formulae the set of models in which those formulae are true, i.e., $mod(\Gamma) \equiv \{M \in Mod \mid M \models \Gamma\}$.

Corollaries of Theorem 6.4 are the compositionality of the class of all models and the class of all countable models of an algebraic specification or any other first-order logic. Note that *all* models in which the axioms of a specification are true are needed in the model class semantics. In case of an algebraic specification $\langle \Sigma, \mathcal{E} \rangle$ with signature Σ and set of (possibly conditional) equations \mathcal{E} , the class of all Σ -algebras which satisfy \mathcal{E} is not a model class semantics in the sense of Definition 6.6. The class of all Σ' -algebras satisfying \mathcal{E} where $\Sigma \subset \Sigma'$ is, however, an appropriate model class semantics.

Comparison of the diagrams in Figures 6.3 and 6.4 suggests an alternative way to prove Theorems 6.3 and 6.4 if an extended theory semantics and a model class semantics are defined for a specification formalism. We only have to prove that the diagram in Figure 6.5 commutes.

$$\begin{array}{ccccc}
 \mathcal{Th}(\mathcal{Form}, \vdash) & \times & \mathcal{Th}(\mathcal{Form}, \vdash) & \xrightarrow{\oplus} & \mathcal{Th}(\mathcal{Form}, \vdash) \\
 \text{mod} \downarrow & & \text{mod} \downarrow & & \text{mod} \downarrow \\
 P(\mathcal{Mod}) & \times & P(\mathcal{Mod}) & \xrightarrow{\cap} & P(\mathcal{Mod})
 \end{array}$$

Fig. 6.5. Commutative diagram for extended theory semantics and model class semantics

This can only be proved if the proof relation of the extended theory semantics is *sound* with respect to the truth relation of the model class semantics.

Definition 6.7: Let $\langle \mathcal{Form}, Ax, \vdash \rangle$ be an extended theory semantics and $\langle \mathcal{Form}, Ax, \mathcal{Mod}, \models \rangle$ be a model class semantics for a given specification formalism *Items* both based on the same set of formulae \mathcal{Form} and axiom operator Ax . The proof relation \vdash of $\langle \mathcal{Form}, Ax, \vdash \rangle$ is called *sound* with respect to the truth relation \models of $\langle \mathcal{Form}, Ax, \mathcal{Mod}, \models \rangle$ if

$$\Gamma \vdash \phi \Rightarrow \left[\forall M \in \mathcal{Mod} \ M \models \Gamma \Rightarrow M \models \phi \right]$$

holds for all sets of formulae $\Gamma \subset \mathcal{Form}$ and for all formulae $\phi \in \mathcal{Form}$.

To prove commutativity of the diagram in Figure 6.5 we have to show that

$$\text{mod}(\mathcal{Th}_1) \cap \text{mod}(\mathcal{Th}_2) = \text{mod}(\mathcal{Th}_1 \oplus \mathcal{Th}_2)$$

holds for all theories $\mathcal{Th}_1, \mathcal{Th}_2 \in \mathcal{Th}(\mathcal{Form}, \vdash)$. In other words, we have to prove

$$\begin{aligned}
 \{M \in \mathcal{Mod} \mid M \models \mathcal{Th}_1\} \cap \{M \in \mathcal{Mod} \mid M \models \mathcal{Th}_2\} = \\
 \{M \in \mathcal{Mod} \mid M \models \mathcal{Th}_1 \oplus \mathcal{Th}_2\}.
 \end{aligned}$$

The proof of this is straightforward.

6.3.4 Conclusions

In the previous sections several specification formalisms were presented with their compositional semantics. These formalisms are appropriate for textual modularization. It is unfortunate that the most natural semantics of a specification formalism (like the generated language in case of a grammar, and the initial algebra in case of

an algebraic specification) are not compositional. In view of this, there are several possibilities:

- First, we can look for other module operators and base modularization of the specification formalism at hand on those operators. A module operator could, for instance, forbid certain combinations of specifications, or it could (automatically) rename parts of the specification to prevent from undesired interference. Examples of this are the module operator on grammars which refuses to combine grammars that contain identical non-terminals (apart from the start symbol), or renames the non-terminals of the grammars before combining them.
- A second possibility is to choose a compositional semantics for a given specification formalism. The theory semantics as well as the model class semantics are well suited for textual modularization whereas the initial algebra and the closed theory semantics are not.
- Finally, one can choose to be contented with a semantics that is not compositional, and still base modularization on textual expansion. If the formalism consists of a set of items and if the semantics is easy to understand, it is no problem that the semantics of a composite specification cannot be expressed in terms of the semantics of the composing parts. It is of course desirable to warn the user of such a formalism about this inconvenience, as it is no longer possible to understand a complex specification by inspecting the semantics of all of the parts of it. In ASF as well as in ASF+SDF this option was chosen.

6.4 Algebraic specification of textual modularization

In this section an algebraic specification of textual modularization is given. As before, a sharp distinction is made between issues related to textual modularization itself, and issues related to the specification formalism to which textual modularization is added. Therefore, the specification of textual modularization has as parameter the non-modular specification formalism. In this way, families of modular specification formalisms that all share the same modular constructs are created.

In four subsections the basic notions, the specification formalisms, the modular specifications corresponding to them, and an example of the modular specification of signatures are respectively discussed.

6.4.1 Sets

In the following parameterized module `Sets`, sets of items of an arbitrary abstract data type are defined. Upon import of this module in another module, the formal

sort `ITEM` in the parameter can be bound to any actual sort to obtain a specification of sets of items of that sort.

The syntax of sets is defined in the function

```
"{" {ITEM " , "}* "}" -> SET.
```

This module furthermore contains definitions of functions to calculate the union of two sets (+), to remove an element from a set (-), and to test whether an element is in a set (`elm`). To define the latter, a specification of Booleans is needed and to this end the module `Booleans` from Section 4.3.1.2 is used. Via this import, module `Layout` of Section 4.3.1.1 is indirectly import imported. This module is also used in the rest of the specification to define its layout. The priority declaration `"-" > "+"` disambiguates expressions like `{a, b} + {c, d} - b`. Note that without disambiguation both possible parses of this sentence represent different sets.

When testing this specification in the ASF+SDF system (see Chapter 5), we obviously have to remove equation [8] in which the transposition of elements of a set is stated. Each term containing a set with at least two different elements, would otherwise result in an infinite loop when evaluating that term. Without equation [8] it is perfectly possible to test the specification, since equation [7] takes care of the removal of identical elements. One has to be aware of the fact, however, that the data type given in the module is no longer sets but lists in which only the last occurrence of several identical elements is retained. When testing equality or inequality of the left- and right-hand sides of conditions of an equation, the system will not compare them as sets.

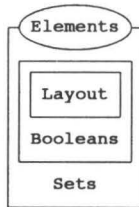


Fig. 6.6. Structure diagram of `Sets`

```

module Sets

imports Booleans

parameter Elements
  sorts ITEM

exports
  sorts SET
  context-free syntax

```

```

"{" {ITEM " , "}* "}" -> SET
SET "+" SET           -> SET assoc % union %
SET "-" ITEM          -> SET           % delete an item %
ITEM elm SET          -> BOOL
"(" SET ")"           -> SET bracket

priorities
  "-" > "+"

hiddens
  variables
    i [12]* -> ITEM
    x [1-3]* -> {ITEM " , "}*
    y [12]  -> {ITEM " , ")+

equations
[7]  {x1, i, x2, i, x3} = {x1, x2, i, x3}
[8]  {y1, y2}           = {y2, y1}
[9]  {x1} + {x2} = {x1, x2}
[10] {} - i       = {}
[11] {i, x} - i = {x} - i
[12] i1 != i2 ==> {i1, x} - i2 = {i1} + ({x} - i2)
[13] i elm {}     = false
[14] i elm {i, x} = true
[15] i1 != i2 ==> i1 elm {i2, x} = i1 elm {x}

```

6.4.2 Non-modular specification formalisms

As sets of items and sets of error messages are needed in the specification of non-modular specification formalisms, modules *Items* and *Errors-of-Spec-Form* are first defined. These will be bound to the parameter of *Sets* in the imports of module *Specification-Formalisms*. Since we would like to reuse this specification for different non-modular specification formalisms, both module *Items* as well as *Errors-of-Spec-Form* are themselves parameterized modules. In Section 6.4.4 one example of this for a non-modular formalism for the specification of signatures is given.

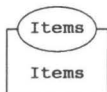


Fig. 6.7. Structure diagram of *Items*

```

module Items
  parameter Items
    sorts ITEM
  exports
    variables
      item -> ITEM

```

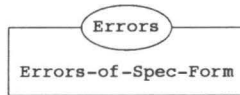


Fig. 6.8. Structure diagram of Errors-of-Spec-Form

```

module Errors-of-Spec-Form
  parameter Errors
    sorts SPEC-FORM-ERROR

```

The generic definition of non-modular specification formalisms in module *Specification-Formalisms* has an extra parameter *Typechecker* which has to be actualized with the typechecker of the non-modular formalism in question. It contains a function which returns the set of error messages corresponding to a set of items of the formalism. With this parameter the total number of parameters of this module is three: the items of the formalism in the inherited parameter *Items*, the error messages in the inherited parameter *Errors*, and the typechecker in *Typechecker*.

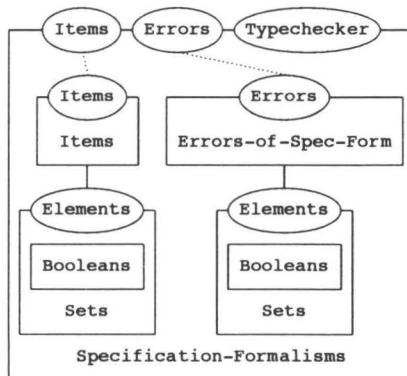


Fig. 6.9. Structure diagram of Specification-Formalisms

```

module Specification-Formalisms

imports
  Sets
    Elements bound by
      sorts ITEM => ITEM
    to Items
    renamed by
      sorts SET => ITEM-SET
  Sets
    Elements bound by
      sorts ITEM => SPEC-FORM-ERROR
    to Errors-of-Spec-Form
    renamed by
      sorts SET => SPEC-FORM-ERRORS

parameter Typechecker
  context-free syntax
  tc ITEM-SET -> SPEC-FORM-ERRORS

exports
  variables
    items -> {ITEM " , "}*
    iset  -> ITEM-SET

```

6.4.3 Modular specification formalisms

6.4.3.1 Modular-Specifications

As each modular specification is a set of modules, these are defined in a separate module and `Sets` is actualized with it. Each module is a tuple containing the name of the module, the imports as a set of module names, and a set of items.

The lexical definition of module names is given in the following module `Module-Names`. Module names are lists of one or more letters, digits, underscores, or minus-signs starting with a capital letter and not ending in an underscore or minus-sign.

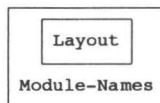


Fig. 6.10. Structure diagram of `Module-Names`

```

module Module-Names

imports Layout

```

```

exports
  sorts NAME
  lexical syntax
    [A-Z]                                     -> NAME
    [A-Z] [A-Za-z0-9\_\-]* [A-Za-z0-9] -> NAME
  variables
    name ['']* -> NAME
    imp      -> NAME

```

A module as tuple of a module name (NAME), imports (NAME-SET), and items (ITEM-SET) is defined in module Modules. We also define three functions (mod-name, imports, and items) which return one of the corresponding elements from the tuple.

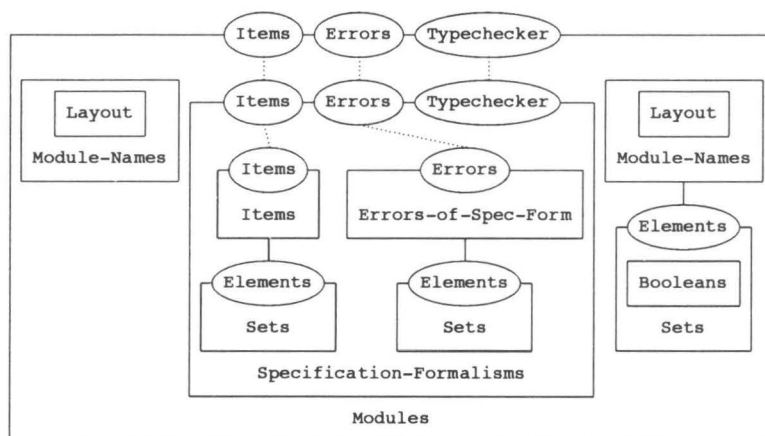


Fig. 6.11. Structure diagram of Modules

```

module Modules

imports Module-Names Specification-Formalisms
  Sets
    Elements bound by
      sorts ITEM => NAME
    to Module-Names
    renamed by
      sorts SET => NAME-SET

exports
  sorts MODULE
  context-free syntax
    "<" NAME "," NAME-SET "," ITEM-SET ">" -> MODULE
    modname of MODULE                      -> NAME
    "imports" of MODULE                     -> NAME-SET

```

```

items of MODULE                                -> ITEM-SET
variables
  names -> {NAME " , "}*
  impls -> NAME-SET
  nset  -> NAME-SET
  mod   -> MODULE

equations
[16] modname of <name, impls, iset> = name
[17] imports of <name, impls, iset> = impls
[18] items of <name, impls, iset> = iset

```

Modular-Specifications now consist of a simple import of Sets actualized with Modules.

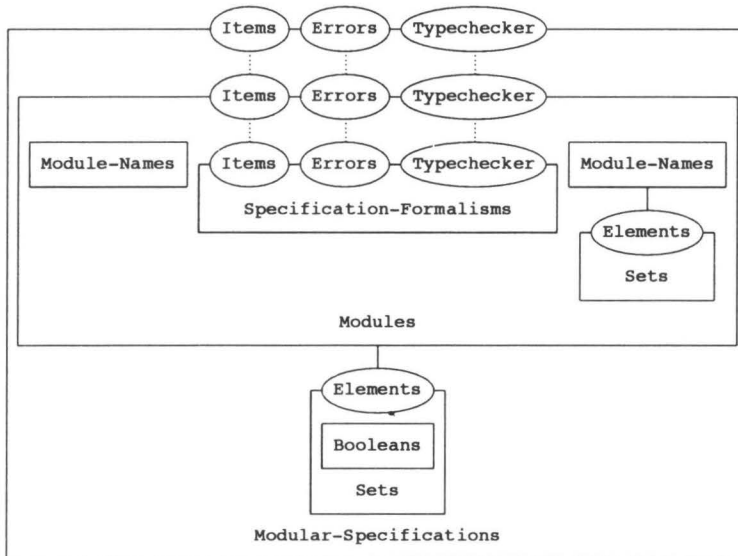


Fig. 6.12. Structure diagram of Modular-Specifications

```

module Modular-Specifications
imports
  Sets
  Elements bound by
    sorts ITEM => MODULE
  to Modules
  renamed by
    sorts SET => MOD-SPEC

```

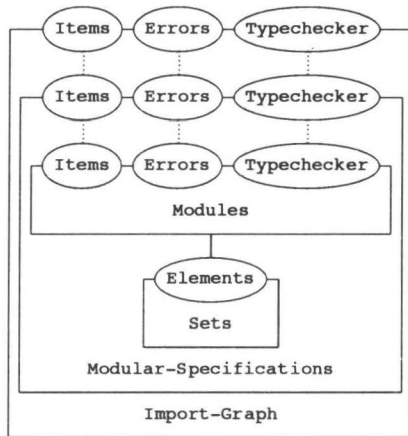



Fig. 6.13. Structure diagram of Import-Graph

```
"imp*" MODULE in MOD-SPEC with known MOD-SPEC -> MOD-SPEC
lookup NAME-SET in MOD-SPEC                      -> MOD-SPEC
```

equations

```
[19] imp+ mod in spec
    = imp+ mod in (spec + {mod}) with known {}

[20] imp+ mod in spec with known spec'
    = imp* (lookup imports of mod in spec)
      in spec with known spec'

[21] imp* {} in spec with known spec' = spec'
[22] imp* mod in spec with known spec' = spec''
    =====
    imp* {mod, mods} in spec with known spec'
    = imp* {mods} in spec with known spec''

[23]   mod elm spec' = true
    =====
    imp* mod in spec with known spec' = spec'

[24]   mod elm spec' = false
    =====
    imp* mod in spec with known spec'
    = imp+ mod in spec with known (spec' + {mod})

[25] lookup nset in {} = {}
[26]   modname of mod elm nset = true
    =====
    lookup nset in {mod, mods}
    = {mod} + lookup nset in {mods}
```

```
[27]      modname of mod elm nset = false
=====
      lookup nset in {mod, mods} = lookup nset in {mods}
```

6.4.3.3 Normalization

With the previous specification of the import graph, it is very easy to define the normal form of a module in the context of a modular specification. We only need a hidden function `items` which returns the set of all items present in the modules of a specification.

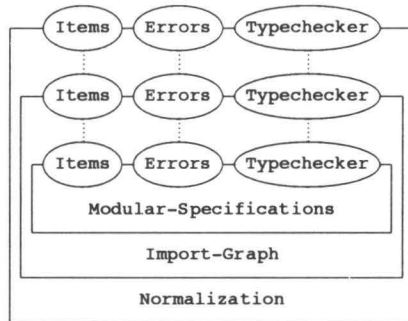


Fig. 6.14. Structure diagram of Normalization

```
module Normalization
imports Import-Graph
exports
  context-free syntax
    nf MODULE in MOD-SPEC -> ITEM-SET
hiddens
  context-free syntax
    items MOD-SPEC -> ITEM-SET
equations
[28] nf mod in spec = items ({mod} + imp+ mod in spec)
[29] items {}          = {}
[30] items {mod, mods} = items of mod + items {mods}
```

6.4.3.4 Typechecking

Before defining the typechecking of modular specifications, the syntax of error messages is given in the following module `Errors`.

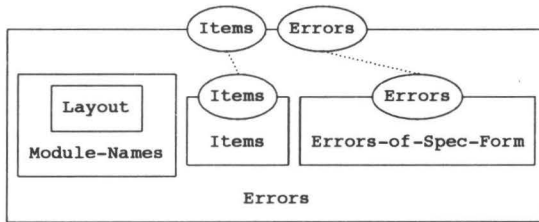


Fig. 6.15. Structure diagram of Errors

```

module Errors
imports Module-Names Items Errors-of-Spec-Form
exports
  sorts ERROR
  context-free syntax
    "module" NAME defined more than once          -> ERROR
    "module" NAME ":"
      imported "module" NAME not yet defined      -> ERROR
    "module" NAME ":" cyclic import                -> ERROR
    "module" NAME ":"
      item ITEM has different origins NAME and NAME -> ERROR
    "module" NAME ":" SPEC-FORM-ERROR              -> ERROR

```

To complete the definition of modular specifications, the following module `Typechecking` defines the typechecking of such specifications. The exported function

```
tc MOD-SPEC -> ERRORS
```

returns the set of error messages of a modular specification. A hidden function

```
tc MOD-SPEC in MOD-SPEC -> ERRORS
```

is used to find the error messages for each of the modules of a specification in the context of the complete specification. There are the following hidden functions each of which corresponds to precisely one of the semantic constraints defined in Definition 6.3:

- The function `unique-name` checks whether the name of a module does not occur among the names of the other modules.
- The function `check-imps` gives error messages for all names in the imports of a module for which no module exists in the specification.
- The function `no-cycles` returns an error message whenever a module occurs in the transitive closure of its imports.

- The function `org-rule` checks the origin rule by traversing the reflexive and transitive closure of the imports looking for items which occur in more than one module. To this end, two hidden functions

```

item-check ITEM-SET org NAME in MOD-SPEC of NAME
                                                    -> ERRORS, and
item-check ITEM org NAME in MOD-SPEC of NAME -> ERRORS

```

are used. The first of these has as first argument a set of items which all have as origin a module whose name is the second argument of the function. An error message is returned whenever an item is found in the rest of the import graph that is identical to any of the items from this set.

- The function `convert` prefixes the name of a module to the error messages found by the typechecker of the formalism when applied to the normal form of that module.

A function `modnames` which returns the set of all module names of a given specification and an if-then-else-fi function for error messages are used.

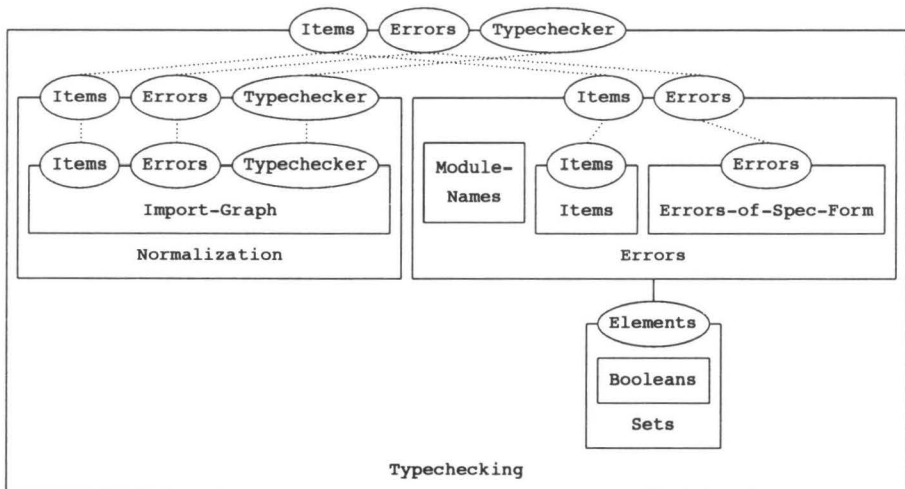


Fig. 6.16. Structure diagram of Typechecking

```

module Typechecking
imports Normalization
  Sets
    Elements bound by
      sorts ITEM => ERROR
    to Errors

```

```

renamed by
  sorts SET => ERRORS

exports
  context-free syntax
    tc MOD-SPEC -> ERRORS

hiddens
  context-free syntax
    tc MOD-SPEC in MOD-SPEC -> ERRORS
    unique-name NAME in NAME-SET -> ERRORS
    check-imps NAME-SET of NAME in NAME-SET -> ERRORS
    no-cycles MODULE in MOD-SPEC -> ERRORS
    org-rule MOD-SPEC of NAME -> ERRORS
    convert SPEC-FORM-ERRORS of NAME -> ERRORS
    modnames of MOD-SPEC -> NAME-SET
    if BOOL then ERRORS else ERRORS fi -> ERRORS
    item-check ITEM-SET org NAME in MOD-SPEC of NAME
                                          -> ERRORS
    item-check ITEM org NAME in MOD-SPEC of NAME
                                          -> ERRORS

variables
  spec-er -> SPEC-FORM-ERROR
  spec-ers -> {SPEC-FORM-ERROR ", "}*
  ers [12] -> ERRORS

equations
[31] tc spec = tc spec in spec
[32] tc {} in spec = {}
[33]   name = modname of mod,
      impgraph = imp+ mod in spec
      =====
      tc {mod, mods} in spec
      = unique-name name in (modnames of {mods}-mod) +
        check-imps (imports of mod) of name
          in (modnames of spec) +
        no-cycles mod in impgraph +
        org-rule {mod} + impgraph of name +
        convert tc nf mod in spec of name +
        tc {mods} in spec
[34] unique-name name in nset
      = if name elm nset
        then {module name defined more than once}
        else {}
        fi
[35] check-imps {} of name in nset = {}

```

```

[36] check-imps {imp, names} of name in nset
    = if imp elm nset
      then {}
      else {module name: imported module imp not yet defined}
      fi +
      check-imps {names} of name in nset
[37] no-cycles mod in impgraph
    = if mod elm impgraph
      then {module modname of mod: cyclic import}
      else {}
      fi
[38] org-rule {} of name = {}
[39] org-rule {mod, mods} of name
    = item-check (items of mod) org modname of mod
      in {mods}-mod of name +
      org-rule {mods} of name
[40] convert {} of name = {}
[41] convert {spec-er, spec-ers} of name
    = {module name: spec-er} + convert {spec-ers} of name
[42] modnames of {} = {}
[43] modnames of {mod, mods}
    = {modname of mod} + modnames of {mods}
[44] if true then ers1 else ers2 fi = ers1
[45] if false then ers1 else ers2 fi = ers2
[46] item-check {} org name in spec of name' = {}
[47] item-check {item, items} org name in spec of name'
    = item-check item org name in spec of name' +
      item-check {items} org name in spec of name'
[48] item-check item org name in {} of name' = {}
[49] item-check item org name in {mod, mods} of name'
    = if item elm items of mod
      then {module name':
           item item has different origins
           name and modname of mod}
      else {}
      fi +
      item-check item org name in {mods} of name'

```

6.4.4 Modular specification of signatures

In this section, the specification of a non-modular specification formalism in which many-sorted signatures can be specified, is first given. Next, textual modularization is added to it by instantiating the specification of modular specifications with it.

In the first module, the elements of the non-modular formalism to specify signatures is defined. These are declarations of sorts and of functions with their input type and result type.

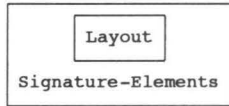


Fig. 6.17. Structure diagram of Signature-Elements

```

module Signature-Elements

imports Layout

exports
  sorts SORT FNC-NAME FUNCTION SIG-ELEM
  lexical syntax
    [A-Z]                                -> SORT
    [A-Z] [A-Za-z0-9\_]* [A-Za-z0-9] -> SORT
    [a-z]                                -> FNC-NAME
    [a-z] [A-Za-z0-9\_]* [A-Za-z0-9] -> FNC-NAME
  context-free syntax
    FNC-NAME ":" {SORT "#"}* "->" SORT -> FUNCTION
    SORT                                           -> SIG-ELEM
    FUNCTION                                       -> SIG-ELEM
  variables
    srt -> SORT
    srts -> {SORT "#"}*
    fnc -> FUNCTION
    Name -> FNC-NAME
    item -> SIG-ELEM
  
```

The typechecking of signatures discovers only one type of errors: an error message is given whenever an undeclared sort is used in a function. In the following module Signature-Errors the syntax of this error message is specified.

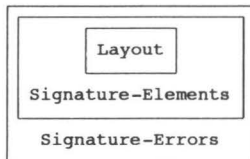


Fig. 6.18. Structure diagram of Signature-Errors

```

module Signature-Errors
imports Signature-Elements
exports
  sorts SIG-ERROR
  context-free syntax
  undeclared sort SORT in function FUNCTION -> SIG-ERROR

```

In Signatures both previous modules are used as actual modules in the parameter binding of module Sets to define respectively sets of signature elements and sets of error messages. The typechecker of the formalism is also given in this module.

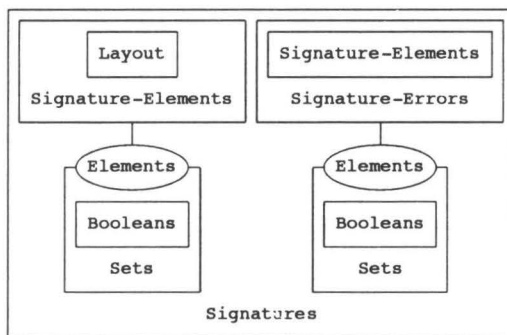


Fig. 6.19. Structure diagram of Signatures

```

module Signatures
imports
  Sets
    Elements bound by
      sorts ITEM => SIG-ELEM
    to Signature-Elements
    renamed by
      sorts SET => SIGNATURE
  Sets
    Elements bound by
      sorts ITEM => SIG-ERROR
    to Signature-Errors
    renamed by
      sorts SET => SIG-ERRORS
exports
  context-free syntax
  tc SIGNATURE -> SIG-ERRORS
hiddens
  context-free syntax

```

```

tc SIGNATURE in SIGNATURE          -> SIG-ERRORS
tc FUNCTION in SIGNATURE            -> SIG-ERRORS
tc {SORT "#"}* of FUNCTION in SIGNATURE -> SIG-ERRORS
variables
  elms -> {SIG-ELEM ","}*
  sig  -> SIGNATURE

equations

[50] tc sig = tc sig in sig
[51] tc {} in sig          = {}
[52] tc {srt, elms} in sig = tc {elms} in sig
[53] tc {fnc, elms} in sig = tc fnc in sig + tc {elms} in sig
[54]   srt elm sig = true
      =====
      tc Name: srts -> srt in sig
      = tc srts of Name: srts -> srt in sig
[55]   srt elm sig = false
      =====
      tc Name: srts -> srt in sig
      = tc srts of Name: srts -> srt in sig +
        {undeclared sort srt in function Name: srts -> srt}
[56] tc of fnc in sig = {}
[57]   srt elm sig = true
      =====
      tc srt # srts of fnc in sig = tc srts of fnc in sig
[58]   srt elm sig = false
      =====
      tc srt # srts of fnc in sig
      = {undeclared sort srt in function fnc} +
        tc srts of fnc in sig

```

Finally, the typechecker is specified by binding the parameters of the module Typechecking to the appropriate actual modules Signature-Elements, Signature-Errors, and Signatures.

```

module Signature-Typechecking

imports
  Typechecking
  Items bound by
    sorts ITEM => SIG-ELEM
  to Signature-Elements
  Errors bound by
    sorts SPEC-FORM-ERROR => SIG-ERROR
  to Signature-Errors
  renamed by

```

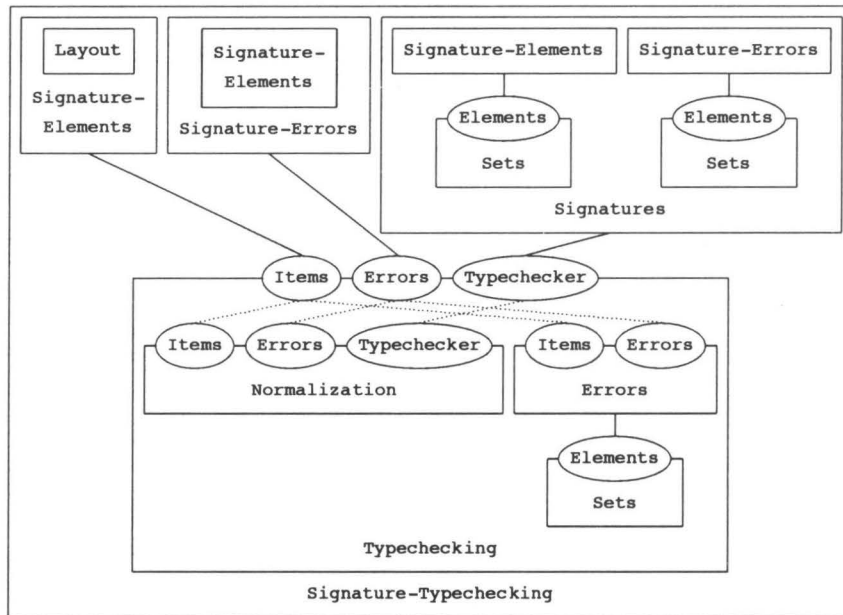


Fig. 6.20. Structure diagram of Signature-Typechecking

```

sorts
  ITEM-SET      => SIGNATURE
  SPEC-FORM-ERRORS => SIG-ERRORS
Typechecker bound
to Signatures

```

In this way, a specification of modular signature definitions and their typechecking is obtained by completely reusing textual modularization constructs, and related typechecking and normalization as defined earlier in the parameterized specification in Section 6.4.3. Continuing in this direction a family of modular specification formalisms sharing the same modular constructs can be created.

Incremental Processing of Modular Specifications

In the previous chapter specification formalisms whose modularization is based on textual expansion have been discussed. Here, the global architecture of a system for the incremental processing of specifications written in such formalisms is described as well as the main algorithms used in its implementation.

7.1 Introduction

A modular specification consists of a number of modules labeled with unique module names. Each module may contain a list of one or more names of modules that have to be imported in it. If modularization is based on textual expansion, the meaning of such an import is that each name in the import list has to be replaced by the text of the corresponding module.

Examples of specification formalisms suited for this form of modularization are formalisms to specify grammars, algebraic specifications, or logics. In each of these cases a specification consists of a *set* of different elements, such as grammar rules, non-terminals, sorts, functions, equations, or axioms. As in the previous chapter, the term *items* will be used when referring to these elements of a formalism. It is necessary that a specification consists of a *set* of items if the imports in a module are a *set* of module names.

From an implementor's point of view, textual expansion introduces two problems:

- It leads to a combinatorial explosion of the size of normalized, i.e., completely expanded modules. When generating code for the normal forms of modules independently of each other, the generated codes will often have parts in common.
- It does not lend itself to the *incremental* processing of specifications as each modification in a module influences the generated code of all modules in which

that module is imported. (Incrementality means that the time needed to process a change in a specification is proportional to the size of the change.)

In this chapter the implementation of formalisms whose modularization is based on textual expansion is studied and it is shown how both problems can be solved. Starting points for this implementation are:

- The implementation has to be *incremental* as it is to be used in an interactive environment in which the user can edit a modular specification and test it immediately.
- The processing of the modular structure of the specification should be separated from the processing of other items. The advantage of this approach is that all information on the modular structure can be concentrated. In this way, it is also possible to study how to modularize the implementation of non-modular specification formalisms.

This results in the global architecture of the implementation shown in Figure 7.1.

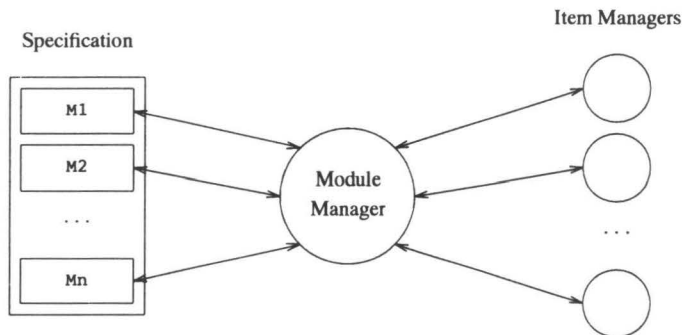


Fig. 7.1. Architecture of the implementation of a modular specification formalism

A modular specification is processed by a *module manager* which handles the modular structure of the specification and directs information to one or more *item managers*. Each of these item managers is responsible for the implementation of parts of the non-modular specification formalism. In the following Section 7.2, it is investigated which functionality should be provided by item managers to be used in an architecture as shown in Figure 7.1. The functionality of the module manager is also described in it. Section 7.3 contains the description of the three main algorithms needed in the implementation of the module manager. The final Section 7.4 describes how the techniques presented in this chapter have been applied in the ASF+SDF system (see Chapter 5).

7.2 Functionality of system components

7.2.1 Item managers

Each item manager has to know the set of items in the specification for which it is responsible. Each module in the specification can be represented as a subset of this global set of items. We shall call such a subset a *selection*. As we do not give preference to one of the modules in the specification as the only module to be used, the item manager should be prepared to switch between modules by using selections.

We assume that each item manager manages a two-dimensional table whose first index contains the items of the specification and the second index contains the selections. The contents of this table indicate whether an item is an element of a selection or not. If an item is an element of a selection we call it *enabled* in that selection, otherwise it is *disabled*. As an example, consider the following specification:

| | | | | | | |
|--------------------------------------|---|----|----|----|----|----|
| module M1 items a, b | | M1 | M2 | M3 | M4 | M5 |
| module M2 items c | a | x | | x | x | x |
| module M3 imports M1 items d | b | x | | x | x | x |
| module M4 imports M3 items c, e | c | | x | | x | x |
| module M5 imports M1, M2, M3 items f | d | | | x | x | x |
| | e | | | | x | |
| | f | | | | | x |

The table on the right-hand side shows the corresponding two-dimensional table where enabled items are indicated with a x.

The functionality of an item manager follows immediately from the necessity to manipulate these two-dimensional tables. First of all, an item manager can be created using the function

```
itemman-create() → <itemman>.
```

It creates a new item manager that contains an empty table.

To update the set of items known to the item manager it has to provide functions

```
add-item(<itemman>, <item>) → <handle> and
del-item(<itemman>, <handle>).
```

With these functions we can add a row to the table or delete a row from it. The <handle> which add-item returns may be any structure identifying the item to the item manager. If an item is added more than once, a single deletion of that item should not result in its complete removal. This property is needed to handle identical items defined in different modules (as item c in the above example). If these

modules are (directly or indirectly) imported in one module it causes a violation of the origin rule (see Section 6.2.2 and [BHK89b]).

The functions

```
new-selection(<itemman>) → <selection> and
del-selection(<itemman>, <selection>)
```

are necessary to add a column to or delete a column from the table. In the example, it is suggested that selections are labeled with module names. These module names are, however, irrelevant to the item manager. Therefore, the only argument of the function `new-selection` is the item manager itself.

When an item or a selection is added to the table, all its entries are disabled. The functions

```
enable(<itemman>, <selection>, <handle>) and
disable(<itemman>, <selection>, <handle>)
```

are needed to explicitly enable or disable items in selections.

Finally, the function

```
apply(<itemman>, <selection>, [<parameters>])
```

is used to apply the implementation generated for a selection of items to the given parameter values.

An item manager is implemented easiest if the implementation for the non-modular specification formalism is already incremental and can handle selections. Examples of such implementations are the modular scanner generator MSG [Kli91], the modular parser generator MPG [Rek89b] (which are the main parts of the syntax manager (see Section 5.4.1)), and the equation manager (see Section 5.4.2). In each of these cases the generated code can be viewed as a set of states and transitions labeled with items from the specification. When changing the selection the appropriate part of the generated code is selected by inspecting those labels. If an implementation of a specification formalism is not specifically tailored towards incrementality and the handling of selections, it is also possible to use it as item manager. In that case, changing selections may become very expensive.

7.2.2 Module manager

A module manager provides the fundamental operations for the incremental creation and modification of modular specifications, and its functionality is mainly determined by this incremental behavior. As in the case of item managers, we start with an initialization function

```
modman-create() → <modman>
```

which creates a new module manager. Next, functions are needed to add or delete complete modules, and to add or delete imports or items to or from already existing modules:

```
add-module(<modman>, <module>),
del-module(<modman>, <module>),
add-import(<modman>, <module name>, <import>),
del-import(<modman>, <module name>, <import>),
add-item(<modman>, <module name>, <item>), and
del-item(<modman>, <module name>, <item>).
```

Finally, a function

```
apply(<modman>, <module name>, [<parameters>])
```

is needed which applies the implementation generated for the normal form of a module to given parameter values. The algorithms for implementing these functions are given in Section 7.3.2.

7.3 Algorithms

This section describes the three main algorithms needed to implement a module manager and to connect it to a syntax-directed editor.

To update the import graph of a modular specification incrementally, a general algorithm for maintaining the transitive closure of a given binary relation is used. This incremental transitive closure algorithm, whose implementation is inspired by Yellin's algorithms in [Yel88], is described in Section 7.3.1. This algorithm cannot handle binary relations whose transitive closure contains cycles.

Section 7.3.2 gives the algorithms which implement the functions of a module manager as described in Section 7.2.2.

Section 7.3.3 describes how to connect the module manager to a syntax-directed editor. Such an editor is assumed to maintain an abstract syntax tree of the edited text. When coupling a module manager to such an editor, changes in the abstract syntax tree have to be translated to appropriate calls to the module manager. The algorithm in Section 7.3.3 solves this problem in a more general setting: it analyses the differences between old and new abstract syntax tree and translates them into calls to an arbitrary incremental tool such as, for instance, a module manager.

7.3.1 Incremental transitive closure

If R is a binary relation on a given set S i.e. $R \subset S \times S$ the transitive closure $R^+ \subset S \times S$ of R is defined as the least relation satisfying

$$s_1 R^+ s_3 \Leftrightarrow s_1 R s_3 \vee \left(\exists s_2 \in S \quad s_1 R^+ s_2 \wedge s_2 R s_3 \right)$$

The following algorithm is used to maintain the transitive closure of a binary relation incrementally. It updates a structure containing three components:

- the *basis*, which contains the pairs of the relation R ,
- the *closure*, in which all pairs of the transitive closure of R are stored, and
- the *supports*, which are all triples $\langle s_1, s_2, s_3 \rangle$ of elements of S with $s_1 R^+ s_2$ and $s_2 R s_3$.

To illustrate the transitive closure algorithm and the above structure the following example is used:

Basis: $R = \{\langle a, b \rangle, \langle b, c \rangle, \langle b, d \rangle, \langle e, f \rangle\}$
 Closure: $R^+ = \{\langle a, b \rangle, \langle a, c \rangle, \langle a, d \rangle, \langle b, c \rangle, \langle b, d \rangle, \langle e, f \rangle\}$
 Supports: $\{\langle a, b, c \rangle, \langle a, b, d \rangle\}$.

If we add the pair $\langle c, e \rangle$ to the basis R of this example, the situation changes to

Basis: $R = \{\langle a, b \rangle, \langle b, c \rangle, \langle b, d \rangle, \langle c, e \rangle, \langle e, f \rangle\}$
 Closure: $R^+ = \{\langle a, b \rangle, \langle a, c \rangle, \langle a, d \rangle, \langle a, e \rangle, \langle a, f \rangle, \langle b, c \rangle, \langle b, d \rangle, \langle b, e \rangle, \langle b, f \rangle, \langle c, e \rangle, \langle c, f \rangle, \langle e, f \rangle\}$
 Supports: $\{\langle a, b, c \rangle, \langle a, b, d \rangle, \langle a, c, e \rangle, \langle a, e, f \rangle, \langle b, c, e \rangle, \langle b, e, f \rangle, \langle c, e, f \rangle\}$.

The deletion of the pair $\langle b, c \rangle$ results in:

Basis: $R = \{\langle a, b \rangle, \langle b, d \rangle, \langle c, e \rangle, \langle e, f \rangle\}$
 Closure: $R^+ = \{\langle a, b \rangle, \langle a, d \rangle, \langle b, d \rangle, \langle c, e \rangle, \langle c, f \rangle, \langle e, f \rangle\}$
 Supports: $\{\langle a, b, d \rangle, \langle c, e, f \rangle\}$.

The respective situations are shown in Figure 7.2 where a pair from the basis is represented as a dotted arrow, and the other pairs of the transitive closure are represented as ordinary arrows.

The incremental computation of a transitive closure is initialized by the function `trans-clos-create` defined as follows:

```
trans-clos-create(addfun, delfun)
TC := new Transitive Closure(
  Basis :=  $\emptyset$ , Closure :=  $\emptyset$ , Supports :=  $\emptyset$ ,
  Addfun := addfun, Delfun := delfun).
```

It creates a new structure containing the five fields *Basis*, *Closure*, *Supports*, *Addfun*, and *Delfun*. The first three are initialized to the empty set. The two arguments `addfun` and `delfun` are stored in the fields *Addfun* and *Delfun*, respectively, and

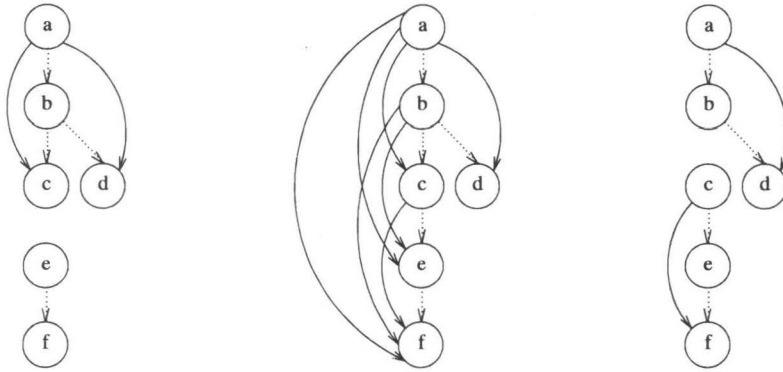


Fig. 7.2. Example of the incremental maintenance of the transitive closure of a relation

will be applied, later on, to each pair which will be added to or deleted from the transitive closure.

The functions

`add-trans-clos(<trans-clos>, <pair>)` and
`del-trans-clos(<trans-clos>, <pair>)`

can be used to add a pair to or delete a pair from the relation R . These functions are symmetrical in their use meaning that if a certain pair is added twice using the function `add-trans-clos`, the function `del-trans-clos` should be used twice to remove it.

The algorithm to add a pair to the relation is the following:

```

add-trans-clos( $TC$ ,  $\langle e_1, e_2 \rangle$ )
if  $\langle e_1, e_2 \rangle \in TC.Basis$ 
then  add  $\langle e_1, e_2 \rangle$  to  $TC.Basis$ 
else  add  $\langle e_1, e_2 \rangle$  to  $TC.Basis$ 
      add  $\langle e_1, e_2 \rangle$  to  $To-be-added$ 
      add-basis( $TC$ ,  $\langle e_1, e_2 \rangle$ )
      for all  $\langle a, b \rangle \in To-be-added$ 
        unless  $\langle a, b \rangle \in TC.Closure$ 
          add-clos( $TC$ ,  $\langle a, b \rangle$ )
          add  $\langle a, b \rangle$  to  $TC.Closure$ 
           $TC.Addfun(\langle a, b \rangle)$ .
```

The algorithm checks whether the new pair $\langle e_1, e_2 \rangle$ is already in the basis and adds it to the basis immediately (the structure $TC.Basis$ remembers how often a pair is added to it). If the new pair was not in the basis, it is added to the list of pairs

To-be-added containing all pairs which should be added to the transitive closure. We aim at the situation where all combinations of pairs from *TC.Closure* with pairs from *TC.Basis* are in *TC.Supports*, and for all triples $\langle a_1, a_2, a_3 \rangle$ in *TC.Supports* the pair $\langle a_1, a_3 \rangle$ is either in *TC.Closure* or in *To-be-added*. Pairs in *To-be-added* can already be in *TC.Closure*. The function *add-basis* is used to update *TC.Supports* and *To-be-added* using the new pair in the basis $\langle e_1, e_2 \rangle$.

```

add-basis(TC,  $\langle e_1, e_2 \rangle$ )
for all  $\langle a, e_1 \rangle \in \textit{TC.Closure}$ 
  add  $\langle a, e_1, e_2 \rangle$  to TC.Supports
  add  $\langle a, e_2 \rangle$  to To-be-added

```

In the next step of *add-trans-clos*, each of the pairs in the list *To-be-added* is handled. Unless such a pair $\langle a, b \rangle$ is already in the closure, the function *add-clos* is used to search the basis to create combinations in *TC.Supports*. Note that this function adds new pairs to *To-be-added* which also have to be treated in the for-loop of *add-trans-clos*. The test whether $\langle a, b \rangle$ is in the closure not only prevents duplication of work, but it also assures termination of the algorithm whenever a cycle occurs in the transitive closure of a relation.

```

add-clos(TC,  $\langle a, b \rangle$ )
for all  $\langle b, c \rangle \in \textit{TC.Basis}$ 
  add  $\langle a, b, c \rangle$  to TC.Supports
  add  $\langle a, c \rangle$  to To-be-added

```

Finally, the pair $\langle a, b \rangle$ is added to the closure and the function *TC.Addfun* is applied to it.

The algorithm to delete a pair $\langle e_1, e_2 \rangle$ is more or less symmetrical to the above algorithm to add a pair.

```

del-trans-clos(TC,  $\langle e_1, e_2 \rangle$ )
delete  $\langle e_1, e_2 \rangle$  from TC.Basis
unless  $\langle e_1, e_2 \rangle \in \textit{TC.Basis}$ 
  add  $\langle e_1, e_2 \rangle$  to To-be-removed
  del-basis(TC,  $\langle e_1, e_2 \rangle$ )
  for all  $\langle a, b \rangle \in \textit{To-be-removed}$ 
    when  $\langle a, b \rangle \in \textit{TC.Closure} \wedge \neg \textit{supported}(\textit{TC}, \langle a, b \rangle)$ 
      del-clos(TC,  $\langle a, b \rangle$ )
      delete  $\langle a, b \rangle$  from TC.Closure
      TC.Delfun( $\langle a, b \rangle$ )

```

The algorithm deletes the new pair once from the basis *TC.Basis*. Unless there is still a pair $\langle e_1, e_2 \rangle$ in the basis, it is added to the list *To-be-removed*. This list

contains the pairs which are potential candidates for removal from the transitive closure. They will only be removed whenever there is no other reason for such a pair to be in the transitive closure. As in case of addition of pairs, *TC.Supports* is updated such that it contains all combinations of pairs from *TC.Closure* with pairs from *TC.Basis*. Whenever a triple is removed from *TC.Supports* the pair consisting of its start and end is added to *To-be-removed*. The function *del-basis* restores this situation by processing the deletion of $\langle e_1, e_2 \rangle$.

```
del-basis(TC,  $\langle e_1, e_2 \rangle$ )
for all  $\langle a, e_1, e_2 \rangle \in \textit{TC.Supports}$ 
    add  $\langle a, e_2 \rangle$  to To-be-removed
    delete  $\langle a, e_1, e_2 \rangle$  from TC.Supports
```

In the next step of *del-trans-clos*, each of the pairs $\langle a, b \rangle$ in the list *To-be-removed* is processed. If such a pair is still in the closure and has no other support, it is removed using the function *del-clos*, it is removed from *TC.Closure*, and the function *TC.Delfun* is applied to it. The algorithm for *del-clos* is the following:

```
del-clos(TC,  $\langle a, b \rangle$ )
for all  $\langle a, b, c \rangle \in \textit{TC.Supports}$ 
    add  $\langle a, c \rangle$  to To-be-removed
    delete  $\langle a, b, c \rangle$  from TC.Supports.
```

The function which checks whether a pair $\langle a, b \rangle$ is still supported for being part of the closure is

```
supported(TC,  $\langle a, b \rangle$ )
 $\langle a, b \rangle \in \textit{TC.Basis} \vee \exists \langle a, c, b \rangle \in \textit{TC.Supports}$ .
```

The algorithms as given here are inspired by the algorithms which Yellin presents in [Yel88]. He uses a directed acyclic graph (the so-called support graph) to store the information which I store in the basis and the supports. The latter two structures are less complex than the structure of the support graph. His description of the algorithm to add a pair to the relation is more complex than the algorithm in my presentation. Instead of my list of pairs *To-be-added*, he uses two lists called *addToClosure* and *newEdges*. He presents two algorithms to delete a pair from the relation. The first method marks the part of the closure and supports which are still valid and implicitly garbage collects the remainder. The second method is similar to the algorithm presented here. He furthermore proves correctness of his algorithms and analyses their complexity.

As mentioned in the introduction to this section, the algorithm does not work correctly when cycles occur in the transitive closure. To be more precise, it fails to

remove a pair $\langle a, b \rangle$ for which b is an element of a cycle in the transitive closure, i.e., $\langle b, b \rangle \in R^+$. It is impossible to repair this without changing the structures updated in the algorithm. There is a plan to re-implement the algorithm such that it can handle cycles.

The incremental transitive closure algorithm is used three times in the ASF+SDF system (see Chapter 5 and Section 7.4). Once in the module manager to maintain the transitive closure of the import graph (see next section), and twice in the syntax manager (see Section 5.4.1) to update the transitive closure of priority declarations (see Section 1.3) and of chain rules (see Section 5.3.2).

7.3.2 Module manager

Having at our disposal an algorithm for the incremental maintenance of transitive closures, the algorithms needed to implement the functions of the module manager as already described in Section 7.2.2 can now be described:

```
modman-create() → <modman>,
add-module(<modman>, <module>),
del-module(<modman>, <module>),
add-import(<modman>, <module name>, <import>),
del-import(<modman>, <module name>, <import>),
add-item(<modman>, <module name>, <item>),
del-item(<modman>, <module name>, <item>), and
apply(<modman>, <module name>, [<parameters>]).
```

In this description the functions of the item managers as described in Section 7.2.1 are used:

```
itemman-create() → <itemman>,
add-item(<itemman>, <item>) → <handle>,
del-item(<itemman>, <handle>),
new-selection(<itemman>) → <selection>,
del-selection(<itemman>, <selection>),
enable(<itemman>, <selection>, <handle>),
disable(<itemman>, <selection>, <handle>), and
apply(<itemman>, <selection>, [<parameters>]).
```

In the following description only one item manager is attached to the module manager. The extension to more than one item manager is easy.

A new module manager is created by the function `modman-create` which is defined as follows:

```
modman-create()
MM := new Module Manager(
    Modules :=  $\emptyset$ ,
```

```

ImpGraph := trans-clos-create(add-imp, del-imp),
Itemman := itemman-create).

```

It creates a new structure containing an empty list of modules (*Modules*), an import graph (*ImpGraph*) which results from the initialization of the transitive closure algorithm as described in the previous section, and an item manager (*Itemman*).

The functions *add-imp* and *del-imp* which are the parameters of *trans-clos-create* in the initialization of the import graph, define what has to be done whenever a module is (directly or indirectly) imported in another one. These functions are defined as follows:

```

add-imp(MM, <Modinfo, Impinfo>)
  for all Item ∈ Impinfo.Items
    do enable(MM.Itemman, Modinfo.Selection, Item.Handle)

```

and

```

del-imp(MM, <Modinfo, Impinfo>)
  for all Item ∈ Impinfo.Items
    do disable(MM.Itemman, Modinfo.Selection, Item.Handle).

```

If a new pair consisting of an importing module described by *Modinfo* and an imported module described by *Impinfo* is added to the import graph, all items of *Impinfo* are enabled in the selection of *Modinfo*. The reverse is done whenever such a pair is deleted.

Whenever a new module $\mathcal{M} \equiv \langle \text{Name}(\mathcal{M}), \text{Imp}(\mathcal{M}), \text{Items}(\mathcal{M}) \rangle$ (for definition see Section 6.2.2) is added, the function *add-module* first creates a structure containing the information of the module. It contains the name of the module, the items as defined in the module, and the selection by which this module is known to the item manager. This structure is then added to *MM.Modules*. Next, the imports and the items of the module are added using the respective functions *add-import* and *add-item*. This results in the following algorithm:

```

add-module(MM,  $\mathcal{M}$ )
  Modinfo := new Module Information(
    Name := Name( $\mathcal{M}$ ),
    Items :=  $\emptyset$ ,
    Selection := new-selection(MM.Itemman))
  add Modinfo to MM.Modules
  for all Imp ∈ Imp( $\mathcal{M}$ ) do add-import(MM, Name( $\mathcal{M}$ ), Imp)
  for all Item ∈ Items( $\mathcal{M}$ ) do add-item(MM, Name( $\mathcal{M}$ ), Item).

```

The function *del-module* is the exact reverse of the above algorithm:

```

del-module(MM, M)
for all Item ∈ Items(M) do del-item(MM, Name(M), Item)
for all Imp ∈ Imp(M) do del-import(MM, Name(M), Imp)
Modinfo := search Name(M) in MM.Modules
delete Modinfo from MM.Modules
del-selection(MM.Itemman, Modinfo.Selection).

```

First, the items and imports of the module are removed using the functions *del-item* and *del-import*. Next, the information of the module is searched in *MM.Modules*, and this information is then removed. Finally, the removal of the corresponding selection is made known to the item manager.

The addition and deletion of an import are implemented easily by adding or deleting the pair *<Modinfo, Impinfo>*, consisting of the importing module and the imported module to or from the import graph. This gives:

```

add-import(MM, Name, Imp)
Modinfo := search Name in MM.Modules
Impinfo := search Imp in MM.Modules
add-trans-clos(MM.ImpGraph, <Modinfo, Impinfo>)

```

and

```

del-import(MM, Name, Imp)
Modinfo := search Name in MM.Modules
Impinfo := search Imp in MM.Modules
del-trans-clos(MM.ImpGraph, <Modinfo, Impinfo>).

```

The algorithm to add an item is the following:

```

add-item(MM, Name, Item)
Modinfo := search Name in MM.Modules
Iteminfo := new Item Information(
    Item := Item,
    Handle := add-item(MM.Itemman, Item))
add Iteminfo to Modinfo.Items
enable(MM.Itemman, Modinfo.Selection, Iteminfo.Handle)
for all <Modinfo', Modinfo> ∈ MM.ImpGraph
    when Modinfo' ≠ Modinfo
        do enable(MM.Itemman, Modinfo'.Selection, Iteminfo.Handle).

```

After searching the information of the module in *MM.Modules*, it creates a structure in which the new item and its handle as returned by the item manager are stored. Next, that structure is stored in *Modinfo.Items*. Finally, the handle of the new item

is enabled in all selections of modules in which the item is known. First, the handle is enabled in the selection of the module in which the item is defined. Next, it is enabled in the selections of modules in which that module is either directly or indirectly imported. The condition which tests equality of *Modinfo* and *Modinfo'* in the for-loop is necessary to prevent enabling the handle twice in case the module is imported in itself.

The algorithm to delete an item is the exact reverse of the above algorithm to add an item:

```
del-item(MM, Name, Item)
Modinfo := search Name in MM.Modules
Iteminfo := search Item in Modinfo.Items
for all <Modinfo', Modinfo> ∈ MM.ImpGraph
  when Modinfo' ≠ Modinfo
    do disable(MM.Itemman, Modinfo'.Selection, Iteminfo.Handle)
  disable(MM.Itemman, Modinfo.Selection, Iteminfo.Handle)
del-item(MM.Itemman, Iteminfo.Handle)
delete Iteminfo from Modinfo.Items.
```

In this version of *del-item*, all occurrences of an item in selections in which it is enabled are disabled before removing it. If we would be more sloppy in disabling items, the algorithm would become:

```
del-item(MM, Name, Item)
Modinfo := search Name in MM.Modules
Iteminfo := search Item in Modinfo.Items
del-item(MM.Itemman, Iteminfo.Handle)
delete Iteminfo from Modinfo.Items.
```

Probably, the latter algorithm will be more efficient as less calls to the item manager are needed. It is, however, more dangerous because the item manager should delete an item which might still be enabled in several selections.

The description of the module manager is concluded with the function which applies the implementation generated for the normal form of a module to given parameter values:

```
apply(MM, Name, Params)
Modinfo := search Name in MM.Modules
apply(MM.Itemman, Modinfo.Selection, Params).
```

It simply calls the function *apply* of the item manager with the appropriate selection.

One can easily verify that the module manager as described here has the following properties:

- The set of items appearing in the selection as calculated for a module by the module manager is identical to the set of items appearing in the normal form of that module obtained by textual expansion.
- Apart from the discussion on the necessity to disable all occurrences of an item before removing it, the module manager produces the minimum amount of calls to the item manager.

7.3.3 Difference analysis algorithm - DAA

How can we now establish a link between the operations provided by the module manager and the interactive editing of specifications? The primary problem to be solved is how changes made to a specification during editing can be translated into the fixed set of add and delete operations provided by the module manager.

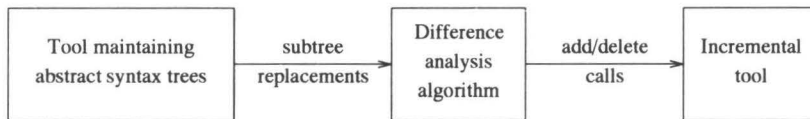


Fig. 7.3. Architecture of a system using DAA

This section describes a difference analysis algorithm (DAA) that solves the above problem in a generic way. The algorithm analyses differences between old and new abstract syntax trees and translates those into appropriate calls to an incremental tool (see Figure 7.3).

If a mutation in the abstract syntax tree is detected, DAA has three functions to process it:

```

create(<tables>, <new>),
destroy(<tables>, <old>), and
change(<tables>, <context>, <old>, <new>).

```

The function `create` is used whenever a tree is created from scratch. It is, for example, used after the first successful parse of the text in the editor. Its counterpart is the function `destroy` which can be used when the editor is left containing a text which cannot be parsed. It removes all information the incremental tool had about the tree. The main function of the algorithm is the function `change` which is called for all other mutations. The arguments `<old>` and `<new>` contain (pointers to) respectively the old and new subtree. The `<context>` contains the position in the tree in which the mutation is found (see Figure 7.4). The argument `<tables>`

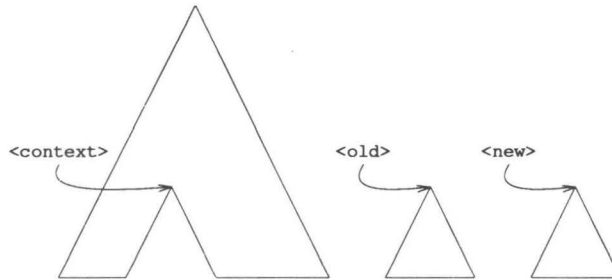


Fig. 7.4. Arguments of function change

contains three tables which are used by the difference analysis algorithm to process mutations.

The difference analysis algorithm is now described in three stages. First, the algorithm is presented in its simplest form. Then, two optimizations are introduced. In each stage a new table is needed to guide the algorithm.

7.3.3.1 DAA with grains

In the first version of the algorithm, it is assumed that the functionality of the incremental tool for which calls have to be generated can be described in terms of additions and deletions of subtrees. In many cases, however, it will not have at its disposal add and delete functions for all possible operators that may appear at the root of a subtree. Some subtrees are too small to be processed. Hence, the existence of a table containing all operators that can be handled by the incremental tool is assumed. This table determines the grain-size with which the incremental tool processes mutations and is therefore called the *table of grains*. Each grain consists of the following components:

- an operator,
- an acceptance test on the context of subtrees with that operator at their root,
- the functions *Addfun* to add and
- *Delfun* to delete a subtree, and
- the function *Argfun* which can be applied to the context of the subtree to extract extra information from it.

The acceptance test and the function *Argfun* are optional.

The grain is only applied to the subtree of the mutation if the acceptance test is either absent or if the context of the mutation passes the test successfully. The acceptance test gives extra flexibility as it is now possible to handle situations in which different functions of the incremental tool have to be applied depending on the context of the same operator.

As addition and deletion are symmetrical operations, it is assumed that the incremental tool provides add and delete functions in pairs. The extra information which both functions need from the context of the mutated subtree should always be identical.

With these prerequisites the first version of the difference analysis algorithm is as follows:

```

change(Tbl, Context, Old, New)
unless Old = New
  OldGrain := search grain of Old.Op with Context in Tbl.Grains
  NewGrain := search grain of New.Op with Context in Tbl.Grains
  if OldGrain is found  $\wedge$  NewGrain is found
  then  OldGrain.Delfun(Old, OldGrain.Argfun(Context))
        NewGrain.Addfun(New, NewGrain.Argfun(Context))
  else  change(Tbl, Context.Up, Old.Up, New.Up).

```

First, it checks whether both trees *Old* and *New* are identical as further processing is only useful if both trees are different. Then, a search is made for the grains corresponding to the top operator of respectively *Old* and *New* in the table of grains *Tbl.Grains*. In this search, the context of the mutation *Context* is needed in case an acceptance test of some grain has to be applied to it. As *Old.Op* and *New.Op* might be different, it is possible that different grains will be found if grains can be found for both. If both grains have been found, they are applied: *Old* is removed using the delete function *Delfun* and the argument function *Argfun* from the corresponding grain *Oldgrain*, and *New* is added using its grain. If neither of the grains can be found, the change function is applied recursively to the parents of the modified subtrees.

The latter situation is the only case in which the algorithm can possibly fail. It is possible that ascent in a given tree is impossible. In other words: the whole tree is changed and no grain could be found for either the old tree or the new tree of the mutation. To preclude this situation, all operators that may occur at the top are required to have an entry in the table of grains. In fact, the incremental tool is used in a non-incremental way.

The difference analysis algorithm can be used perfectly to connect a *non-incremental* tool. In that case, the table of grains will only contain entries for all possible top operators. Each call to DAA results in a walk through both trees until the top is reached. The entire old tree is then removed and the new one is added.

7.3.3.2 DAA with grains for list operators

The above version of DAA is not optimal if used for abstract syntax trees containing *variadic* operators (also called *list operators*, see Section 3.4). It often happens that

an incremental tool can process the children of a list operator independently. In that case, a permutation of the children of such a node should not result in any processing of the incremental tool. In the above version of the algorithm, an addition or deletion of some of the children of a list operator can only result in removing all old children followed by adding all new ones.

The next version of DAA repairs this deficit. Now, an extra table *ListGrains* is needed in which the *grains for list operators* are specified. Each entry in this table is a grain consisting of the same five components as in the previous version. The differences between *Grains* and *ListGrains* are that the operators in the latter should be list operators and the functions *Addfun* and *Delfun* will be applied to the children of the varyadic node. This gives the following algorithm:

```

change(Tbl, Context, Old, New)
unless Old = New
  ListGrain := search grain of Old.Op with Context in Tbl.ListGrains
  if Old.Op = New.Op  $\wedge$  ListGrain is found
  then   for all Child  $\in$  Old.Children - New.Children
          ListGrain.Delfun(Child, ListGrain.Argfun(Context))
          for all Child  $\in$  New.Children - Old.Children
          ListGrain.Addfun(Child, ListGrain.Argfun(Context))
  else   OldGrain := search grain of Old.Op with Context in Tbl.Grains
          NewGrain := search grain of New.Op with Context in Tbl.Grains
          if OldGrain is found  $\wedge$  NewGrain is found
          then   OldGrain.Delfun(Old, OldGrain.Argfun(Context))
                  NewGrain.Addfun(New, NewGrain.Argfun(Context))
          else   change(Tbl, Context.Up, Old.Up, New.Up).

```

In case the top operator of the mutation is a list operator which is not modified, a grain must be looked for first in *Tbl.ListGrains* before searching in *Tbl.Grains*. If such a grain can be found, *Delfun* is applied to the children which were in the old tree and which cannot be found in the new one. *Addfun* is applied to all new children. In the implementation of the algorithm, care is taken that multiple occurrences of identical subtrees are handled correctly. *Old.Children* and *New.Children* should be envisaged as multisets of trees. If a tree occurs twice as child in the old tree and only once in the new tree, it is still necessary to generate one call of the delete function.

7.3.3.3 DAA for operators with independent children

Normally, the incremental tool will process a small modification faster than a big one. Consequently, it will pay to look for identical subtrees in the modified tree. In the final version of the algorithm, a start is made by checking whether the top

operator of the modification is changed. If it is not changed, the algorithm is applied to the respective children of *Old* and *New*. The latter, however, is useless if the incremental tool cannot process those children independently or if any of the children cannot be processed. An extra table *IndepChildren* (called the *table of operators with independent children*) is needed to describe these properties. It contains all operators whose children can be processed independently by the incremental tool. An acceptance test which will be applied to both trees *Old* and *New* can be added to each operator. The algorithm will only descend in the trees if

- both top operators are identical,
- both trees have the same number of children (which is important in case of list operators),
- There is an entry in the table of operators with independent children whose operator equals the top operator of the mutation, and either there is no acceptance test or both trees pass the test with success.

The final version of the change function now becomes:

```

change(Tbl, Context, Old, New)
unless Old = New
  if Old.Op = New.Op  $\wedge$ 
    Old.NrChild = New.NrChild  $\wedge$ 
    entry of Old.Op with Old and New in Tbl.IndepChildren is found
  then  for  $1 \leq i \leq \text{Old.NrChild}$ 
        change(Tbl, Context.Childi, Old.Childi, New.Childi)
  else  ListGrain := search grain of Old.Op with Context in Tbl.ListGrains
        if Old.Op = New.Op  $\wedge$  ListGrain is found
        then  for all Child  $\in$  Old.Children - New.Children
              ListGrain.Delfun(Child, ListGrain.Argfun(Context))
              for all Child  $\in$  New.Children - Old.Children
              ListGrain.Addfun(Child, ListGrain.Argfun(Context))
        else  OldGrain := search grain of Old.Op with Context in Tbl.Grains
              NewGrain := search grain of New.Op with Context in Tbl.Grains
              if OldGrain is found  $\wedge$  NewGrain is found
              then  OldGrain.Delfun(Old, OldGrain.Argfun(Context))
                    NewGrain.Addfun(New, NewGrain.Argfun(Context))
              else  change(Tbl, Context.Up, Old.Up, New.Up).
```

Hitherto, the algorithms of the functions `create` and `destroy` have not been presented. Their definition can be derived as special cases from the definition of `change`:

```

create(Tbl, New)
ListGrain := search grain of New.Op with New in Tbl.ListGrains
if ListGrain is found
then   for all Child ∈ New.Children
        ListGrain.Addfun(Child, ListGrain.Argfun(New))
else   NewGrain := search grain of New.Op with New in Tbl.Grains
        if NewGrain is found
        then NewGrain.Addfun(New, NewGrain.Argfun(New))
        else print error message

and

destroy(Tbl, Old)
ListGrain := search grain of Old.Op with Old in Tbl.ListGrains
if ListGrain is found
then   for all Child ∈ Old.Children
        ListGrain.Delfun(Child, ListGrain.Argfun(Old))
else   OldGrain := search grain of Old.Op with Old in Tbl.Grains
        if OldGrain is found
        then OldGrain.Delfun(Old, OldGrain.Argfun(Old))
        else print error message.

```

In both functions, a grain must be searched for in the table of grains for list operators. If it can be found then the children of the list operator are added (respectively deleted); if it cannot be found, a grain must be searched for in the table of grains. As mentioned before, this search has to succeed and the grain found is applied to the complete tree.

The difference analysis algorithm works perfectly well if the table of operators with independent children is empty. Each entry in this table just optimizes the handling of mutations in which parts of the mutated tree did not change.

Note that this last version of DAA might loop if the contents of the tables are not coherent. This happens if the table of operators with independent children contains an operator whose children are not handled appropriately either by inspecting the table of grains or the table of grains for list operators. The algorithm does not loop if, for all operators in the table of operators with independent children and for all possible subtrees of that operator, an entry can be found either in the table of grains or the table of grains for list operators. This has to be checked dynamically.

The above version of the algorithm is not optimal in the case of list operators. On the one hand, if the table of operators with independent children contains an entry of a list operator, DAA only descends if its number of children is not changed. In that case, a permutation of the children can result in several calls to the

incremental tool. The latter would not happen if no entry is defined in the table of operators with independent children. On the other hand, if no entry is defined for a list operator in the table of operators with independent children, DAA never descends to its children. Hence, if only a small modification which can be handled by the incremental tool is made in any of the children, the whole child is replaced.

7.3.3.4 Applications of DAA

In this section, two applications of the difference analysis algorithm are shown. To show the use of DAA in the incremental treatment of a standard programming language, Pico is handled first. It is a toy programming language whose algebraic specification is given in [BHK89a]. Application of DAA in this particular example gives that each modification in the abstract syntax tree of a Pico program is translated by DAA into appropriate calls to an incremental tool (an incremental typechecker, for example).

Consider the following specification of the grammar of Pico written in ASF+SDF.

```

module Pico-syntax
imports Identifiers Types Expressions
exports
  sorts Program Decls DeclList Decl StatList Stat
  context-free syntax
    begin Decls StatList end -> Program      % prog-op      %
    declare DeclList ";" -> Decls            % decls-op      %
    {Decl ","}* -> DeclList                  % decllist-op %
    Id ":" Type -> Decl                      % decl-op      %
    {Stat ";"}* -> StatList                  % statlist-op %
    Id "!=" Exp -> Stat                      % assign-op    %
    if Exp then StatList else StatList fi -> Stat      % if-op        %
    while Exp do StatList od -> Stat          % while-op     %

```

For convenience of description, names are given to each operator in the abstract syntax corresponding to this grammar. These names are given in the comment behind each rule in the context-free syntax section. In this definition the list operators which would normally be generated automatically are defined explicitly.

Suppose the incremental tool which handles Pico has functions to add or delete programs, declarations, or statements:

```

add-prog(<program>),
del-prog(<program>),
add-decl(<declaration>),
del-decl(<declaration>),

```

add-stat(<statement>), and
del-stat(<statement>).

It is assumed that the incremental tool is capable of handling mutations of statements within the then- and else-part of an if-op, and in the do-od-part of while-op. It is furthermore assumed that no other changes can be handled by the incremental tool.

Figures 7.5 and 7.6 respectively give the tables of grains and grains for list operators in these circumstances.

| operator | test | Addfun | Delfun | Argfun |
|-----------|------|----------|----------|--------|
| prog-op | | add-prog | del-prog | |
| decl-op | | add-decl | del-decl | |
| assign-op | | add-stat | del-stat | |
| if-op | | add-stat | del-stat | |
| while-op | | add-stat | del-stat | |

Fig. 7.5. Grains in case of Pico

| operator | test | Addfun | Delfun | Argfun |
|-------------|------|----------|----------|--------|
| decllist-op | | add-decl | del-decl | |
| statlist-op | | add-stat | del-stat | |

Fig. 7.6. Grains of list operators in case of Pico

The entry for prog-op in the table of grains is the only entry which is absolutely necessary. DAA would fail to handle the initial creation of a Pico program if that entry was missing.

Note that no entry for decls-op is given in the table of grains. Consequently, if a mutation is processed whose top operator is decls-op (which can only happen if decls-op is the top operator of <old> as well as <new>), DAA ascends and removes the complete old Pico program and replaces it by the new one. The latter is prevented by adding decls-op to the table of operators with independent children as shown in Figure 7.7.

| operator | test |
|-------------|---------------------|
| prog-op | |
| decls-op | |
| decllist-op | |
| statlist-op | |
| if-op | $Old.D_1 = New.D_1$ |
| while-op | $Old.D_1 = New.D_1$ |

Fig. 7.7. Operators with independent children in case of Pico

The last two entries in this table take care that mutations in the list of statements occurring in the then- and else-part of if-op, and in the do-od-part of while-op will be handled without removing the complete if-then-else-fi and while-do-od. This is of course only possible if the type of statement and the expression of the statement are not changed. The latter is expressed by the acceptance test $Old.D_1 = New.D_1$ in the entries for if-op and while-op in Figure 7.7. The expression $Old.D_1$ represents the first child (the expression) of the tree Old .

As a second application of DAA, its use in handling the parameters as defined in the syntax part of ASF+SDF (see Section 5.3.1) is demonstrated. Consider the following ASF+SDF specification of the grammar of parameters.

```

module Parameters

imports Identifiers Functions

exports
  sorts
    ParmList Parameter SectionList Section
    SortList FunctionList
  context-free syntax
    Parameter*          -> ParmList      % parmlist-op %
    "parameter" Id SectionList
                                -> Parameter % parm-op      %
    Section*            -> SectionList % seclist-op %
    "sorts" SortList     -> Section      % sorts-op      %
    "functions" FunctionList -> Section  % fncls-op      %
    Id+                 -> SortList     % sortlist-op %
    Function+           -> FunctionList % fnclist-op %

```

Suppose the incremental tool which has to handle parameters has functions to add or delete parameters, sections, sorts, or functions:

```

add-parm(<parameter>),
del-parm(<parameter>),
add-section(<section>, <parameter name>),
del-section(<section>, <parameter name>),
add-sort(<sort>, <parameter name>),
del-sort(<sort>, <parameter name>),
add-function(<function>, <parameter name>), and
del-function(<function>, <parameter name>).

```

All functions except the functions to add/delete complete parameters (add-parm and del-parm) need as extra argument the name of the parameter <parameter name>. Mutations in parts of a function (the elements of Function) cannot be handled by the incremental tool.

The tables of grains and grains for list operators are shown in Figures 7.8 and 7.9 respectively.

| operator | test | Addfun | Delfun | Argfun |
|----------|---------------------------------|--------------|--------------|---|
| parm-op | <i>Context.Up</i> = sortlist-op | add-parm | del-parm | |
| sorts-op | | add-section | del-section | <i>Context.Up₂.D₁</i> |
| fncs-op | | add-section | del-section | <i>Context.Up₂.D₁</i> |
| id-op | | add-sort | del-sort | <i>Context.Up₄.D₁</i> |
| fnc-op | | add-function | del-function | <i>Context.Up₄.D₁</i> |

Fig. 7.8. Grains in case of parameters

| operator | test | Addfun | Delfun | Argfun |
|-------------|------|--------------|--------------|---|
| parmlist-op | | add-parm | del-parm | |
| seclist-op | | add-section | del-section | <i>Context.Up.D₁</i> |
| sortlist-op | | add-sort | del-sort | <i>Context.Up₃.D₁</i> |
| fnclist-op | | add-function | del-function | <i>Context.Up₃.D₁</i> |

Fig. 7.9. Grains of list operators in case of parameters

In the entry *Argfun* the meaning of *Context.Up₂.D₁* is that the extra argument needed for handling the grain is found by ascending two steps (*Up₂*) followed by descending to the first child (*D₁*). If the entry of *Argfun* is empty, no extra argument is needed by *Addfun* and *Delfun*.

The only entry in these tables which is absolutely needed is the entry of *parmlist-op* in the table of grains for list operators as it is the top operator of the language. Without that entry, the algorithm would fail and a mutation in the list of parameters (an addition of a new parameter for example) would give an error. No appropriate grain could then be found by the algorithm and hence it would try to ascend. The latter, however, is impossible in this case. If the grain for *parmlist-op* only is given, the incremental tool which handles the parameters is used in a non-incremental way.

In the above grammar, identifiers which are defined with their operator *id-op* in the imported module *Identifiers*, are used in several rules. They occur in the definitions of parameters (the name of the parameter in operator *parm-op*), sorts (*sortlist-op*), and functions (*fnc-op* in module *Functions*). There is only one entry in the table of grains for *id-op*. The test in this grain prevents it to be used if its father is not *sortlist-op*. Consequently, a change in the name of a parameter results in ascending to its father *parm-op*. As a grain is defined for *parm-op*, the old parameter is removed completely followed by adding the parameter with its new name. A change of an identifier in a function results in ascending until the operator of the tree equals *fnc-op*. The old function is then removed and replaced by the

new one. The only mutation of an identifier which is handled by DAA without ascending is the change of the name of a sort. There is only one entry in the table of grains for `id-op`.

The only situation in this example in which two different grains are needed to handle a mutation is when a `sorts` section is changed into a `functions` section or vice versa.

The table of operators with independent children in case of the above example is shown in Figure 7.10.

| operator | test |
|--------------------------|---------------------|
| <code>parmlist-op</code> | $Old.D_1 = New.D_1$ |
| <code>parm-op</code> | |
| <code>seclist-op</code> | |
| <code>sorts-op</code> | |
| <code>fncl-op</code> | |
| <code>sortlist-op</code> | |
| <code>fnclist-op</code> | |

Fig. 7.10. Operators with independent children in case of parameters

The test in the entry for operator `parm-op` takes care that DAA will only descend in the case that the operator of a mutation is `parm-op` if the name of the parameter is unchanged. Without this test DAA loops as a change in the name of a parameter results in ascending in the tree (no applicable grain is defined for `id-op` in the table of grains) followed by descending back to the change in the name of the parameter.

7.4 Application: the ASF+SDF system

An instance of the module manager whose functionality and main algorithms have been described in the previous sections is used in the ASF+SDF system (see Chapter 5). It is an incremental system in which specifications written in the combination of ASF [BHK89a] and SDF [HK89b, HHKR89] (see Section 1.4) can be developed and tested. Each module written in ASF+SDF consists of a syntax part and an equations part (see Section 5.2.2). The syntax part contains the syntax rules and the imports defined in that module. The equations part contains the (conditional) equations which use the syntax as defined in the syntax part of the module and in all its imports. The ASF+SDF system contains two item managers:

- The syntax manager SM (see Section 5.4.1) generates a parser for syntax rules as defined in SDF.

- The equation manager EQM (see Section 5.4.2) generates a term rewriting system for the parsed equations of the module.

Figure 7.11 gives the global architecture of the ASF+SDF system.

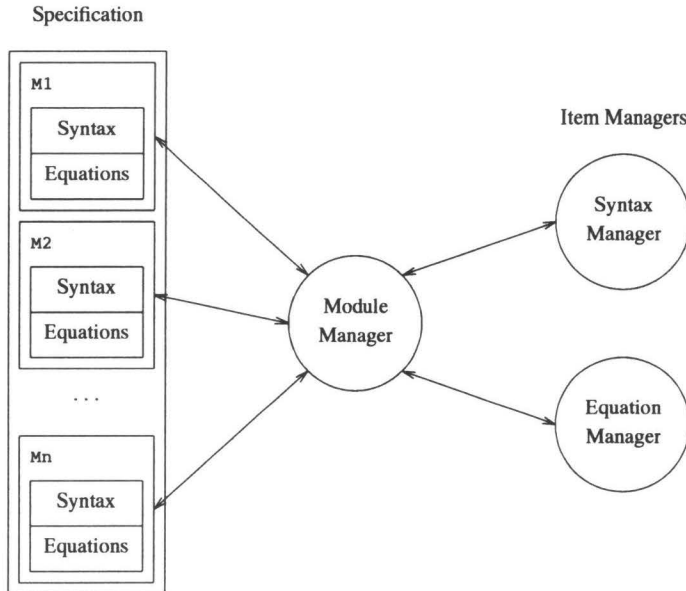


Fig. 7.11. Architecture of the ASF+SDF system

This scheme is an instance of the more general architecture of the implementation of a modular specification formalism as shown in Figure 7.1.

The syntax manager SM is an enhanced version of the implementation of SDF. It consists of a lazy, incremental, and modular parser generator (MPG [Rek89b]) which can handle arbitrary context-free grammars. It generates a table-driven parser based on Tomita's algorithm [Tom85, Rek89a] which returns all possible parse trees of a given text.

In this component a lazy, incremental, and modular scanner generator (MSG [Kli91]) is used to generate a finite automaton from the lexical part of an SDF specification. This automaton is also capable of handling ambiguous regular expressions as it returns all possible interpretations of a given string.

All modifications in the syntax part of a module are translated into appropriate calls to functions of SM which add or delete

- a sort,
- a lexical function,
- a context-free function,

- a variable,
- a relative priority, or
- the associativity of a group.

For each module in the specification, two selections are created by the module manager (MM). One for the purpose of parsing the equations part of a module, and the other one for parsing terms according to the grammar defined by the module. MM enables and disables (see Section 7.2.1) the elements of the syntax which constitute these selections. If text is to be parsed, MM is asked to give the appropriate selection, and SM is called to parse the text using that selection.

The equation manager EQM is an incremental and modular tool which interprets the equations of a specification as rewrite rules (conditions are interpreted as described in Section 2.4.2). Upon evaluation it uses leftmost innermost rewriting modulo lists (see Chapter 3) to rewrite a given term (the representation of the text in a term editor) to its normal form. EQM responds to additions and deletions of equations.

In all three components (the Modular Scanner Generator, the Modular Parser Generator, and the equation manager) modularity has been implemented using selections (see Section 7.2.1).

Index

- abstract syntax 6
- algebraic specification 33
 - ~ with associativity 37
 - ~ with lists 45
- ASF 1
 - ~ system 15
 - combination of ~ and SDF 8
 - conditional equations 1
 - exports 3
 - functions 1
 - hiddens 3
 - implementation 3
 - imports 3
 - initial algebra 2
 - modularization 3
 - natural numbers in ~ 1, 4
 - normalization 4
 - parameters 3
 - renamings 4
 - sets in ~ 5
 - sets of natural numbers in ~ ... 1, 6
 - signature 1
 - sorts 1
 - structure diagrams 4
- ASF+SDF 8
 - ~ system 91
 - Booleans in ~ 62
 - layout in ~ 10, 61
 - modularization 8
 - natural numbers in ~ 11, 63
 - sets in ~ 12, 146
 - sets of natural numbers in ~ 12
 - structure diagrams 4
 - tables in ~ 65
- assignment 34
- associativity 31, 37
 - ~ of groups 115
- axiom operator 139, 142
- basic lexical elements 110
- basis 168
- Booleans 62
 - ~ in ASF+SDF 62
- chain rule 118
- closed terms 2, 33
- closure 168
- complete 24
- compositional semantics 134
- concrete syntax 6
 - ~ of Mini-ML 56
- conditional
 - ~ equational logic 34
 - ~ equations 1, 33, 117
- conditions 117
- confluence 3, 26
- confusion 2
- context-free
 - ~ elements 111
 - ~ functions 111
 - ~ syntax 8
- correct 24, 136
- DAA 176
 - table of grains 177
 - table of operators with independent children ... 180
 - table with grains for list operators

| | | |
|--|--------------------|---|
| ... 179 | hiddens | 3 |
| defined functions | 21 | |
| diagrams | 4 | |
| difference analysis algorithm | 176 | |
| disable | 165 | |
| enable | 165 | |
| EQM | 118, 121, 188 | |
| equation manager | 118, 121, 188 | |
| equational logic | 34 | |
| conditional ~ | 34 | |
| equations | | |
| conditional ~ | 1, 33, 117 | |
| unconditional ~ | 33, 117 | |
| exports | 3 | |
| extended signature | 45 | |
| extended theory semantics | 139 | |
| axiom operator | 139 | |
| proof relation | 139 | |
| semantic domain | 139 | |
| semantics | 139 | |
| flattened terms | 38 | |
| focus | 96 | |
| formulae | 136 | |
| functions | 1, 33 | |
| context-free ~ | 111 | |
| defined ~ | 21 | |
| lexical ~ | 110 | |
| generalization of types | 58, 75 | |
| generalized type | 58 | |
| generic syntax-directed editor ... | 96, | |
| 118, 122 | | |
| generic type variable | 58 | |
| grains | 177 | |
| ~ for list operators | 179 | |
| GSE | 96, 118, 122 | |
| implementation | | |
| ~ of ASF | 3 | |
| ~ of modular specifications | 163 | |
| import graph | 129, 152 | |
| imports | 3, 128 | |
| incremental | 163 | |
| inequalities | 56 | |
| inference system | 55 | |
| initial algebra | 2 | |
| confusion | 2 | |
| junk | 2 | |
| instantiation of generalized types ... | | |
| 58, 75 | | |
| item manager | 164 | |
| items | 126, 128, 148, 163 | |
| iterated sorts | 31, 45 | |
| junk | 2 | |
| layout | 8 | |
| ~ in ASF+SDF | 10, 61 | |
| left-linear | 26 | |
| lexical | | |
| basic ~ elements | 110 | |
| ~ elements | 110 | |
| ~ functions | 110 | |
| ~ syntax | 8 | |
| lists | 31, 45 | |
| literal characters | 109 | |
| literals | 109 | |
| matching term | 42 | |
| meta-variables | 97 | |
| Mini-ML | 56 | |
| concrete syntax | 56 | |
| generalization of types | 58, 75 | |
| generalized type | 58 | |

- generic type variable 58
- instantiation of generalized types
 - ... 58, 75
- static semantics of \sim 55
- type 57, 67
- type environment 58, 59
- type errors 79
- type variable 57
- typecheck algorithm 59, 80
- typechecking 55, 57, 80
- MM 118, 121, 188
- model class semantics 142
 - axiom operator 142
 - semantic domain 142
 - semantics 143
 - truth relation 142
- modifiers 113
- modular specification 129, 151
 - implementation 163
 - import graph 129, 152
 - modules 150
 - normal form 129
 - normalization 125, 129, 154
 - statically correct 131
 - type errors 154
 - typechecking 131, 156
- modularization 125
 - \sim in ASF 3
 - \sim in ASF+SDF 8
 - textual \sim 125
- module editors 96
- module manager ... 118, 121, 164, 188
- modules 129, 150
- most general type 55, 57
- natural numbers
 - \sim in ASF 1, 4
 - \sim in ASF+SDF 11, 63
 - \sim in SDF 7
- non-ambiguous 26
- non-compositional semantics 134
- non-modular specification formalism
 - ... 128, 149
 - items 128, 148
 - semantic domain 128
 - semantics 128
 - static correctness property 128
 - type errors 148
- normal form
 - \sim in term rewriting 3
 - \sim of a module 129
- normalization 4, 125, 129, 154
- open terms 24
- operators with independent children ...
 - 180
- origin rule 131, 132
- overlapping input types 47
- parameter bindings 113
- parameters 3
- plussed sorts 45
- polymorphism 55
- priorities 8
- priority declaration 8, 67
- proof relation 136, 139
 - correct 136
 - sound 144
- regular 26
- regular sorts 47
- relative priority 114
- renamings 4, 113
- rewriting
 - \sim modulo associativity 32, 38
 - \sim modulo lists 32, 53
- SDF 6

| | | | |
|---|---------------|---|---------------|
| abstract syntax | 6 | supports | 168 |
| combination of ASF and \sim | 8 | SV | 119, 122 |
| concrete syntax | 6 | syntax manager | 118, 120, 187 |
| context-free syntax | 8 | | |
| layout | 8 | table of grains | 177 |
| lexical syntax | 8 | table of operators with independent children ... | 180 |
| natural numbers in \sim | 7 | table with grains for list operators | 179 |
| priorities | 8 | tables | 64 |
| sets of natural numbers in \sim | 7 | \sim in ASF+SDF | 65 |
| sorts | 7 | tag identifiers | 117 |
| variables | 8 | tags | 117 |
| selection | 165 | tautologies | 135 |
| semantic domain | 128 | term editors | 96 |
| semantics | | term rewriting | 3 |
| compositional \sim | 134 | confluence | 3, 26 |
| non-compositional \sim | 134 | left-linear | 26 |
| static \sim of Mini-ML | 55 | non-ambiguous | 26 |
| sets | | normal form | 3 |
| \sim in ASF | 5 | regular | 26 |
| \sim in ASF+SDF | 12, 146 | termination | 3, 25 |
| sets of natural numbers | | terminal skeleton | 9 |
| \sim in ASF | 1, 6 | termination | 3, 25 |
| \sim in ASF+SDF | 12 | terms | 33 |
| \sim in SDF | 7 | closed \sim | 2, 33 |
| signature | 1, 33 | flattened \sim | 38 |
| extended \sim | 45 | matching \sim | 42 |
| simplification ordering | 25 | open \sim | 24 |
| SM | 118, 120, 187 | textual modularization | 125 |
| sorts | 1, 7, 33 | theory | 127, 136 |
| iterated \sim | 31, 45 | theory semantics | 135, 136 |
| plussed \sim | 45 | extended \sim | 139 |
| regular \sim | 47 | formulae | 136 |
| starred \sim | 45 | proof relation | 136 |
| sound | 144 | semantic domain | 136 |
| starred sorts | 45 | semantics | 136 |
| static correctness property | 128 | transitive closure | 167 |
| static semantics of Mini-ML | 55 | basis | 168 |
| structure diagrams | 4 | closure | 168 |
| supervisor | 119, 122 | | |

| | |
|----------------------------------|--------------|
| supports | 168 |
| translated term | 35 |
| truth relation | 142 |
| sound | 144 |
| type | |
| ~ environment | 58, 59 |
| ~ errors | 79, 148, 154 |
| ~ inference | 55 |
| ~ variable | 57 |
| type errors | |
| ~ in a specification formalism | 148 |
| ~ in Mini-ML | 79 |
| ~ in modular specifications | 154 |
| typecheck algorithm | 59, 80 |
| typechecking | |
| ~ of Mini-ML | 55, 57, 80 |
| ~ of modular specifications ... | 131, |
| 156 | |
| types in Mini-ML | 57, 67 |
| unconditional equations | 33, 117 |
| variables | 8 |
| meta-~ | 97 |

References

- [BHK89a] J.A. Bergstra, J. Heering, and P. Klint (eds.), *Algebraic Specification*, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley (1989).
- [BHK89b] J.A. Bergstra, J. Heering, and P. Klint, "The algebraic specification formalism ASF," in: *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, and P. Klint, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 1-66 (1989), Chapter 1.
- [BHK89c] J.A. Bergstra, J. Heering, and P. Klint, "A simple programming language and its implementation," in: *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, and P. Klint, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 67-132 (1989), Chapter 2.
- [BHK90] J.A. Bergstra, J. Heering, and P. Klint, "Module algebra," *Journal of the ACM* **37**(2), pp. 335-372 (1990).
- [BGS88a] H. Bertling, H. Ganzinger, and R. Schäfers, "CEC: a system for conditional equational completion - User's manual (version 1.4)," Technical Report, Universität Dortmund (1988).
- [BGS88b] H. Bertling, H. Ganzinger, and R. Schäfers, "CEC: a system for the completion of conditional equational specifications," in: *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan and J.-P. Jouannaud, Lecture Notes in Computer Science **308**, Springer-Verlag, pp. 249-250 (1988).
- [BCKSV88] M. Bidoit, F. Capy, C. Choppy, S. Kaplan, F. Schlienger, and F. Voisin, "ASSPEGIQUE: an integrated specification environment," in: *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan and J.-P. Jouannaud, Lecture Notes in Computer Science **308**, Springer-Verlag, pp. 251-252 (1988).
- [BW89] L.G. Bouma and H.R. Walters, "Implementing algebraic specifications," in: *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, and P. Klint, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 199-282 (1989), Chapter 5.
- [BG80] R.M. Burstall and J.A. Goguen, "The semantics of Clear, a specification language," in: *Abstract Software Specifications*, ed. D. Bjørner, Lecture Notes in Computer Science **86**, Springer-Verlag, pp. 292-332

- (1980).
- [Car84] L. Cardelli, "Basic polymorphic typechecking," Computing Science Technical Report no. 112, AT&T Bell Laboratories, Murray Hill (1984).
- [CW85] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *Computing Surveys* 17(4), pp. 471-522 (1985).
- [CENT89] *The CENTAUR Documentation - Version 0.9*, INRIA, Sophia-Antipolis (1989).
- [CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn, "A simple applicative language: Mini-ML," in: *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, ACM, pp. 13-27 (1986).
- [CI89a] D. Clément and J. Incerpi, *The Centaur structure editor*, The CENTAUR Documentation - Version 0.9, Volume II - User's Manual, INRIA, Sophia-Antipolis (1989).
- [CI89b] D. Clément and J. Incerpi, *Graphical objects within Centaur*, The CENTAUR Documentation - Version 0.9, Volume II - User's Manual, INRIA, Sophia-Antipolis (1989).
- [CIL89] D. Clément, J. Incerpi, and B. Lang, *The virtual tree processor*, The CENTAUR Documentation - Version 0.9, Volume II - User's Manual, INRIA, Sophia-Antipolis (1989).
- [CACDHGGR88] D. Coleman, P. Arnold, A. Camilleri, C. Dollin, H. Hayes, H. Gilchrist, P. Goldsack, and T. Rush, "The AXIS papers," Technical Report HPL-ISC-TR-88-031, HPL-ISC-TM-87-47, HPL-ISC-TM-88-18, HPL-ISC-TM-88-019, HPL-ISC-TR-88-034, HPL-ISC-TM-88-033, Hewlett-Packard Laboratories, Bristol (1988).
- [DM82] L. Damas and R. Milner, "Principal type-schemes for functional programs," in: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, pp. 207-212 (1982).
- [Der87] N. Dershowitz, "Termination of rewriting," *Journal of Symbolic Computation* 3, pp. 69-115 (1987).
- [DOS88] N. Dershowitz, M. Okada, and G. Sivakumar, "Confluence of conditional rewrite systems," in: *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan and J.-P. Jouannaud, Lecture Notes in Computer Science 308, Springer-Verlag, pp. 31-44 (1988).
- [DF85] D. Detlefs and R. Forgaard, "A procedure for automatically proving the termination of a set of rewrite rules," in: *Proceedings of the First International Conference on Rewriting Techniques and Applications*,

- ed. J.-P. Jouannaud, *Lecture Notes in Computer Science* **202**, Springer-Verlag, pp. 255-270 (1985).
- [Deu91] A. van Deursen, "An algebraic specification for the static semantics of Pascal," Report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1991), to appear.
- [Die86] N.W.P. van Diepen, "A study in algebraic specification: a language with goto-statements," Report CS-R8627, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1986).
- [DK89] M.H.H. van Dijk and J.W.C. Koorn, *Generic syntax editor*, The CENTAUR Documentation - Version 0.9, Volume I - User's Guide, INRIA, Sophia-Antipolis (1989).
- [DK90] M.H.H. van Dijk and J.W.C. Koorn, "GSE, a generic syntax-directed editor," Report CS-R9045, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1990).
- [DE84] K. Drosten and H.-D. Ehrich, "Translating algebraic specifications to Prolog programs," *Informatik-Bericht Nr. 84-08*, Technische Universität Braunschweig (1984).
- [DC90] G.D.P. Dueck and G.V. Cormack, "Modular attribute grammars," *The Computer Journal* **33**(2), pp. 164-172 (1990).
- [EM85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications*, vol. I, *Equations and Initial Semantics*, Springer-Verlag (1985).
- [EY87] M.H. van Emden and K. Yukawa, "Logic programming with equations," *Journal of Logic Programming* **4**, pp. 265-288 (1987).
- [FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," in: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ed. B. Reid, ACM, pp. 52-66 (1985).
- [Gau84] M.-C. Gaudel, *A first introduction to PLUSS, Draft*, Université de Paris-Sud, Orsay (December 1984).
- [GV89] R. van Glabbeek and F. Vaandrager, "Modular specifications in process algebra with curious queues," in: *Algebraic Methods: Theory, Tools and Applications*, ed. M. Wirsing and J.A. Bergstra, *Lecture Notes in Computer Science* **394**, Springer-Verlag, pp. 465-506 (1989).
- [GKKMMW88] J. Goguen, C. Kirchner, H. Kirchner, A. Mégreli, J. Meseguer, and T. Winkler, "An introduction to OBJ3," in: *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan and J.-P. Jouannaud, *Lecture Notes in Computer Science* **308**, Springer-Verlag, pp. 258-263 (1988).
- [GM82] J.A. Goguen and J. Meseguer, "Universal realization, persistent

- interconnection and implementation of abstract modules,” in: *Proceedings of the Ninth International Conference on Automata, Languages and Programming*, Lecture Notes in Computer Science **140**, Springer-Verlag, pp. 265-281 (1982).
- [GMP83] J.A. Goguen, J. Meseguer, and D. Plaisted, “Programming with parameterized abstract objects in OBJ,” in: *Theory and Practice of Software Technology*, ed. D. Ferrari, M. Bolognani, and J. Goguen, North-Holland, pp. 163-193 (1983).
- [HMM86] R. Harper, D. MacQueen, and R. Milner, “Standard ML,” LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, Edinburgh (1986).
- [HMT87] R. Harper, R. Milner, and M. Tofte, “The semantics of Standard ML version 1,” Report ECS-LFCS-87-36, University of Edinburgh, Edinburgh (1987), also published as CSR-244-87.
- [HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers, “The syntax definition formalism SDF - reference manual,” *SIGPLAN Notices* **24**(11), pp. 43-75 (1989).
- [HK89a] J. Heering and P. Klint, “PICO revisited,” in: *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, and P. Klint, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 359-379 (1989), Chapter 9.
- [HK89b] J. Heering and P. Klint, “The syntax definition formalism SDF,” in: *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, and P. Klint, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 283-297 (1989), Chapter 6.
- [HKR87] J. Heering, P. Klint, and J. Rekers, “Incremental generation of lexical scanners,” Report CS-R8761, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1987), to appear in *ACM Transactions on Programming Languages and Systems*.
- [HKR90] J. Heering, P. Klint, and J. Rekers, “Incremental generation of parsers,” *IEEE Transactions on Software Engineering* **16**(12), pp. 1344-1351 (1990).
- [Hen87] P.R.H. Hendriks, “Type-checking Mini-ML: an experience with user-defined syntax in an algebraic specification,” in: *Conference Proceedings of Computing Science in the Netherlands, CSN’87*, SION, pp. 21-38 (1987).
- [Hen88a] P.R.H. Hendriks, “ASF system user’s guide,” Report CS-R8823, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1988).
- [Hen88b] P.R.H. Hendriks, “Automatic implementation of algebraic specifica-

- tions," in: *Conference Proceedings of Computing Science in the Netherlands, CSN'88 1*, SION, pp. 83-94 (1988).
- [Hen89a] P.R.H. Hendriks, "Lists and associative functions in algebraic specifications - semantics and implementation," Report CS-R8908, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1988).
- [Hen89b] P.R.H. Hendriks, "Typechecking Mini-ML," in: *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, and P. Klint, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 299-337 (1989), Chapter 7.
- [HKK89] M. Hermann, C. Kirchner, and H. Kirchner, *Implementations of term rewriting systems*, LORIA-CNSR and LORIA-INRIA, Vandœuvre-lès-Nancy (1989), to appear in *Computer Journal*, British Computer Society.
- [HOD82] C.M. Hoffmann and M.J. O'Donnell, "Programming with equations," *ACM Transactions on Programming Languages and Systems* **4**(1), pp. 83-112 (1982).
- [HU79] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley (1979).
- [JJWW90] P. de Jager, W. Jonker, A. Wammes, and J. Wester, *On the generation of interactive programming environments - A LOTOS case study*, PTT Research Tele-informatics, Groningen (1990), to appear in *Software engineering and its applications*.
- [Joh86] S.C. Johnson, "YACC: yet another compiler-compiler," in: *UNIX Programmer's Supplementary Documents, Volume 1 (PS1)*, Bell Laboratories (1986).
- [JW87] J.-P. Jouannaud and B. Waldmann, "Reductive conditional term rewriting systems," in: *Proceedings of the Third IFIP Working Conference on Formal Description of Programming Concepts*, ed. M. Wirsing, Elsevier, pp. 223-244 (1987).
- [Kah87] G. Kahn, "Natural semantics," in: *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, ed. F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Lecture Notes in Computer Science **247**, Springer-Verlag, pp. 22-39 (1987).
- [Kap87] S. Kaplan, "Simplifying conditional term rewriting systems: unification, termination and confluence," *Journal of Symbolic Computation* **4**, pp. 295-334 (1987).
- [KR78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).
- [KKM88] C. Kirchner, H. Kirchner, and J. Meseguer, "Operational Semantics of

- OBJ-3," in: *Proceedings of the Fifteenth International Conference on Automata, Languages and Programming*, ed. T. Lepistö and A. Salomaa, Lecture Notes in Computer Science **317**, pp. 287-301 (1988).
- [Kli90] P. Klint, "A meta-environment for generating programming environments," Report CS-R9064, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1990), to appear in: *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, ed. J.A. Bergstra and L.M.G. Feijs, Lecture Notes in Computer Science, Springer-Verlag.
- [Kli91] P. Klint, "Scanner generation for modular regular grammars," Report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1991), to appear, preliminary version appeared in: *J.W. de Bakker, 25 Jaar Semantiek, Liber Amicorum*, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, pp. 291-305 (1989).
- [Laa90] W. van der Laan, *A program generator for database applications*, University of Amsterdam (1990), Master thesis.
- [Lisp87] *LeLisp, Version 15.21, le manuel de référence*, INRIA, Rocquencourt (1987).
- [LS86] M.E. Lesk and E. Schmidt, "LEX - A lexical analyzer generator," in: *UNIX Programmer's Supplementary Documents, Volume 1 (PS1)*, Bell Laboratories (1986).
- [Log88] M.H. Logger, "An integrated text and syntax-directed editor," Report CS-R8820, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1988).
- [MG85] J. Meseguer and J.A. Goguen, "Initiality, induction, and computability," in: *Algebraic Methods in Semantics*, ed. M. Nivat and J.C. Reynolds, Cambridge University Press, pp. 459-541 (1985).
- [Meu88] E.A. van der Meulen, "Algebraic specification of a compiler for a language with pointers," Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1988).
- [Meu90] E.A. van der Meulen, "Deriving incremental implementations from algebraic specifications," Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1990).
- [ODo85] M.J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press (1985).
- [PWBBP85] F. Pereira, D. Warren, D. Bowen, L. Byrd, and L. Pereira, *C-Prolog User's Manual, Version 1.5*, SRI International, Menlo Park, California (1985).

- [Rek89a] J. Rekers, "An implementation of SDF," in: *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, and P. Klint, ACM Press Frontier Series, The ACM Press in co-operation with Addison-Wesley, pp. 339-358, Amsterdam (1989), Chapter 8.
- [Rek89b] J. Rekers, "Modular parser generation," Report CS-R8933, Centrum voor Wiskunde en Informatica (CWI), Amsterdam (1989).
- [RT89] T. Reps and T. Teitelbaum, *The Synthesizer Generator: a System for Constructing Language-Based Editors*, Springer-Verlag (1989).
- [RC88] T. Rush and D. Coleman, "Architecture for conditional term rewriting," in: *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, ed. S. Kaplan and J.-P. Jouannaud, Lecture Notes in Computer Science **308**, Springer-Verlag, pp. 266-278 (1988).
- [Rus85] M. Rusinowitch, "Path of subterms ordering and recursive decomposition ordering revisited," in: *Proceedings of the First International Conference on Rewriting Techniques and Applications*, ed. J.-P. Jouannaud, Lecture Notes in Computer Science **202**, Springer-Verlag, pp. 225-240 (1985).
- [Tip91] F. Tip, *The equation debugger*, University of Amsterdam (1991), Master thesis.
- [Tom85] M. Tomita, *Efficient Parsing for Natural Languages*, Kluwer Academic Publishers (1985).
- [Wal89] H.R. Walters, "Hybrid implementations of algebraic specifications," Report P8913, University of Amsterdam (1989).
- [Wal90] H.R. Walters, "Hybrid implementations of algebraic specifications," in: *Proceedings of the Second International Conference on Algebraic and Logic Programming*, ed. H. Kirchner and W. Wechler, Lecture Notes in Computer Science **463**, Springer-Verlag, pp. 40-54 (1990).
- [Wie87] F. Wiedijk, "Termherschrijfsystemen in Prolog," Rapport P8704, University of Amsterdam (1987), in Dutch.
- [Yel88] D. Yellin, "A dynamic transitive closure algorithm," Research Report RC 13535 (#60495) 2/22/88, IBM Research Division, Yorktown Heights, N.Y. (1988).

Samenvatting

Inleiding

Deze Nederlandstalige samenvatting is bedoeld om aan niet-vakgenoten duidelijk te maken wat het onderwerp van dit proefschrift is. De vakgenoten worden verwezen naar de Engelse inleiding (Preface).

De samenvatting bestaat uit twee delen. Eerst wordt uitgelegd wat de titel: "Implementatie van Modulaire Algebraïsche Specificaties" betekent. Daarbij wordt een specificatie als voorbeeld behandeld en wordt het kader beschreven waarbinnen het onderzoek valt dat tot dit proefschrift heeft geleid. Daarna volgt een korte inhoudsbeschrijving van de verschillende hoofdstukken.

Algebraïsch specificeren

In de informatica betekent *specificeren* het zo exact mogelijk beschrijven wat software doet of zou moeten doen. Zo'n specificatie speelt een belangrijke rol bij de totstandkoming of verandering van software. Ze vormt dan het discussiestuk voor ontwerper en eindgebruiker op basis waarvan contracten worden afgesloten. Ze bevat de gegevens voor de programmeur omtrent wat hij/zij dient te produceren. Bij het testen van de software kan ze gebruikt worden om te controleren of de ontwikkelde software voldoet. Tenslotte, is ze van wezenlijk belang als documentatie bij bestaande software.

In welke taal worden specificaties geschreven? In veel gevallen worden specificaties in een natuurlijke taal (Nederlands, Engels) geschreven. Alhoewel natuurlijke taal voor iedereen leesbaar is, is ze soms niet precies genoeg en kan ze aanleiding geven tot meerdere interpretaties. Natuurlijke taal is dus niet geschikt als medium voor de exacte beschrijving van software. Verder heeft natuurlijke taal het nadeel dat deze niet door een computer verwerkt kan worden. De grammatica is dusdanig gecompliceerd dat computers (nog) niet in staat zijn om zinnen te ontleden. Laat staan dat ze kunnen "begrijpen" wat de zinnen betekenen.

Verschillende specificatieformalismen zijn ontwikkeld om de genoemde tekortkomingen op te heffen. Zo'n specificatieformalisme heeft een dusdanige grammatica dat deze wel door computers verwerkt kan worden. Verder hebben deze formalismen een wiskundige basis zodat de betekenis van een specificatie eenduidig is. Vaak zijn er computerprogramma's ontwikkeld die ondersteuning leveren bij het controleren op tegenstrijdigheden, het testen, of het bewijzen van bepaalde eigenschappen van specificaties. Op deze manier krijgen we meer vertrouwen in een specificatie hetgeen bijvoorbeeld wezenlijk is bij het afsluiten van contracten voor nog te bouwen systemen. We kunnen bij de ontwikkeling van een nieuw systeem reeds in een vroeg stadium fouten in het ontwerp ontdekken waardoor onkosten bespaard blijven. Verder is het vaak ook mogelijk om op basis van een specificatie prototypen van een nog te bouwen systeem te genereren zodat gebruikers sneller inzicht krijgen in hetgeen ze van een systeem kunnen verwachten.

Algebraïsche specificatieformalismen vormen een van de grotere takken aan de rijk geschakeerde boom van specificatieformalismen. Een algebraïsche specificatie bestaat uit de declaratie van functies waarvan de definitie wordt gegeven met behulp van vergelijkingen. Zo'n specificatie kan met behulp van de computer getest worden door de vergelijkingen te gebruiken bij het uitrekenen van termen opgebouwd uit de gedeclareerde functies. We zullen dit zo dadelijk aan de hand van een voorbeeld toelichten.

In dit proefschrift staat het algebraïsche specificatieformalisme ASF (Algebraic Specification Formalism) centraal. Dit formalisme werd ontwikkeld in het kader van de ESPRIT projecten 348 (GIPE - Generation of Interactive Programming Environments) en 2177 (GIPE II). Deze projecten hebben tot doel om op basis van een formele specificatie van een programmeertaal een interactieve programmeeromgeving voor die taal te genereren. Een programmeeromgeving bevat de verzameling van alle software die noodzakelijk of behulpzaam is bij het werken in een specifieke programmeertaal. Voor het vastleggen van de grammatica van een programmeertaal is in dit kader het formalisme SDF (Syntax Definition Formalism) ontwikkeld. De combinatie van ASF met SDF levert een formalisme voor de specificatie van programmeertalen.

Als klein voorbeeld van een specificatie in ASF+SDF behandelen we een taal waarmee een wandeling door New York (ofwel een wandeling op ruitjespapier) beschreven kan worden. Dit levert de volgende specificatie waarbij de regelnummers aan de rechterkant zijn toegevoegd om de bespreking te vereenvoudigen.

```

module Wandelingen                                     (1)
  exports                                              (2)
    sorts Opdracht Wandeling                         (3)
    lexical syntax                                    (4)
```

| | | |
|-----------------|-----------------------------------|------|
| [\n] | -> LAYOUT | (5) |
| context-free | syntax | (6) |
| noord | -> Opdracht | (7) |
| oost | -> Opdracht | (8) |
| zuid | -> Opdracht | (9) |
| west | -> Opdracht | (10) |
| start Opdracht* | stop -> Wandeling | (11) |
| hiddens | | (12) |
| variables | | (13) |
| Opd1 | -> Opdracht* | (14) |
| Opd2 | -> Opdracht* | (15) |
| equations | | (16) |
| [1] | start Opd1 zuid west Opd2 stop | (17) |
| | = start Opd1 west zuid Opd2 stop | (18) |
| [2] | start Opd1 zuid oost Opd2 stop | (19) |
| | = start Opd1 oost zuid Opd2 stop | (20) |
| [3] | start Opd1 noord west Opd2 stop | (21) |
| | = start Opd1 west noord Opd2 stop | (22) |
| [4] | start Opd1 noord oost Opd2 stop | (23) |
| | = start Opd1 oost noord Opd2 stop | (24) |
| [5] | start Opd1 noord zuid Opd2 stop | (25) |
| | = start Opd1 Opd2 stop | (26) |
| [6] | start Opd1 zuid noord Opd2 stop | (27) |
| | = start Opd1 Opd2 stop | (28) |
| [7] | start Opd1 west oost Opd2 stop | (29) |
| | = start Opd1 Opd2 stop | (30) |
| [8] | start Opd1 oost west Opd2 stop | (31) |
| | = start Opd1 Opd2 stop | (32) |

In de regels 3 tot en met 11 wordt de grammatica (ookwel de syntax genoemd) vastgelegd. Een wandeling bestaat uit de opdracht start gevolgd door nul of meer opdrachten gevolgd door stop (zie regel 11). Het sterretje * achter Opdracht betekent namelijk dat we nul of meer opdrachten op die plaats achter elkaar mogen zetten. De verschillende mogelijke opdrachten zijn noord, zuid, west en oost (regels 7 tot en met 10). Deze staan steeds voor een wandeling ter lengte van één blok in de aangegeven richting. In regel 5 wordt de *layout* gedefinieerd. Dit is de verzameling van symbolen (in dit geval de spatie en de overgang naar een nieuwe regel \n) die mogen worden overgeslagen bij het herkennen van de tekst. Met dit gedeelte van de specificatie is de computer al in staat om een zin als

start zuid oost noord west noord oost zuid stop

te herkennen als een wandeling. Op deze manier kan de in de specificatie gedefinieerde syntax getest worden.

De vergelijkingen in de regels 16 tot en met 32 leggen vast welke wandelingen hetzelfde resultaat hebben. De eerste vergelijking ([1] in regels 17 en 18) drukt uit dat een wandeling waarin zuid gevolgd door west voorkomt, hetzelfde is als de wandeling met die twee opdrachten omgedraaid. Daarbij staan Opd1 en Opd2 voor nul of meer opeenvolgende opdrachten. Ook deze twee *variabelen* moeten we in de specificatie opnemen en dat gebeurt in de regels 12 tot en met 15.

Om de vergelijkingen van de specificatie te testen worden ze gebruikt als herschrijfgeregels. Dat wil zeggen dat ze zo vaak als mogelijk van links naar rechts worden toegepast. Het bovenstaande voorbeeld kan hierdoor vereenvoudigd worden tot:

```
start oost stop.
```

Grotere specificaties (specificaties van een programmeertaal lopen al snel uit op enkele honderden zo niet duizenden vergelijkingen) zouden voor de mens onleesbaar worden, indien er geen mogelijkheid zou zijn om een specificatie op te splitsen in logisch bij elkaar behorende delen. Zo'n deel van een specificatie noemen we een *module*. Het gedeelte van de specificatie dat alleen lokaal (dat wil zeggen: binnen de module) gebruikt mag worden, wordt opgenomen onder *hiddens*. Datgene dat ook buiten de module beschikbaar is valt onder *exports*.

Stel bijvoorbeeld dat we bovenstaande specificatie willen uitbreiden met een extra opdracht dubbelnoord. De betekenis van deze opdracht is gelijk aan twee achtereenvolgende opdrachten noord. De volgende specificatie geeft dit precies weer.

```
module Wandelingen-Plus

  imports
    Wandelingen

  exports
    context-free syntax
      dubbelnoord -> Opdracht

  hiddens
    variables
      Opd1 -> Opdracht*
      Opd2 -> Opdracht*

  equations

    [1]  start Opd1 dubbelnoord Opd2 stop
         = start Opd1 noord noord Opd2 stop
```

Inhoudsbeschrijving

Het eerste hoofdstuk van het proefschrift bevat een informele introductie in de formalismen die in de rest van het proefschrift gebruikt worden. Dit gebeurt aan de hand van een aantal voorbeelden waarin natuurlijke getallen en (eindige) verzamelingen van natuurlijke getallen beschreven worden. Eerst wordt het algebraïsch specificatieformalisme ASF uitgelegd, daarna het formalisme SDF voor de specificatie van syntax, en tenslotte de combinatie van deze twee: ASF+SDF.

In hoofdstuk 2 wordt een beschrijving gegeven van het ASF systeem. Dit is een eenvoudig, batch-georiënteerd systeem waarmee specificaties geschreven in ASF verwerkt en getest kunnen worden. *Batch-georiënteerd* wil zeggen dat een specificatie in zijn geheel verwerkt wordt. Bij elke wijziging in de specificatie, zal deze in zijn geheel opnieuw verwerkt moeten worden voordat men de nieuwe specificatie kan testen. Bij de verwerking van een specificatie wordt eerst gecontroleerd op syntactische (een spellingsfout bijvoorbeeld) en statisch semantische (een soort wordt wel gebruikt maar is niet gedeclareerd) fouten. Als dat goed verloopt wordt er code gegenereerd waarmee de specificatie getest kan worden. De vergelijkingen uit de specificatie worden gebruikt om een term uit te rekenen.

Hoe de generatie van code in het ASF systeem werkt, wordt beschreven in hoofdstuk 3. In dat hoofdstuk worden tevens twee uitbreidingen van ASF beschreven, die nodig zijn om ASF te kunnen combineren met SDF. Het gaat hierbij om de uitbreiding met lijst constructoren en associatieve operatoren. Met behulp van *lijst constructoren* kunnen lijsten met willekeurig veel termen gemaakt worden. Dit is onder andere nodig voor de verwerking van de * zoals die voorkomt in het hierboven gegeven voorbeeld op regels 11 en 14. *Associativiteit* betekent voor een binaire operator \circ dat de vergelijking

$$(x \circ y) \circ z = x \circ (y \circ z)$$

geldt. De optelling van getallen is een voorbeeld van een zo'n operator. Bovenstaande vergelijking levert in het algemeen problemen op omdat bij het uitrekenen van termen het soms nodig is om deze vergelijking van links naar rechts te gebruiken en soms van rechts naar links.

In hoofdstuk 4 wordt een groter voorbeeld van een specificatie geschreven in ASF+SDF behandeld. De specificatie definieert de statische semantiek van een programmeertaal met polymorfie en type inferentie. *Type inferentie* betekent dat in die programmeertaal het type van operatoren niet gedeclareerd wordt. Uit het gebruik van de operatoren moet dan worden afgeleid of er in het programma strijdigheden voorkomen. *Polymorfie* houdt in dat eenzelfde operator op meerdere manieren getypeerd kan zijn.

Het ASF+SDF systeem wordt in hoofdstuk 5 beschreven. Dit is een *interactief* systeem dat de ontwikkeling en het testen van specificaties geschreven in ASF+SDF ondersteunt. Interactief betekent in dit geval dat een specificatie ontwikkeld wordt met behulp van een syntax-gestuurde editor (een editor die de grammatica kent en waarschuwt indien de gebruiker grammaticale fouten maakt). Na elke edit-operatie wordt de bij de specificatie behorende implementatie bijgewerkt zodat deze onmiddellijk getest kan worden. Dit is een belangrijke verbetering ten opzichte van het ASF systeem uit hoofdstuk 2 waarbij na elke verandering de generatie van code helemaal opnieuw moet plaatsvinden.

Specificaties worden uitermate onbegrijpelijk als ze niet zijn opgesplitst in logisch samenhangende fragmenten. De meeste specificatieformalismen ondersteunen dan ook een of andere vorm van modularisering. De eenvoudigste variant hiervan is de tekstuele modularisering. Daarbij wordt een specificatie opgedeeld in modules met elk een eigen naam. Indien we een bepaalde module willen gebruiken in een andere, dan wordt de naam van de eerste toegevoegd aan de lijst van imports van de tweede.

In hoofdstuk 6 wordt een wiskundige definitie van deze modulariseringstechniek gegeven zodanig dat deze techniek kan worden toegevoegd aan elk specificatieformalisme dat zelf geen modularisering ondersteunt. De gevolgen van deze toevoeging worden voor enkele specifieke specificatieformalismen uitgewerkt. Tevens wordt in dit hoofdstuk een specificatie in ASF+SDF van deze modulariseringstechniek gegeven.

De globale architectuur van een systeem dat specificaties geschreven in een formalisme dat tekstuele modularisatie ondersteunt, wordt beschreven in hoofdstuk 7. Tevens worden drie algoritmes behandeld, die in de implementatie van zo'n systeem een rol spelen. Het ASF+SDF systeem is een speciaal geval van de hier beschreven architectuur.