The Requirement and Design Specification Language SPECTRUM An Informal Introduction Version 1.0*

The Munich SPECTRUM Group:

Manfred Broy, Christian Facchi, Radu Grosu,
Rudi Hettler, Heinrich Hussmann, Dieter Nazareth,
Franz Regensburger, Oscar Slotosch, Ketil Stølen

Fakultät für Informatik, Technische Universität München
80290 München, Germany

E-Mail: spectrum@informatik.tu-muenchen.de

In Cooperation with
Tobias Nipkow, Martin Wirsing

May 17, 1993

*This work is sponsored by the German Ministry of Research and Technology (BMFT) as part of the compound project "KORSO - Korrekte Software" and by the German Research Community (DFG) project SPECTRUM.

Abstract

This paper gives a short introduction to the algebraic specification language SPECTRUM. Using simple, well-known examples, the objectives and concepts of SPECTRUM are explained.

The SPECTRUM language is based on axiomatic specification techniques and is oriented towards functional programs. SPECTRUM includes the following features:

- partial functions, definedness logic and fixed point theory
- higher-order elements and typed λ-abstraction
- non-strict functions and infinite objects
- full first-order predicate logic with induction principles
- predicative polymorphism with sort classes
- parameterization and modularization

SPECTRUM is based on the concept of loose semantics.

Contents

1 Introduction **3**

2 Specifying in the Small **5**
2.1 General Structure of a SPECTRUM Specification 5
2.2 Loose Semantics 7
2.3 Generation Principles and Freeness 8

3 Functions **10**
3.1 Partial Functions 11
3.2 Non-Strict Functions 13
3.3 Higher-Order Functions and λ-Abstraction 14
3.4 Infinite Elements 15

4 Predicates and non–continuous Functions **20**

5 The Sort System of SPECTRUM **22**
5.1 Sort Constructors 22
5.2 Polymorphism 23
5.3 Sort Classes 25
5.3.1 Partial Order on Sort Classes 28
5.3.2 Overloading with Sort Classes 29
5.4 Sort Inference 30

6 Derived Constructs **32**
6.1 Semantics of **strict**, **total** and **strong** 32
6.2 Semantics of **freely generated by** 33
6.3 Data Type 36
6.4 Sort Synonyms 39
6.5 Let and Letrec Expressions 40
6.5.1 Let Expressions 40
6.5.2 Letrec Expressions 41
6.5.3 If–Then–Else Expressions 42
6.6 Built–in Data Types 43

6.6.1 Numeric Data Types 43
6.6.2 The Data Type List 43
6.6.3 Strings 43

7 Specifying in the Large **44**
7.1 Combination 45
7.2 Renaming 46
7.3 Export and Hiding 47
7.4 Enrichment 48
7.5 Parameterization 49
7.5.1 Parameterization with Sort Classes 49
7.5.2 Classical Parameterization 51

8 Executable Sublanguage **55**
8.1 Executable Sublanguage for Strict Target Languages 56
8.2 Executable Sublanguage for Lazy Target Languages 57

9 Acknowledgement **58**

A Concrete Syntax of SPECTRUM **63**
A.1 Notational Conventions 63
A.2 Lexical Syntax 64
A.3 Contextfree Syntax 66

B Models and Logic **70**
B.1 Signatures 70
B.2 The Language of Terms 74
B.2.1 Context Free Language (Pre-Terms) 74
B.2.2 Context Sensitive Language 74
B.2.3 Well-formed Terms and Sentences 77
B.3 Algebras 78
B.3.1 The Sort Algebras 80
B.4 Models 83
B.4.1 Interpretation of Sort Assertions 83
B.4.2 Satisfaction and Models 85

C Generation Principles and Induction **87**
C.1 Reachable Algebras 87
C.2 Induction on Constructor Terms 88
C.3 Semantics of **generated by** 90

D SPECTRUM's Predefined Specification **91**

E Standard Library **94**

Chapter 1

Introduction

Software adequacy and reliability are two of the main goals in software production. Unfortunately, given the current state of the art in software engineering, these goals are very hard to achieve. The systematic use of correctness-oriented, stepwise development techniques based on formal derivation and verification methods has been proposed as a step forward.

A necessary prerequisite for correctness-oriented program development is a formal description of the program's task. This description is often called requirement specification. In fact, the requirement specification forms the borderline between informal program description and formal program development with specification and verification. A simple and well-structured requirement specification may considerably ease coding and verification phases.

The SPECTRUM project concentrates on the process of developing well-structured, precise specifications. SPECTRUM comprises a specification language, a deduction calculus and a development methodology. As the name of the project indicates, a wide range of specification styles is envisaged, including means to describe concurrent systems. SPECTRUM allows the user to write very abstract (and definitely non-executable) specifications as well as specifications which can be read and executed as functional programs.

SPECTRUM is based on algebraic specification techniques and on the experience gained in the project CIP [BBB+85]. However, in contrast to most algebraic specification languages (like e.g. OBJ [JKKM88], LARCH [GHW85], ACT ONE [EM85], OBSCURE [LL88] and ACT TWO [EM90]), it contains explicit support for partial functions (as a generalization of [BW82]). Moreover, SPECTRUM is not restricted to equational or conditional-equational axioms, since it does not primarily aim at executable specifications.

SPECTRUM does not feature any built-in notion of a program state, and the whole specification style is oriented towards functional programming. As a consequence of this dedication a number of functional language concepts have been integrated into the specification language. SPECTRUM's sort system provides parametric polymorphism and sort classes in the style of functional programming lan-

guages like Haskell [HJW92] or theorem provers like ISABELLE [Nip91]. Furthermore the language supports the denotation of functions by typed λ-abstraction and higher-order functions. Also the semantics have been adapted to recursion in functional programming using complete partial orderings for carrier sets.

Since writing well-structured specifications is one of our main goals, a flexible language for structuring of specifications has been designed for SPECTRUM. This language was originally inspired by ASL [SW83]. The current version is more closely related to functional programs, to LARCH and to PLUSS [Gau86].

There are only a few specification languages available which are comparable with SPECTRUM. PLUSS has many similarities with SPECTRUM (e.g. partial functions). In fact, PLUSS has influenced the design of SPECTRUM in a number of ways (e.g. concerning the treatment of hiding and of term generation conditions). Nevertheless, many of the SPECTRUM features are not supported by PLUSS, in particular non-strict functions, higher-order functions and the advanced sort system. The language Pannda-S developed in the PROSPECTRA project [KBH91] can be seen as a predecessor of SPECTRUM.

The definition of the semantics and the proof system was influenced by languages and systems of the LCF family [Pau87]. The expressiveness of SPECTRUM is close to that of Cambridge LCF but in contrast to this system SPECTRUM uses a three-valued logic, supports sort classes and offers a concrete syntax for a language that is to be used in the process of system specification.

The paper at hand is an update of the previous informal introduction paper. It describes informally the concepts and syntax of the language. A number of papers giving a more formal definition are in preparation.

The document is organized as follows: Chapter 2 introduces the most basic specification constructs. The handling of partial, non-strict and higher-order functions is the topic of Chapter 3. In SPECTRUM a function is always continuous. What is usually meant by noncontinuous function is called mapping in SPECTRUM. Mappings are used only for specification purposes. Chapter 4 is devoted to the specification of mappings. The sort system of SPECTRUM is described in Chapter 5. Chapter 6 introduces some constructs that do not have a direct algebraic semantics but have a semantics that is defined in terms of more basic constructs. Instead they are syntactically checked and expanded into SPECTRUM text which could be written without them. Chapter 7 deals with the structuring of specifications, and with parameterization. Chapter 8 addresses the executable sublanguage of SPECTRUM that builds the borderline to target languages like Haskell or ML.

Appendix A contains the concrete syntax and the ASCII representations of graphic symbols, respectively. Appendix B gives a brief introduction to the semantics and underlying logic of SPECTRUM. Finally, the predefined signature and a standard library that defines often used sorts and functions can be found in Appendix D.

Chapter 2

Specifying in the Small

Within SPECTRUM the notation for specifying a component "in the small" and the notation for structuring a specification "in the large" are carefully separated. The latter notation may be used to combine and adapt specifications into new and often more complex specifications. Moreover, this notation is intended for building-up and structuring large and complicated system descriptions. For the design of a single specification unit, i.e. a basic component, on the other hand, only the first notation is employed.

This chapter gives a brief introduction to specifying a component "in the small". A more detailed discussion of specific features follows in Chapters 3, 4 and 5. The structuring of a specification "in the large" is the topic of Chapter 7.

2.1 General Structure of a SPECTRUM Specification

We start by giving a specification of the natural numbers. This simple example is well-suited for introducing the most basic constructs of a SPECTRUM specification.

```
NAT = {

    --All defined functions are strict and total
    strict;
    total;

    sort Nat;

    --Generator functions
    zero : Nat;
    succ : Nat → Nat;
```

```
    --Non-generator functions
    .+. : Nat × Nat ⟶ Nat      prio 6:left;
    .*. : Nat × Nat ⟶ Nat      prio 7:left;

    --Induction principle for natural numbers
    Nat generated by zero, succ;

    axioms ∀ n, m : Nat in

  {free1}  zero ≠ succ n;
  {free2}  succ n = succ m ⇒ n = m;

  {plus1}  n + zero = n;
  {plus2}  n + succ m = succ (n + m);

 {times1}  n * zero = zero;
 {times2}  n * succ m = n + n * m;

    endaxioms;
}
```

The first thing to observe is that in SPECTRUM every comment starts with a "−−". In this paper we use a bold font to distinguish keywords from other specification text[1]. A SPECTRUM specification "in the small", as usual within the algebraic framework, consists of a *signature* part and an *axioms* part. A basic specification can, however, contain more than one signature or axioms part. A signature part contains defined sorts as well as defined constant and function symbols together with their sorts and *functionalities* (argument and result sorts), respectively. The following should be observed:

Strictness and Totality: The keywords **total** and **strict** restrict all functions of a basic specification to be total and strict. These terms are abbreviations of axioms and will be explained in Chapter 3.

Infix Function Symbols: Binary function symbols can be defined as infix function symbols with an user-definable priority and associativity, like in many modern functional and logic programming languages. The keyword **prio** followed by a natural number p assigns the priority p to the infix symbol in question. The function symbol with the higher priority binds stronger.

[1]The SPECTRUM syntax employs graphic symbols like ∀ as we intend to use the language in a graphic environment. There is also an ASCII version of the syntax (see Appendix A.1).

Given the above specification this means for example that $\mathsf{l} + \mathsf{m} * \mathsf{n}$ is an abbreviation for $\mathsf{l} + (\mathsf{m} * \mathsf{n})$. The parsing of flattened terms like $\mathsf{l} + \mathsf{m} + \mathsf{n}$ is declared with the keywords **right** and **left** (right and left bracketing), respectively. With respect to NAT this means that $\mathsf{l} + \mathsf{m} + \mathsf{n}$ is an abbreviation for $(\mathsf{l} + \mathsf{m}) + \mathsf{n}$.

Generators: The **generated by**-phrase can be thought of as an axiom scheme. It basically allows the usual structural induction. This concept will be discussed in more detail in Sections 2.3 and 3.4.

Built-in Equality: A universal equality predicate denoted by .=. and its negation .≠. are automatically available for all sorts (see Appendix D).

General Predicate Logic: The axioms are not restricted to equations or conditional equations. They may contain arbitrary formulae from a predicate logic calculus (with equality) built over the given signature. They may contain even, for instance, nested quantifiers.

Scope of Variables: The scope of the universally quantified variables of the **axioms**-construct includes the whole list of axioms. In our introduction example these are the variables n and m, both declared to be of sort Nat.

Identifier for Axioms: Every axiom may be labeled with an identifier. For the definition of identifier see Appendix A.

2.2 Loose Semantics

The semantics of a SPECTRUM specification is denoted by the class of all algebras[2] with appropriate signature which fulfill the given axioms. This is known as *loose semantics* (cf. CIP-L [BBB+85], LARCH [GHW85], ASL [Wir86]) and differs from the approach taken in many other algebraic specification languages (e.g. ACT ONE [EM85], ACT TWO [EM90], ASF [BHK89], OBJ [JKKM88]), where a particular model, the *initial* one, is chosen. Loose semantics allows the user to write down very abstract specifications, which leaves a large degree of freedom for later implementation. By reducing the number of models, e.g. by giving more axioms, and thereby imposing design decisions, a refinement notion for the stepwise development of data structures and algorithms can be achieved. There is also another reason for using a loose semantics: for the general class of axioms admitted in SPECTRUM initial models do not always exist.

[2] The notion of algebra is clarified in Appendix B.

2.3 Generation Principles and Freeness

As already indicated, the model class may be restricted not only by axioms of first-order predicate logic, but also by a statement of the form

S **generated by** G;

where S is a list of sorts and G is a list of constants and function symbols. All constants have a sort $\mathsf{s} \in \mathsf{S}$ and the functions have argument sorts $a \in (\mathsf{S} \cup \mathsf{P})$ and range $\mathsf{s} \in \mathsf{S}$ where P is a set of primitive sorts not occurring in S. At the syntactic level this phrase allows the use of simultaneous structural induction on all sorts listed in S with respect to the constructors listed in G. Semantically, this means that all carriers for the sorts in S are generated by the primitive sorts in P and the functions and constants in G. In the literature this is also called the inductive closure of P and G (cf. [Gal86]). This semantics ensures that structural induction is sound and it is an extension of the usual notion of term generatedness because structural induction may be sound for a sort s even if there are no term models for s at all. Suppose the case where s is built on top of a primitive sort $\mathsf{p} \in \mathsf{P}$ that is not representable by terms.

In Section 3.4 we will see a further extension of the concept of term generation. In SPECTRUM all carriers are complete partial orderings and therefore it is possible to extend the principle of structural induction. All properties which are compatible with the partial ordering (cf. admissible or chain complete predicates [Pau87]) can be proved inductively for all elements that are expressible by the constructors and then the property is propagated to the limit points of chains which yields the universality of the property. In Section 3.4 we give examples for sorts with limit points and appropriate induction rules.

In SPECTRUM a freely generated structure may be specified by the following phrase

S **freely generated by** G;

This scheme imposes exactly the same generation principle as **generated by**, but with the additional constraint that the carriers for the sorts in S are *freely* generated by the primitive sorts in P and the functions and constants in G (cf. [Gal86]). Roughly speaking, different constructors generate different elements and all constructor functions are injective. In SPECTRUM it is possible to express the additional freeness constraints directly by axioms[3] although it is boring to write down all the necessary axioms. The keyword **freely generated by** was added for the sake of readability and it is simply an abbreviation for these axioms.

This means for example, that if

[3] In the special case of free generation even the principle of structural induction can be formalized directly by axioms. This technique is used by LCF and is described in [Pau87].

Nat **freely generated by** zero, succ;

is substituted for

Nat **generated by** zero, succ;

in the NAT specification, then the two axioms

zero $\neq$ succ n;
succ n = succ m $\Rightarrow$ n = m;

are no longer needed since they are implied by the **freely generated by** phrase.

Chapter 3

Functions

In the previous chapter it was not necessary to mention partial functions, the values of undefined terms and how the specified functions deal with them. Moreover, functions which take other functions as arguments were not specified. This is no coincidence — actually, Chapter 2 presents only one facet of SPECTRUM, namely the specification of *first-order* functions which are both *total* and *strict*:

Total: A function f is *total* iff f yields a defined result whenever all arguments are defined.

Strict: A function f is *strict* iff f yields an undefined result whenever f is applied to at least one undefined argument.

First-order: A function f is *first-order* iff no functional sorts appear in a parameter sort or the result sort of f. In other words, the sort of f contains the function sort constructor $\longrightarrow$ only at the top level.

On many occasions, however, more general instances of functions are needed. Functions that are not (necessarily) total are called *partial* functions. This definition of partiality contains the total functions as a special case. A function that does not need to obey the above strictness requirement is called *non-strict*. As before, this includes strict functions as special case. A function that is not first-order is called *higher-order*. Obviously, according to this definition, first-order functions cannot be seen as a special case of higher-order functions.
The objective of this chapter is to explain how SPECTRUM can be used to define the latter sorts of functions. In SPECTRUM functions are always *continuous*. What is usually meant by *noncontinuous function* is called *mapping* in SPECTRUM. They are used for specification purposes. The specification of mappings is the topic of Chapter 4.

Remark: From now on we do not distinguish between function symbols and the semantic values they denote where this distinction is clear from the

context. In other words, instead of writing "the function denoted by f yields ..." we will often write "the function f yields ...".

3.1 Partial Functions

The specification NAT of Chapter 2 was an example of a specification that contains only strict and total functions. In this example, totality of all functions was enforced by the line reading

total;

at the top of the specification. This line is called a *totality axiom*. It demands totality of all functions introduced in the signature[1]. To specify partial functions we use a variant of this totality axiom that requires totality only of some functions. As an example we give an extended version of the specification of natural numbers:

```
NAT' = {
    strict;

    sort Nat;

    zero : Nat;
    succ : Nat ⟶ Nat;
    pred : Nat ⟶ Nat;
    .≤.,.<. : Nat × Nat ⟶ Bool          prio 6;
    .+. : Nat × Nat ⟶ Nat               prio 6:left;
    .−. : Nat × Nat ⟶ Nat               prio 6;
    .∗. : Nat × Nat ⟶ Nat               prio 7:left;
    .div.,.mod. : Nat × Nat ⟶ Nat       prio 7;

    succ,.≤.,.<.,.+.,.∗. total;

    Nat freely generated by zero, succ;

    axioms ∀ n, m : Nat in

        δ (pred n) ⇔ n ≠ zero;
        pred(succ n) = n;
```

[1] Properties like totality and strictness can, of course, be stated in terms of axioms using the definedness predicate δ (see Section 6.1). However, for concepts which are as commonly used as totality and strictness, the more comfortable shorthand notations of strictness and totality axioms have been introduced into SPECTRUM.

$\mathsf{zero} \leq \mathsf{n};$
$\neg(\mathsf{succ\ n} \leq \mathsf{zero});$
$\mathsf{succ\ n} \leq \mathsf{succ\ m} \Leftrightarrow \mathsf{n} \leq \mathsf{m};$
$\mathsf{n} < \mathsf{m} = (\mathsf{n} \leq \mathsf{m} \wedge \mathsf{n} \neq \mathsf{m});$

$\mathsf{n} + \mathsf{zero} = \mathsf{n};$
$\mathsf{n} + \mathsf{succ\ m} = \mathsf{succ}\ (\mathsf{n} + \mathsf{m});$

$\delta\ (\mathsf{n} - \mathsf{m}) \Leftrightarrow \mathsf{m} \leq \mathsf{n};$
$(\mathsf{n} + \mathsf{m}) - \mathsf{m} = \mathsf{n};$

$\mathsf{n} * \mathsf{zero} = \mathsf{zero};$
$\mathsf{n} * \mathsf{succ\ m} = \mathsf{n} + \mathsf{n} * \mathsf{m};$

$\delta\ (\mathsf{n\ div\ m}) \Leftrightarrow \mathsf{m} \neq \mathsf{zero};$
$\delta\ (\mathsf{n\ mod\ m}) \Leftrightarrow \mathsf{m} \neq \mathsf{zero};$
$\mathsf{m} \neq \mathsf{zero} \Rightarrow \mathsf{n\ mod\ m} < \mathsf{m};$
$\mathsf{m} \neq \mathsf{zero} \Rightarrow \mathsf{n} = (\mathsf{n\ div\ m}) * \mathsf{m} + \mathsf{n\ mod\ m};$

endaxioms;
}

In this example the following details should be observed:

Strictness: As in NAT the line reading

strict;

requires all functions that are introduced in the signature of NAT' to be strict.

Totality: The totality axiom

succ,.≤.,.<.,.+.,.∗. **total**;

demands totality of the functions succ, .≤., .<., .+. and .∗.. No such requirement is made for pred, .−., .div. and .mod., therefore those functions are partial (i.e. may yield an undefined result for defined arguments).

Universal Quantifier: The universal quantifier in the **axioms** construct ranges over defined values only.

Definedness Predicate: In order to deal with partial functions, a standard predicate symbol δ is available for all sorts. Its interpretation evaluates to true if the interpretation of its argument term is defined and to false otherwise. This definedness predicate is mainly used to describe which arguments of a function lead to defined results. For example, the strictness of pred and the third axiom in the above specification imply that pred is defined whenever its argument is defined and not equal to zero.

Strong Equality: As already mentioned, a universal equality predicate denoted by .=. and its negation .≠. are automatically available for all sorts. This standard equality is *strong* by definition, i.e. it always yields true or false, considering all "undefined values" of identical sort to be equal, and all "defined values" to be different from "undefined values". This means for example that

$$\neg\ \delta(\mathsf{x}) \wedge \neg\ \delta(\mathsf{y}) \Rightarrow \mathsf{x} = \mathsf{y}$$

3.2 Non-Strict Functions

The possibility to specify non-strict functions has been integrated into SPECTRUM for several reasons. First, they support the logical level (as explained in Appendix B the logical connectives .∧., .∨. and .⇒. are all non-strict). A second reason is that non-strict functions allow to specify infinite elements[2]. Moreover, non-strict functions can be used to extend the specification language in a very convenient way. As a first example, suppose we want to specify an if_then_else function for the sort Nat[3]. The signature of this function is

$$\mathsf{nat\_if} : \mathsf{Bool} \times \mathsf{Nat} \times \mathsf{Nat} \longrightarrow \mathsf{Nat};$$

Totality of nat_if is expressed by

nat_if **total**;

Furthermore, we observe that nat_if is non-strict (e.g. nat_if(true,e,⊥) yields e). However, as nat_if(⊥,e,f) always yields ⊥, it is *strict in the first argument*. Note that in SPECTRUM every sort has an undefined value which is denoted by ⊥, i.e. $\neg(\delta\bot)$ always holds. The notion of *strictness axioms* in SPECTRUM is able to express strictness on the argument level. In our example we can write

nat_if **strict in** 1;

[2]This will be shown in Section 3.4.

[3]In Appendix D we will introduce a more general if_then_else function in almost the same way.

to express the strictness properties of nat_if[4]. Now we can specify the behaviour of nat_if:

axioms $\forall^{\perp}$ t, e : Nat **in**
    nat_if(true, t, e) = t;
    nat_if(false, t, e) = e;
endaxioms;

The following should be observed:

Quantification over Undefined Values: It has already been pointed out that $\forall$ quantifies only over defined values, and the same is true for $\exists$. However, on many occasions quantification over both defined and undefined values leads to considerably simpler specifications. For this purpose SPECTRUM offers two additional quantifiers $\forall^{\perp}$ and $\exists^{\perp}$, the universal quantifier and the existential quantifier over both defined and undefined values, respectively[5]. The "new" universal quantifier $\forall^{\perp}$ is used in the example above. If $\forall$ had been employed instead we would have needed 6 additional axioms to state an equivalent specification (see also how $\forall^{\perp}$ is employed to quantify over the empty stream in the specification STREAM on page 18).

3.3 Higher-Order Functions and λ-Abstraction

As pointed out in the introduction, SPECTRUM aims at the development of functional programs. Therefore a number of useful features of functional programming languages (like ML [HMM86], Haskell [HJW92], MIRANDA [Tur85], OPAL [Gro91]) have been included in the language. Among these concepts are *higher-order functions*. A function f is called higher-order if $\longrightarrow$ occurs in its argument and/or result sort.

As an example see the following specification fragment which extends the NAT' specification of Section 3.1 by the higher-order function twice:

twice : (Nat $\longrightarrow$ Nat) $\longrightarrow$ (Nat $\longrightarrow$ Nat);

twice **strict total**;

axioms $\forall$ x : Nat, f : Nat $\longrightarrow$ Nat **in**
    twice f x = f(f x);
endaxioms;

[4]Of course, the same fact could have been stated as a logical axiom using the definedness predicate δ. Like totality axioms, strictness axioms are just convenient shorthand notations (see Section 6.1).

[5]Note that $\forall$x. P(x) is only a shorthand for $\forall^{\perp}$x. δ x $\Rightarrow$ P(x).

For every function f of sort Nat $\longrightarrow$ Nat, twice(f) yields a function of sort Nat $\longrightarrow$ Nat that has the same effect as the composition of f with itself. In the above axioms the function twice is characterized by its application to a function f and a natural number x. For denoting functions classical typed λ-notation is available in SPECTRUM. We can for example write

$\boldsymbol{\lambda}$ x : Nat. succ x;

to denote the successor function by an explicit λ-term. Thus, alternatively we could have characterized the function twice by the axiom

twice = $\boldsymbol{\lambda}$ f : Nat $\longrightarrow$ Nat. $\boldsymbol{\lambda}$ x : Nat. f(f x);

A specific higher-order function that is extremely useful is the fixed point operator fix. In SPECTRUM it is predefined for every sort α. The functionality of fix is

fix : $(\alpha \longrightarrow \alpha) \longrightarrow \alpha$;

It is characterized by the axioms

fix f = f(fix f);
y = f y $\Rightarrow$ fix f $\sqsubseteq$ y;

The first axiom implies that fix f is a fixed point of f. The second axiom specifies that this fixed point is the least one. In SPECTRUM all domains are partially ordered with $\perp$ as the least element. The symbol $\sqsubseteq$ is used to denote this ordering relation (see Chapter 4 and Appendix B). It is predefined for every sort. Since in SPECTRUM all functions are assumed to be continuous, least fixed points always exist.

The fixed point operator supports the explicit use of recursively defined functions. For example, we may represent the division on natural numbers by

```
div = fix(λ f : Nat × Nat ⟶ Nat.
          λ x, y : Nat.
          nat_if(x < y, zero, f(x − y, y) + 1)
         );
```

3.4 Infinite Elements

There is another reason for allowing non-strict functions in SPECTRUM: non-strict constructor functions may be used to describe infinite elements [Möl82]. It has already been explained (see Section 2.3) what

S **generated by** G;

means semantically: all carriers for the sorts in S are generated by the primitive sorts in P and the functions and constants in G. As all carriers have to be complete partial orderings this means that the sorts in S are complete with respect to chains and with respect to application of functions in G.

If all constructors in G are strict and all chains in the primitive sorts are finite[6] then the inductive closures of the sorts in S with respect to the functions in G are trivially chain-complete. This is due to the strictness of the constructors. As an example consider the sort Nat specified in NAT. An appropriate rule for structural induction on Nat is

$$\frac{\begin{array}{l} H1 \vdash \forall^{\perp} n.\delta(P[n/m]) \\ H2 \vdash P[\perp/m] \\ H3 \vdash P[\mathsf{zero}/m] \\ H4 \vdash \forall^{\perp} n.\delta(\mathsf{succ}(n)) \wedge P[n/m] \Rightarrow P[\mathsf{succ}(n)/m] \end{array}}{H1, H2, H3, H4 \vdash \forall^{\perp} n.P[n/m]}$$

The first premise guarantees that the boolean term P characterizes a property in Nat. The second premise is necessary since all carriers are pointed cpo's. The $\perp$ term is a standard "constructor" for every sort although it is not mentioned explicitly in the **generated by** phrase. Therefore the second premise is a standard base case of induction. The third premise is the base case for the constructor zero and the fourth premise formalizes the induction step. The four premises together prove the property P for all elements expressible by finite constructor application. Since there are no other elements in the carrier of Nat it is sound to conclude that P holds for all elements in Nat[7]. If the desired conclusion of the induction is $\forall^{\perp} n.\delta n \Rightarrow P[n/m]$ the rule can be refined using $\forall$ instead of $\forall^{\perp}$.

$$\frac{\begin{array}{l} H1 \vdash \forall n.\delta(P[n/m]) \\ H2 \vdash P[\mathsf{zero}/m] \\ H3 \vdash \forall n.P[n/m] \Rightarrow P[\mathsf{succ}(n)/m] \end{array}}{H1, H2, H3 \vdash \forall n.P[n/m]}$$

Infinite chains in primitive sorts together with strict constructors (e.g. lists of functions) lead to infinite chains in the generated sorts but this is irrelevant for structural induction. All elements in the generated sorts are still expressible by finite constructor application using variables for elements of primitive sorts.

In the general case, when also non-strict functions may occur in G, the semantics are more complicated. Non-strict constructors lead also to infinite chains, but on top may be real limit points (infinite elements) that are not expressible by

[6] In LCF such sorts are called chain-finite [Pau87].

[7] In the Appendix C we will justify the principle of induction on constructor terms.

finite constructor application. These infinite chains and infinite elements require special attention when formulating a principle for structural induction.

We now give an example for a sort with an infinite element to illustrate the problem. We specify the natural numbers with the additional infinite element ω and the natural ordering $\bot \sqsubseteq \mathsf{succ}(\bot) \ldots \sqsubseteq \omega$. We use lnat (infinite naturals) instead of Nat and isucc instead of succ to emphasize the difference.

```
INAT = {

    sort lnat;

    isucc : lnat ⟶ lnat;

    lnat freely generated by isucc;

    axioms ∀⊥ n : lnat in
        δ(isucc ⊥);
        δ n ⇒ δ(isucc n);
    endaxioms;
}
```

First, observe that the two axioms state that isucc is total but not strict. Since all carriers are cpo's, $\bot$ denotes the least element and isucc is continuous and therefore monotonic with respect to $\sqsubseteq$, we get a chain with

$$\bot \sqsubseteq \mathsf{isucc}(\bot) \sqsubseteq \mathsf{isucc}^2(\bot) \sqsubseteq \ldots \sqsubseteq \mathsf{isucc}^n(\bot) \ldots,$$

where isucc^i represents isucc composed with itself $\mathsf{i} - 1$ times. Due to the loose semantics of SPECTRUM this chain may be trivial in some models. The intended model of the above specification is the one where all the isucc^i terms denote different elements in the carrier of lNat which should also contain the upper bound ω of this nontrivial chain. If we had used **generated by** instead of **freely generated by** we would not get the desired result since **generated by** simply means that lnat is generated by isucc but not how this is done. Being more precise about $.\sqsubseteq.$ we can overcome this problem by explicitly adding the axiom

$$\forall^{\bot}\mathsf{m,n}.\ \mathsf{isucc(n)} = \mathsf{isucc(m)} \Rightarrow \mathsf{n} = \mathsf{m};$$

which states the injectivity of isucc. We used **freely generated by** because this incorporates the semantics of **generated by** and automatically adds axioms that allow to deduce the stronger theorem[8]

[8]See Section 6.2 for a full description of **freely generated by**.

$\forall^{\perp}$m,n. isucc(n) $\sqsubseteq$ isucc(m) $\Rightarrow$ n $\sqsubseteq$ m;

Now it can be inferred that the carrier set for lnat contains exactly one element for each of these isucc^i terms plus one "infinite" element (the ω) representing the least upper bound of the resulting chain. Moreover, since there is a predefined least fixed point operator fix such that

fix isucc = isucc(fix isucc)
y = isucc y $\Rightarrow$ fix isucc $\sqsubseteq$ y

it follows that this infinite element is denoted by fix isucc. Therefore it is not necessary to introduce ω explicitly in the signature.
Now we have to be careful when formulating an induction principle. An appropriate rule for structural induction on lnat is

$$\frac{\begin{array}{l} H1 \vdash \forall^{\perp} n.\delta(P[n/m]) \\ H2 \vdash P[\perp/m] \\ H3 \vdash \forall^{\perp} n.\delta(\mathsf{isucc}(n)) \wedge P[n/m] \Rightarrow P[\mathsf{isucc}(n)/m] \end{array}}{H1, H2, H3 \vdash \forall^{\perp} n.P[n/m]} \quad (P \text{ admissible in } m)$$

The three premises together prove the property for all elements expressible by finite constructor application. In order to establish the universality of the property P we have to prove it for the limit points, too. In our example INAT we know that there is exactly one limit point and we could add P[fix(isucc)/m] as an additional premise and prove it directly by fixed point induction. This is not very elegant and in general, if there are (infinitely) many limit points, also impossible. The solution to this technical problem is to constrain P to be admissible which is also natural since explicit fixed point induction for all the limit points (if possible at all) would do the same.

A predicate is *admissible* if, whenever it holds for all elements of a chain, it also holds for the least upper bound of the chain (see [Man74] for a detailed treatment). Therefore if P is admissible it is sound by definition of admissibility to conclude the universality of P if only P has been proved for all elements expressible by constructor terms since the generated sort is nothing but the closure with respect to constructor application and chains.

A more interesting example for infinite elements is that of streams. Streams may represent possibly infinite sequences of actions which are often used for the specification of distributed systems (see for example [Bro88]). In SPECTRUM they can be specified as below:

STREAM = {

sort Stream_Elem;

```
    .&. : Elem × Stream_Elem ⟶ Stream_Elem   prio 7:right;

    ft : Stream_Elem ⟶ Elem;
    rt : Stream_Elem ⟶ Stream_Elem;

    ft,rt strict; ft total;

    Stream_Elem freely generated by .&.;

    axioms ∀⊥ a : Elem, s : Stream_Elem in
        δ(a & s) = δ a;
        δ(a & s) ⇒ ft(a & s) = a;
        δ(a & s) ⇒ rt(a & s) = s;
    endaxioms;
}
```

The undefined element $\perp$ plays the role of the "empty" stream. The stream constructor .&. is not strict in its second argument. However, when applied to defined arguments, .&. always yields a defined result. Since $\perp$ represents the empty stream, it follows that the "first" function, denoted by ft, is total and moreover ft is also strict. The "rest" function, denoted by rt, is strict but not total. The reason for the latter is of course that

$$\mathsf{rt}(\mathsf{a} \mathbin{\&} \perp) = \perp .$$

Due to the phrase **freely generated by** the approximation ordering .$\sqsubseteq$. and therefore also the equality .=. on Stream_Elem are completely determined by those on Elem. From the axioms automatically added by **freely generated by** we may derive

$$\mathsf{s} \sqsubseteq \mathsf{t} = (\neg(\delta\ \mathsf{s}) \vee (\mathsf{ft}\ \mathsf{s} \sqsubseteq \mathsf{ft}\ \mathsf{t} \wedge \mathsf{rt}\ \mathsf{s} \sqsubseteq \mathsf{rt}\ \mathsf{t}));$$

which is the usual prefix ordering on streams. The rule for structural induction on Stream_Elem is

$$\frac{\begin{array}{l} H1 \vdash \forall^{\perp} s.\delta(P[s/ds]) \\ H2 \vdash P[\perp/ds] \\ H3 \vdash \forall^{\perp} s, a.\delta(a\&s) \wedge P[s/ds] \Rightarrow P[a\&s/ds] \end{array}}{H1, H2, H3 \vdash \forall^{\perp} s.P[s/ds]} \quad (P \text{ admissible in } ds)$$

Chapter 4

Predicates and non–continuous Functions

In SPECTRUM functions are assumed to be continuous with respect to $.\sqsubseteq.$. In particular all elements of carrier sets associated with functional sorts have to be continuous, because a function is assumed to be an object for computation and thus continuous.

SPECTRUM uses predicate logic as a core language for specifications and therefore needs the concept of predicates. For technical reasons we decided to use characteristic functions instead of relations to code predicates which are subsets in the semantics. Besides this we use the carrier set Bool also as the space of truth values which leads to an identification of boolean terms and formulae. Of course the characteristic function of a predicate is seldom continuous.

So far we have used the following builtin symbols for predicates:

δ "definedness",

$.=.$ "strong equality",

$.\sqsubseteq.$ "approximation with respect to definedness".

In very abstract specifications it may be convenient to introduce functions that are not continuous but have a range different from Bool. In order to avoid confusion we use the term *mapping* for functions that do not need to be continuous. This includes the special case of characteristic functions for predicates.

In SPECTRUM there is a clear distinction between mappings and their subset of continuous functions (objects to be implemented):

- In SPECTRUM the signature

  $\mathsf{f} : \mathsf{s1} \longrightarrow \mathsf{s2};$

  restricts the denotation of f to be continuous. The elements of $\mathsf{s1} \longrightarrow \mathsf{s2}$ are first class citizens in our algebras, and they may be passed to or returned by

higher-order functions. The sort expression $\mathsf{s1} \longrightarrow \mathsf{s2}$ denotes a carrier whose elements behave like functions and are candidates for an implementation.

- On the other hand, the signature

  g : s1 **to** s2;

  is used to specify a mapping. This means that it is always possible to syntactically distinguish mappings from what SPECTRUM calls functions. There are no constraints on mappings. They are only introduced for specification purposes, and they are not intended to be implemented. An isolated mapping symbol g is never a well-formed term. Thus, mappings may only occur in an application context. As a consequence of this syntactic restriction it is impossible to pass a mapping as an argument to a higher-order function. Another consequence is that we cannot introduce variables for mappings and cannot specify a mapping via λ–abstraction. All mappings that are available in a specification are those that are mentioned explicitly in the signature.

As an example consider existential equality:

$.=_{\exists}.$: s $\times$ s **to** Bool;
axioms $\forall^{\perp}$ x,y : s **in**
    $(\mathsf{x} =_{\exists} \mathsf{y}) = (\ \delta\ \mathsf{x} \wedge \delta\ \mathsf{y} \wedge \mathsf{x} = \mathsf{y})$;
endaxioms;

As a second example consider the non-continuous concatenation of streams:

.o. : Stream-Elem $\times$ Stream-Elem **to** Stream-Elem;
axioms $\forall^{\perp}$ a : Elem, s,t : Stream-Elem **in**
    $\perp \circ \mathsf{s} = \mathsf{s}$;
    $\mathsf{s} \circ \perp = \mathsf{s}$;
    $(\mathsf{a}\ \&\ \mathsf{s}) \circ \mathsf{t} = \mathsf{a}\ \&\ (\mathsf{s} \circ \mathsf{t})$;
endaxioms;

Chapter 5

The Sort System of SPECTRUM

The main goal of specification languages is to increase the reliability of software and hardware systems. The usage of sorts helps to achieve this goal by increasing the readability and understandability of specifications. In languages with a static sort concept the sort correctness of specifications can be determined by static program analysis. Therefore specification errors with respect to the sort correctness can be detected automatically in an early development phase.

In the previous chapters, we have only used basic sorts in our specification examples. In the last ten years, however, many powerful sort systems have been developed. The most well-known is *parametric polymorphism*, which can be found in many modern functional programming languages, for instance ML.

A sort concept which closes the gap between traditional monomorphic functions and polymorphic functions are *sort classes*[1]. The theory of sort classes was introduced by Wadler and Blott [WB89] and originally realized in the functional programming language Haskell. This polymorphic sort concept, which includes parametric polymorphism as a special case, has been adapted for the specification language SPECTRUM.

In this chapter the particular sorting facilities are introduced informally and explained through a number of examples.

5.1 Sort Constructors

As SPECTRUM has an expressive sort system it has a powerful sort language to build complex *sort expressions*. *Sort constructors* are function symbols on the sort level denoting domain constructors which take a (possibly empty) sequence of sorts as argument and yield a sort as result. The user can e.g. define a unary sort constructor List which, applied to an arbitrary sort, yields the sort of lists where the elements are of the sort given as parameter. Together with *sort variables*, sort constructors are the basic elements of the sort language, which is used to

[1] Also called type classes.

build complex sort expressions. Sort constructors that take no sort as argument like e.g. Nat are called *basic sorts*. Up to now we have only used basic sorts in our specification examples.

Note that in SPECTRUM the sort language cannot be mixed with the term language. Accordingly, sorts parameterized by values ("object dependent types") cannot be specified.

The sort constructor $\longrightarrow$ is predefined in SPECTRUM and denotes the function domain constructor. Because functions can only have one argument, n-ary tuples are used to simulate n-ary functions. The n-ary tuple domains are denoted by infinitely many predefined sort constructors $\times$., $\times\times$., ... These sort constructors bind stronger than $\longrightarrow$ and the prefix sort constructors bind stronger than the predefined infix sort constructors.

If the sort constructor List and the basic sort Nat are defined, the following examples are correct sort expressions, where α is a sort variable:

List (List Nat)
List $\alpha \times \alpha \longrightarrow$ List α

5.2 Polymorphism

In conventional languages with strong typing every function has a unique sort. These languages are called *monomorphic languages*. In polymorphic languages, values are allowed to have a set of valid sorts. These values are called *polymorphic values*. Therefore polymorphic functions can be applied to operands of different sort.

Strachey [Str67] distinguished between two major kinds of polymorphism. *Parametric polymorphism* is obtained when a function works uniformly on an infinite range of sorts with a common structure. *Ad-hoc polymorphism*, in contrast, allows a function to work differently on finitely many sorts not having necessarily a related structure. This kind of polymorphism is better known as *overloading* and can be simulated in SPECTRUM (see Section 5.3.2). This section, however, deals with parametric polymorphism only.

Parametric polymorphism is an easy-to-use alternative to parameterized specification modules, as the following example shows.

```
LIST = {

    --Sort constructor List
    sort List α;
```

```
    --Constructors
    [ ]: List α;
    cons: α × List α ⟶ List α;

    first: List α ⟶ α;
    rest: List α ⟶ List α;
    .⊕.: List α × List α ⟶ List α   prio 6:left;

    cons, first, rest, .⊕. strict;
    cons, rest, .⊕. total;

    List α freely generated by [ ], cons;

    axioms ∀ a : α, l, m : List α in
        first [ ] = ⊥;
        first(cons(a,l)) = a;

        rest [ ] = [ ];
        rest(cons(a,l)) = l;

        [ ]⊕l = l;
        cons(a,m)⊕l = cons(a,m⊕l);
    endaxioms;
}
```

The first difference to a conventional specification of lists is the definition of a sort constructor List. The sort variable α is used as a formal parameter, indicating the unarity of the constructor. This definition allows us to build arbitrary complex sort expressions (like List (List Nat)). The second difference is the declaration of polymorphic functions in the signature. These functions are declared to work on an infinite range of sorts. The common structure of the sorts is obtained with the help of the sort constructors in combination with sort variables.

As soon as the LIST-specification is combined with another specification module, the constructor List can be used to build complex sort expressions. The polymorphic functions can be applied to arbitrary operands matching the argument sort.

This can be achieved without instantiating a parameterized specification or renaming a function symbol. It is important for the design of proof support systems that the specification LIST is needed only once in such a system. It is not necessary to have various copies of the same specification in various instances. In particular, it is quite easy to prove theorems "generically" (i.e. in the general specification LIST).

In the polymorphic LIST-specification use is made of the fact that for every

sort there exists a built-in equality. It is more difficult to express parameterized specifications via parametric polymorphism if there are more specific requirements to the parameter, e.g. a partial order on the elements of a list. This can be achieved with the help of the sort class concept. See Section 7.5 for the modeling of more general parameterization in SPECTRUM.

As already mentioned, the polymorphism in SPECTRUM is an adaptation of the parametric ML-polymorphism. Unfortunately, within a specification language the uniform behaviour of polymorphic functions cannot be guaranteed. In contrast to functional languages, where a function is defined by its body and applied in its scope, within axioms there is no distinction between defining and applied occurrence of a function symbol. Every application of a function also defines properties of this function. Thus, a polymorphic function can be specified to behave differently on different sorts. Therefore in SPECTRUM the polymorphism cannot really be called "parametric". But nevertheless, this polymorphism is not a kind of ad-hoc polymorphism, because the functions work on an infinite range of sorts and therefore cannot be replaced by a finite set of monomorphic functions.

It relies on the discipline of the specifier to use SPECTRUM's polymorphism in a parametric way, where the polymorphic values are specified homogeneously on the sort parameter. The LIST-specification is an example of a homogeneous polymorphic specification. Homogeneous specifications are achieved by the generic sort of the variables in the axioms part.

5.3 Sort Classes

The parametric polymorphism offers a kind of sort abstraction for functions. The sort variables in the functionality of a function can be replaced by arbitrary sort expressions yielding a specialized function. Sometimes, however, one wants to restrict the replacement of sort variables to specific sort expressions. If the set of these sort expressions is finite, the problem can be solved by introducing a finite set of overloaded function symbols. But often one wants to restrict the range of a sort variable to an infinite set of sorts. A typical example of such a function is a decidable, weak equality relation. In contrast to the built-in strong equality, which is available for all sorts, but undecidable, a decidable equality must be restricted to appropriate sorts. On the one hand, this equality must not be applicable to the function space. On the other hand, a decidable equality can be defined for all lists, for which a decidable equality is available on the element sort.

This fine grained polymorphism is achieved by partitioning the sort universe. In SPECTRUM these partitions are called *sort classes*. The whole sort concept is similar to the type class concept of Haskell. However, the Haskell classes and the SPECTRUM classes are semantically not equivalent. With the help of sort classes

we can now specify a decidable equality relation in the following way:

```
Equality = {

    class EQ;

    .==. : α :: EQ ⇒ α × α ⟶ Bool;
    .==. strict total;

    axioms α :: EQ ⇒ ∀ a, b : α in
        (a == b) = (a = b);
    endaxioms;
}
```

With the help of the key word **class** we define a new sort class EQ. The functionality of the equality function .==. is restricted by a premise. This premise states, that in each application of .==. the sort variable can only be replaced by a sort expression of class EQ. The same premise can be found in the axioms part. The axioms are valid for all sorts of class EQ. The sort of the object variables must be of class EQ, because otherwise the application of the equality function .==. would result in a sort error.

Now we have defined a function that coincides with the strong equality on defined values and that works on all sorts of class EQ. But we have not defined any sort to be of a particular class. With the following expression we define the sort Nat to be of class EQ.

```
Nat :: EQ;
```

Now the equality function .==. can be applied to elements of sort Nat like in the following example:

```
fac = fix (λf : Nat ⟶ Nat . λx : Nat .
    if x == zero
        then succ zero
        else x * f(pred x)
    endif);
```

It is important to use this weak equality in the λ-abstraction. The built-in strong .=. cannot be used in this case, because .=. is an mapping and in SPECTRUM it is not allowed to apply mappings on λ-bound variables (see Section 4).

As already mentioned, the weak equality should be available on a list, if the weak equality is also available on the sort of the list elements. This can be achieved by the following expression:

```
List :: (EQ)EQ;
```

This means, that if the sort constructor List is applied to a sort of class EQ, the result is also of class EQ. It defines the argument and result class of the sort constructor List. If the sort Nat is of class EQ, the function .==. can now be applied to elements of the sort List Nat, List List Nat, and so on.

In the previous chapters we used sort classes only implicitly, because every sort constructor which is defined without class information has a default one. There is a predefined sort class CPO which is the default class of the arguments and the result of a sort constructor. The definitions

```
sort Nat;
sort List α;
```

imply

```
Nat :: CPO;
List :: (CPO)CPO;
```

If this is not desired, the default sort classes can be avoided as follows:

```
sort Nat :: EQ;
sort List :: (EQ)EQ;
```

But notice, that now the sort constructor List cannot be applied to sort Nat $\longrightarrow$ Nat, assuming Nat $\longrightarrow$ Nat is of class CPO but not of class EQ.

Normally every sort constructor is defined with the default class information. Further class information of a particular sort constructor are allowed and are only additional information, i.e. a sort constructor can be overloaded with class information, like in the following example:

```
Nat :: CPO;
Nat :: EQ;
List :: (CPO)CPO;
List :: (EQ)EQ;
```

This overloading of a sort constructor is the usual way to model the fact, that a function is available on a constructed sort, if the function is also available on the argument sort. Note that we allow for overloading basic sorts only if they have a least sort class (in our example EQ).

If we use a sort variable without giving this variable an explicit class information, the sort variable is also implicitly of class CPO, like in the specification LIST from Section 5.2.

5.3.1 Partial Order on Sort Classes

In SPECTRUM the user can define a partial ordering on sort classes, namely that a sort class is a subclass of another sort class. Semantically, this means that if a sort belongs to a class, it also belongs to all superclasses of this class. This implies that if a function is restricted to the sorts of a particular class, the function is also available on the sorts of all subclasses. By default every declared sort class is a subclass of CPO. The following example demonstrates this feature by extending the specification Equality (see Section 5.3).

```
    ⋮
class PO subclass of EQ;

.≤.: α::PO ⇒ α × α ⟶ Bool;
.≤. strict total;

axioms α::PO ⇒ ∀ a, b, c: α in
    a ≤ a;                          --reflexivity
    a ≤ b ∧ b ≤ c ⇒ a ≤ c;          --transitivity
    a ≤ b ∧ b ≤ a ⇒ a == b;         --antisymmetry
endaxioms;
    ⋮
```

In the first line a new sort class PO is defined to be a subclass of EQ. The next line defines an ordering symbol .≤. for all sorts of class PO. The properties of a partial ordering are specified in the axioms part. In the axiom specifying the antisymmetry of the ordering relation the fact that PO is a subclass of EQ is used, because the weak equality is applied on a sort of class PO. Since each sort of class PO is also of class EQ this application is well-sorted.

The specification of an ordering relation between sort classes is not restricted to the definition of a new sort class, but can take place independently. The same result as above also could have been obtained by writing:

```
class PO;
PO subclass of EQ;
```

The only difference is, that PO is now by default a subclass of CPO and additionally also a subclass of EQ. As EQ is a subclass of CPO, the semantics remains unchanged.

Now we want to use this specification to define an ordering on the natural numbers specified in Chapter 2.1. This can be achieved in the following way:

```
Nat::PO;

axioms ∀ x:Nat in
      x ≤ succ x;
endaxioms;
```

More examples for the use of sort classes can be found in the following chapters.

5.3.2 Overloading with Sort Classes

With the help of sort classes we are able to simulate ad-hoc polymorphism. In Section 5.2 we already explained that the parametric polymorphism can be misused as a kind of ad-hoc polymorphism if the functions are not specified homogeneously on the sort parameter. Since in case of parametric polymorphism the range of a sort variable is infinite, this leads necessarily to a highly underspecified function. If we restrict, however, the range of sort variables by sort classes we can control the degree of overloading exactly. Because this kind of overloading is explicit it can be fully controlled by the specifier. Of course, a symbol cannot be overloaded with functions of different arities. But, anyhow, in the era of graphic displays overloading should be used only in suitable cases.

A typical example for a wise application of overloading are arithmetical operations, like .+., .*., .−. and ./.. These operations are overloaded with the help of sort class NUM. The specification Numericals in Appendix E contains the signature and a few axioms for these operations. In the specification Naturals the operations are specified for Nat (see Appendix E). In the following specification we now define the operations for Int.

```
Integers = {enriches Standard_Lib;

      sort Int :: NUM;

      --generator functions
      izero : Int;
      isucc : Int ⟶ Int;
      ipred : Int ⟶ Int;

      isucc, ipred strict total;

      Int generated by izero, isucc, ipred;
```

```
    axioms x,y : Int in

        isucc x = isucc y ⇔ x = y;
        ipred (isucc x) = x;
        isucc (ipred x) = x;

        --addition
        x + izero = x;
        x + isucc y = isucc(x + y);

        --subtraction
        (x + y) − y = x;

        --multiplication;
        x ∗ izero = izero;
        x ∗ isucc y = x + x∗y;

        --division
        y ≠ izero ⇔ δ(x/y);
        y ≠ izero ⇒ y ∗ (x/y) ≤ x ∧
                    (succ(x−y) ≤ y ∗ (x/y) ∨ succ(x+y) ≤ y ∗ (x/y));

        --Ordering
        x ≤ isucc x;

    endaxioms;
}
```

5.4 Sort Inference

SPECTRUM has a *static sort system*, i.e. the sort of every expression can be determined by static analysis. A static sort system has the advantage that simple specification errors with respect to sort correctness are detected automatically in an early specification phase, before a time-consuming development is started.

The sort language of SPECTRUM has two levels. Therefore testing the sort correctness of a specification is a two-level process. On the level of sort expressions a *sort class checker* tests the well-formedness of sort expressions. In most cases this will be only an arity check, because normally every sort constructor has at least the class information (CPO,...,CPO)CPO.

On the object level, a *sort inference system* infers the most general sort of every expression (including the "unsorted" variables), while still checking the sort correctness of the expressions. The system accepts even a nearly sort-free spec-

ification, while internally generating a fully sorted specification from the input and therefore combines the advantages of static sorting with the convenience of unsorted languages (like Lisp). Only for the identifiers in a signature explicit sort declarations are required. In the specification Equality, for example, the sort information as well as the class information can be omitted in the axioms part. The sort inference system infers exactly this information.

Note that sometimes variables must be sorted explicitly as in the following example which is an extension of the partial ordering specification from Section 5.3.1.

$\vdots$

class TO **subclass of** PO;

axioms α::TO $\Rightarrow \forall$ a,b: α **in**
    a$\leq$b $\vee$ b$\leq$a; $--$ *Totality*
endaxioms;

$\vdots$

If we omit the sort and class information, the sort inference system infers the following most general information

α::PO $\Rightarrow \forall$ a,b:α

because the ordering relation can be applied to all those values. As we want to specify a more specific case, we must restrict the axiom by explicitly giving a sort and class information.

Chapter 6

Derived Constructs

For the practical use of SPECTRUM it is indispensable to introduce adequately chosen syntactic shorthands and to supply a standard library of commonly used data types with their characteristic functions (e.g. natural numbers, characters).

Syntactic shorthands (or derived operators) are constructs which do not have a direct algebraic semantics. Instead they are syntactically checked and expanded into SPECTRUM code which could be written without them.

6.1 Semantics of strict, total and strong

In this section we define the expansion of the axioms **strict**, **total** and **strong**. They are abbreviations for axioms about functions and mappings and their definitions are given in the following.

Definition 6.1 Expansion of the strict construct

Let $\mathsf{f}{:}s_1 \longrightarrow s_2$ be a function. We distinguish two cases:

- s_1 is a product $s_{11} \times \ldots \times s_{1n}$.
  In this case the phrase 'f **strict**' is expanded to the following axiom:

  $\forall^{\perp} \mathsf{x}_1, \ldots, \mathsf{x}_n.\ \neg(\delta\ \mathsf{x}_1) \vee \ldots \vee \neg(\delta\ \mathsf{x}_n) \Rightarrow \neg(\delta(\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n)))$;

  If the user specifies 'f **strict in** $(p_1, \ldots, p_m)$' such that $1 \leq p_i \leq n$ for $1 \leq i \leq m$ then the premise of the above axiom is restricted to the positions p_i and we expand to:

  $\forall^{\perp} \mathsf{x}_1, \ldots, \mathsf{x}_n.\ \neg(\delta\ \mathsf{x}_{p_1}) \vee \ldots \vee \neg(\delta\ \mathsf{x}_{p_m}) \Rightarrow \neg(\delta(\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n)))$;

- s_1 is not a product.
  In this case the phrase 'f **strict**' is expanded to the axiom:

  $\forall^{\perp} \mathsf{x}.\ \neg(\delta\ \mathsf{x}) \Rightarrow \neg(\delta(\mathsf{f}\ \mathsf{x}))$

□

Definition 6.2 Expansion of the total construct

Let f:$s_1 \longrightarrow s_2$ be a function. We distinguish two cases:

- s_1 is a product $s_{11} \times \ldots \times s_{1n}$.
  In this case the phrase '**f total**' is expanded to the axiom:

  $\forall^{\perp} \mathsf{x}_1, \ldots, \mathsf{x}_n.\ \delta\ \mathsf{x}_1 \wedge \ldots \wedge \delta\ \mathsf{x}_n \Rightarrow \delta(\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n))$;

- s_1 is not a product.
  In this case the phrase '**f total**' is expanded to the axiom:

  $\forall^{\perp} \mathsf{x}.\ \delta\ \mathsf{x} \Rightarrow \delta(\mathsf{f}\ \mathsf{x})$;

□

Definition 6.3 Expansion of the strong construct

Let g:s_1 **to** s_2 be a mapping. Here we don't need to consider several cases and simply define the expansion of '**g strong**' to be:

$\forall^{\perp} \mathsf{x}.\ \delta(\mathsf{g}\ \mathsf{x})$;

□

6.2 Semantics of freely generated by

As already indicated in Section 3.4 the phrase **freely generated by** is just a macro for axioms ensuring that different constructors generate different elements and all constructor functions are injective. We sketch the semantics by an example. For further study we refer to [Pau87].

First we introduce the recursive sort Seq α with two constructor functions and a generation principle.

SEQ = **{**

sort Seq α;

−− Constructor functions
empty : Seq α;
cons : $\alpha \times$ Seq $\alpha \longrightarrow$ Seq α;

Seq α **freely generated by** empty, cons;

}

In the following a semantic argumentation is used. To simplify the explanation syntactic identifiers are used instead of their semantic representation. By **generated by** we mean that each element of Seq α can be generated by iterating the application of the constructor functions. This yields an inductive structure. **freely generated by** in addition means that different constructors generate different elements and all constructor functions are injective. Technically this can be achieved by constructing the domain denoted by Seq α as the sum of two domains with two injection functions. This injection functions are represented by:

empty : Seq α;
cons : $\alpha \times$ Seq $\alpha \rightarrow$ Seq α;

Freely generated structures ensure recursive function definition by pattern matching to be sound. The existence of a functional which distinguishes the different cases of a sum leads to those structures. Therefore for sequences the Seq_When functional is introduced. The **freely generated by** statement is expanded to the **generated by** statement and the Seq_When functional, which yields a result of an arbitrary sort β. Every freely generated sort has a specific $When$ functional. This functional makes a distinction on the given argument (of the generated sort) and applies a different function for every constructor. When the argument is constructed by $cons_1$ then apply the function f_1. When the argument is constructed by $cons_n$ then apply the function f_n. Therefore the sort of the $When$ functional depends on the constructors.

In the example of SEQ $n = 2$ and the constructors are empty and cons. The functional Seq_When has the following definition:

Seq_When : $\beta \rightarrow (\ \alpha \times$ Seq $\alpha \rightarrow \beta\ ) \rightarrow$ Seq $\alpha \rightarrow \beta$;

axioms $\forall$ x : α, s : Seq α, $f_1 : \beta$, $f_2 : \alpha \times$ Seq $\alpha \rightarrow \beta$ **in**
Seq_When (f_1) (f_2) $(\perp) = \perp$;
Seq_When (f_1) (f_2) (empty) $= f_1$;
Seq_When (f_1) (f_2) (cons(x, s)) $= f_2$(x, s);
endaxioms;

With this function we can define other functions like first, rest and is_empty. Furthermore we can show some properties of the generated sort.

first : Seq $\alpha \rightarrow \alpha$;
rest : Seq $\alpha \rightarrow$ Seq α;
is_empty : Seq $\alpha \rightarrow$ Bool;

axioms
first = Seq_When($\bot$)(λ(x,s).x);
rest = Seq_When($\bot$)(λ(x,s).s);
is_empty = Seq_When(true)(λ(x,s).false);
endaxioms;

The monotonicity of is_empty and the facts that true $\not\sqsubseteq$ false and false $\not\sqsubseteq$ true allow us to deduce the distinctness axioms by contradiction:

empty $\not\sqsubseteq$ cons(x, s)
cons(x, s) $\not\sqsubseteq$ empty

The invertability

cons(x, l) $\sqsubseteq$ cons(y, k) $\Leftrightarrow$ x $\sqsubseteq$ y $\wedge$ l $\sqsubseteq$ k

can be deduced with the monotonicity of first, rest and cons.

To summarize we give a specification SEQ' without the **freely generated by** statement which is equivalent to the specification SEQ:

SEQ' = **{**

sort Seq α;

−−Constructor functions
empty : Seq α;
cons : $\alpha \times$ Seq $\alpha \rightarrow$ Seq α;

Seq α **generated by** empty, cons;

−−WHEN functional for Sequences
Seq_When : $\beta \rightarrow (\ \alpha \times$ Seq $\alpha \rightarrow \beta\ ) \rightarrow$ Seq $\alpha \rightarrow \beta$;

axioms $\forall$ x : α, s : Seq α, $f_1 : \beta$, $f_2 : \alpha \times$ Seq $\alpha \rightarrow \beta$ **in**
Seq_When (f_1) (f_2) ($\bot$) = $\bot$;
Seq_When (f_1) (f_2) (empty) = f_1;
Seq_When (f_1) (f_2) (cons(x, s)) = f_2(x, s);
endaxioms;

}

6.3 Data Type

As in most functional languages a data type declaration is provided for introducing recursive sorts. For example:

```
data Tree α       = emptytr
                  | mktree(!node : α, !branches : Branches α)
and  Branches α   = emptybr
                  | mkbran(!first : Tree α, !rest : Branches α);
```

introduces two mutually recursive sort constructors: Tree and Branches.

The sort constructor Tree has two element constructors (or simply constructors): emptytr and mktree. The first one is a tree constant. The second one is strict in both arguments and constructs trees of the form mktree(n, br) from elements n : α, br : Branches α. Strictness is expressed by putting an exclamation mark ! on the strict position (default is lazy). Infix constructors can additionally be given priorities.

Beside the constructors, Tree also has by default two discriminators: is_emptytr and is_mktree. Their names are built by prefixing the constructor names with is_.

Optionally it is also allowed to declare selectors for every argument position of a constructor. Two selectors were declared for trees constructed by mktree: node and branches.

Similarly, associated with the sort constructor Branches are the element constructors emptybr and mkbran, the two discriminators is_emptybr and is_mkbran as well as the selectors first and rest.

The above declaration is equivalent to the following specification text:

```
--Introduced sort constructors
sort Tree α, Branches α;

--Tree constructors
emptytr : Tree α;
mktree :  α × Branches α ⟶ Tree α;
mktree strict total;

--Tree selectors
node : Tree α ⟶ α;
branches : Tree α ⟶ Branches α;
node, branches strict;

--Tree discriminators
is_emptytr : Tree α ⟶ Bool;
is_mktree : Tree α ⟶ Bool;
is_emptytr, is_mktree strict total;
```

```
--Branches constructors
emptybr : Branches α;
mkbran : Tree α × Branches α ⟶ Branches α;
mkbran strict total;

--Branches selectors
first : Branches α ⟶ Tree α;
rest : Branches α ⟶ Branches α;
first, rest strict;

--Branches discriminators
is_emptybr : Branches α ⟶ Bool;
is_mkbran : Branches α ⟶ Bool;
is_emptybr, is_mkbran strict total;

--Induction, Distinctness and Partial Order
Tree α, Branches α freely generated by emptytr, mktree, emptybr, mkbran;

--Axioms for selectors and discriminators
axioms
    --freely generated by also introduces the _when functionals:
    -- tree_when : β ⟶ (α × Branches α ⟶ β) ⟶ Tree α ⟶ β
    -- branches_when : β ⟶ (Tree α × Branches α ⟶ β) ⟶ Branches α ⟶ β
    --Tree selectors
    node     = tree_when(⊥)(λ(n,br).n);
    branches = tree_when(⊥)(λ(n,br).br);

    --Tree discriminators
    is_emptytr = tree_when(true)(λ(n,br).false);
    is_mktree  = tree_when(false)(λ(n,br).true);

    --Branches selectors
    first = branches_when(⊥)(λ(e,rt).e);
    rest  = branches_when(⊥)(λ(e,rt).rt);

    --Branches discriminators
    is_emptybr = branches_when(true)(λ(e,rt).false);
    is_mkbran  = branches_when(false)(λ(e,rt).true);
endaxioms;
```

The strictness and totality axioms for selectors and discriminators have only documentation purpose since they are already included in the definition of the

tree_when and branches_when functionals.

Selectors and discriminators could have been defined entirely without using the _when functionals. However, this would have required considerably more axioms, especially in cases where many constructors are defined for sort constructors.

Finally, unlike in functional languages, we do not allow to bury the defined sort in a sort expression on the right hand side of the **data** declaration. For example:

data Point = mkp(x: Int, mv: Int $\longrightarrow$ Point);

is not allowed because the defined sort Point is buried in the sort expression Int $\longrightarrow$ Point. In other words, we require the defined sort to appear only on the top level in the functionality of a constructor. This was not the case above because we get:

mkp : Int $\times$ (Int $\longrightarrow$ Point) $\longrightarrow$ Point;

when we expand the declaration. This restriction also applies in LCF (see [Pau87]) and assures that a structural induction rule can be generated for the defined sorts.

In a datatype declaration it is also allowed to use a context. For example writing:

```
data α :: EQ ⇒ Tree α       = emptytr
                            | mktree(!node : α, !branches : Branches α)
and             Branches α = emptybr
                            | mkbran(!first : Tree α, !rest : Branches α);
```

would restrict the α variable to range only over equality sorts in the signatures of the constructors, selectors, discriminators and the when functional. The new signatures are:

```
--Introduced sort constructors
sort Tree α, Branches α;

--Tree
emptytr : α :: EQ ⇒ Tree α;
mktree : α :: EQ ⇒ α × Branches α ⟶ Tree α;

node : α :: EQ ⇒ Tree α ⟶ α;
branches : α :: EQ ⇒ Tree α ⟶ Branches α;

is_emptytr : α :: EQ ⇒ Tree α ⟶ Bool;
is_mktree : α :: EQ ⇒ Tree α ⟶ Bool;
```

$--Branches$
emptybr : α :: EQ $\Rightarrow$ Branches α;
mkbran : α :: EQ $\Rightarrow$ Tree $\alpha \times$ Branches $\alpha \longrightarrow$ Branches α;

first : α :: EQ $\Rightarrow$ Branches $\alpha \longrightarrow$ Tree α;
rest : α :: EQ $\Rightarrow$ Branches $\alpha \longrightarrow$ Branches α;

is_emptybr α :: EQ $\Rightarrow$: Branches $\alpha \longrightarrow$ Bool;
is_mkbran α :: EQ $\Rightarrow$: Branches $\alpha \longrightarrow$ Bool;

$--When\ functionals$
tree_when : α :: EQ $\Rightarrow$ $\beta \longrightarrow (\alpha \times$ Branches $\alpha \longrightarrow \beta) \longrightarrow$ Tree $\alpha \longrightarrow \beta$
branches_when : α :: EQ $\Rightarrow$ $\beta \longrightarrow ($Tree $\alpha \times$ Branches $\alpha \longrightarrow \beta) \longrightarrow$ Branches $\alpha \longrightarrow \beta$

The axioms and the generation principle remain the same modulo the variable restriction for α.

6.4 Sort Synonyms

It is often convenient to introduce new sort names (or synonyms) for commonly used sort expressions. The new names can be chosen to be shorter and more mnemonic and therefore can considerably improve the readability of a specification. For example:

sortsyn String = List Char;
sortsyn Name = String;

declares the synonyms String and Name. Sort synonyms can also be parameterized over sort variables. For example:

sortsyn Symtab α = Stack Array α;

declares a generic symbol table by using generic stacks and arrays.

Unlike the **data** declaration, the **sortsyn** declaration neither defines a new sort, nor can be recursive. It is merely a syntactic shorthand which will be expanded before giving an algebraic semantics to the specification containing it.

As a consequence in a basic specification sort synonyms may only be used in places where sort expressions are allowed. For example, a sort synonym is not allowed in a **generated by** axiom.

6.5 Let and Letrec Expressions

6.5.1 Let Expressions

When specifying a function, it is often convenient to introduce local names for common subterms occurring in its body. This provides for an abstraction mechanism which is supported in SPECTRUM as in most functional languages via the **let** construct. For example, the following specification of the function f:

```
axioms ∀ i, j : Nat in
    f(i, j) = let
                a = i*i + 2*i*j + j*j
              in
                a*a + a + 1
              endlet;
endaxioms;
```

is much more readable than the following one:

```
axioms ∀ i, j : Nat in
    f(i, j) = (i*i + 2*i*j + j*j)*(i*i + 2*i*j + j*j) + i*i + 2*i*j + j*j + 1;
endaxioms;
```

The above **let** declaration introduces a local identifier a, bound to the term i*i + 2*i*j + j*j whose scope is the term a*a + a + 1.

The corresponding SPECTRUM translation is[1]:

```
(a*a + a + 1) [(i*i + 2*i*j + j*j) / a]
```

Every occurrence of the local variable a is replaced by its definition in the term a*a + a + 1.

This corresponds operationally to a β–reduction. However, the above declaration is *not* equivalent to:

```
(λa. a*a + a + 1) (i*i + 2*i*j + j*j)
```

This is because the **let** declaration allows to define local *polymorphic* terms. Consider the following specification:

```
axioms ∀ li : List Nat, lc : List Char in
    f(li, lc) = let
                  g = λl. length(l) * length(l)
                in
                  h( g(li) + g(lc), g(li) − g(lc) )
                endlet;
endaxioms;
```

[1] This is not SPECTRUM syntax but the usual substitution notation.

where h is an arbitrary function. The **let** declaration is equivalent with:

h(g li + g lc, g li − g lc) [(**λ**l. length l ∗ length l) / g]

By avoiding to declare the sort of the bounded variable l explicitly we give the sort system the freedom to infer different sorts for **λ**l. length l ∗ length l at different occurrences in h. For example the sort system infers:

(**λ**l. length l ∗ length l) : List Int ⟶ Nat

before applying it to li and:

(**λ**l. length l ∗ length l) : List Char ⟶ Nat

before applying it to lc. Since g stands for the above lambda terms it is equivalent to say that g was defined polymorphically and used with different typings in g li and g lc.

Note that avoiding to type l is *not* equivalent with declaring it to have a generic sort α as in:

g = **λ**l : α. length l ∗ length l;

In this case the sort system will bind α to List Int at the first application g li and hence will reject the second application g lc as not well typed.

6.5.2 Letrec Expressions

In SPECTRUM supports also local recursive declarations. For this purpose the **letrec** construct is provided. For example the polymorphic length function could be declared and used as follows:

```
letrec
     length = λl. if l == empty
                  then 0
                  else 1 + length tail l endif
in
     length lc ∗ length lc
endlet;
```

In order to give the translation, we first define the following functional:

```
F = λlength.λl. if l == empty
                then 0
                else 1 + length tail l endif;
```

The **letrec** declaration corresponds then to:

(length lc ∗ length lc) [fix F / length]

It is also possible to define mutually recursive functions locally, as in the following example:

```
letrec
      odd = λn. if n == 1
                then true
                else even(n-1) endif
and
      even = λn. if n == 0
                 then true
                 else odd(n-1) endif
in
      e(odd, even)
endlet;
```

In order to give the translation, analogous to the simple case, we first define a functional:

```
F = λ(odd, even). (
                    λn. if n == 1
                        then true
                        else even(n-1) endif,
                    λn. if n == 0
                        then true
                        else odd(n-1) endif
                    );
```

The **letrec** declaration corresponds then to:

e(odd, even) [π_1 fix F / odd , π_2 fix F / even]

where π_1 and π_2 are the standard polymorphic projection functions defined as:

$\pi_1 : (\alpha \times \beta) \rightarrow \alpha$;
$\pi_2 : (\alpha \times \beta) \rightarrow \beta$;

axioms
π_1 = λ(x, y). x;
π_2 = λ(x, y). y;
endaxioms;

6.5.3 If–Then–Else Expressions

The polymorphic function if_then_else_endif defined in SPECTRUM's built in specification (see Appendix D) is very often used in specifications. We therefore provide a mixfix notation for it:

if b **then** s **else** t **endif** is translated to if_then_else_endif(b, s, t)

6.6 Built–in Data Types

The frequently used numeric data types, the data type List and the sort synonym String are included in the standard library of the SPECTRUM language (See Appendix E). They are also provided with a more mnemonic notation as usually allowed in specifications.

6.6.1 Numeric Data Types

Specifications of numeric data types and in particular for Nat are given in the standard library in Appendix E. In order to improve the readability of numbers we allow to write:

1 **for** (succ 0)
2 **for** (succ (succ 0))
...
n **for** $\underbrace{\text{(succ (...(succ 0)...))}}_{n \text{ times}}$

6.6.2 The Data Type List

The polymorphic LIST specification given in the standard library defines a sort constructor List and the element constructors [] and cons. In order to improve the readability we allow lists to be written as:

[e_1, e_2, ... , e_n]

and translate them as:

cons(e_1, cons(e_2, ... cons(e_n, []) ...))

6.6.3 Strings

As already explained in Section 6.4, String is a sort synonym for List Char. In order to improve the readability we allow strings to be written as:

"a string"

and translate them to:

['a', ' ', 's', 't', 'r', 'i', 'n', 'g']

Chapter 7

Specifying in the Large

Up to now we have only discussed basic specifications. As already explained, these consist of three parts:

- A signature declaring the new sort classes, sorts and functions.
- A list of sort synonym declarations.
- A body consisting of axioms which state the properties of the new sort classes, sorts and functions[1].

Basic specifications are suited to describe small, "mind sized", abstract data types. However, it is more convenient to build large and complex specifications in a structured way, by combining and modifying smaller specifications. This facilitates construction, understanding, analysis and implementation of specifications. It also encourages their reuse.

Structuring is achieved by using so-called specification building operators which map a list of argument specifications into a result specification. Among these operators .+. has a direct, algebraic semantics. We describe it in Section 7.1. The operators **rename**, **hide**, **export** and **enriches** are derived operators. They have no direct algebraic semantics. Instead, specifications containing them are syntactically checked and normalized into specifications using only the operator .+. Derived operators are described in Sections 7.2–7.4.

Beside the specification building operators SPECTRUM provides additional facilities for specification manipulation. One can extract the signature of a specification by using the **SIG** operator, can name a renaming list and can directly declare hidden identifiers or signatures in the body of a specification. These facilities are described in Sections 7.2 and 7.3.

In SPECTRUM the user is also allowed to abstract from sorts or from specification expressions involving the above operators. In other words it is allowed to build parameterized specifications. These can be later instantiated with actual

[1] It is worth noting that **generated by** and **freely generated by** are also treated as axioms.

sorts in the first case or applied to actual specifications which "fit" the signatures of the formal parameters in the second case. We describe parameterization in Section 7.5.

Relations between specifications (for e.g. the constructor–implementation relation) are part of the methodology and not of the specification language. They will be described in a methodological setting.

7.1 Combination

Two specifications SP_1 and SP_2 are combined by writing $\mathsf{SP}_1 + \mathsf{SP}_2$. Combination is modeled by taking the union of signatures, of sort synonym declarations and of axioms.

Remember that sort synonyms are only shorthands and therefore we always expand them before giving an algebraic semantics. This means that models are defined wrt the expansion of the result specification. The models also satisfy the expanded argument specifications.

In the previous chapter we wrote a specification of partial orders. In practice however, a new specification is often built by using a library of specifications. Suppose the library contains the following specifications defining the reflexive, antisymmetric and transitive properties of relations:

Reflexive = {
    class PO;
    $.\leq. : \alpha :: \mathsf{PO} \Rightarrow \alpha \times \alpha \longrightarrow \mathsf{Bool}$ **prio** 6;
    $.\leq.$ **strict total**;
    axioms $\alpha :: \mathsf{PO} \Rightarrow \forall\ \mathsf{a} : \alpha$ **in**
        $\mathsf{a} \leq \mathsf{a}$;
    endaxioms;
}

Antisymmetric = {
    class PO;
    $.\leq. : \alpha :: \mathsf{PO} \Rightarrow \alpha \times \alpha \longrightarrow \mathsf{Bool}$ **prio** 6;
    $.\leq.$ **strict total**;
    axioms $\alpha :: \mathsf{PO} \Rightarrow \forall\ \mathsf{a}, \mathsf{b} : \alpha$ **in**
        $\mathsf{a} \leq \mathsf{b} \wedge \mathsf{b} \leq \mathsf{a} \Rightarrow \mathsf{a} == \mathsf{b}$;
    endaxioms;
}

Transitive = {
    class PO;
    $.\leq. : \alpha :: \mathsf{PO} \Rightarrow \alpha \times \alpha \longrightarrow \mathsf{Bool}$ **prio** 6;
    $.\leq.$ **strict total**;

```
    axioms α :: PO ⇒ ∀ a, b, c : α in
        a ≤ b ∧ b ≤ c ⇒ a ≤ c;
    endaxioms;
}
```

We can combine them to obtain the specification of partial orders as follows[2]:

POrder = Reflexive + Antisymmetric + Transitive

The specification POrder is equivalent to the following specification POrder'.

```
POrder' = {
    class PO;
    .≤. : α :: PO ⇒ α × α → Bool   prio 6;
    .≤. strict total;
    axioms α :: PO ⇒ ∀ a, b, c : α in
        a ≤ a;
        a ≤ b ∧ b ≤ a ⇒ a == b;
        a ≤ b ∧ b ≤ c ⇒ a ≤ c;
    endaxioms;
}
```

Note that the meaning of .≤. in the resulting specification POrder is different from the meaning of .≤. in the argument specifications (it satisfies more axioms). In other words the class of models satisfying POrder is, for each argument specification, different from the class of models satisfying that argument. In general, neither the meaning of SP_1 nor of SP_2 is protected in $SP_1 + SP_2$. The generation of proof obligations which ensures this protection is handled on the methodology level.

7.2 Renaming

As already pointed out, the .+. operator combines the properties of an identifier which occurs in both argument specifications. This effect was exploited in building the specification POrder. To avoid this effect by keeping identifiers from coinciding, to force it by making identifiers to coincide or to give a more appropriate name to an identifier, SPECTRUM provides the **rename** operator.

For example, suppose Reflexive', Antisymmetric' and Transitive' were specified by different teams which used the sort classes REF, ANT, TRA for PO and the relations .ref., .ant., .tra. for .≤.. To obtain the partial order specification we would have to make the sort class and function identifiers to coincide by using renaming before combining the specifications:

[2]Brackets were dropped because + is associative.

```
POrder" = rename Reflexive'       by [ REF  to PO, .ref.  to .≤.]
        + rename Antisymmetric' by [ ANT to PO, .ant. to .≤.]
        + rename Transitive'      by [ TRA to PO, .tra. to .≤.]
```

Renaming can be used on all identifiers occurring in a signature (i.e. on sort classes, sorts, and functions) and on sort synonyms. Renaming of sort classes, sorts and sort synonyms automatically updates sort constructor, function and sort synonym declarations. This assures the well definedness of both the signature and the sort synonym declarations.

In order to increase the flexibility of the renaming operation we allow to introduce a name for a renaming list. For example we could also have written:

```
TPo = [ REF to PO, ANT to PO, TRA to PO,
        .ref. to .≤., .ant. to .≤., .tra. to .≤. ];

POrder" = rename Reflexive' by TPo
        + rename Antisymmetric' by TPo
        + rename Transitive' by TPo
```

Note that renaming a identifier not occurring in the specification's signature leaves the specification unchanged.

Formally, every specification containing **rename** is normalized to a specification without rename. As a consequence the semantics of **rename** is entirely embedded in the static semantics of the SPECTRUM language. Renaming takes place before the expansion of the sort synonyms.

7.3 Export and Hiding

When building large, structured specifications it is very important to be able to control the scope of identifiers. Two operators are provided for this purpose: **export** and its complement **hide**.

By default the scope of identifiers is not limited. More precisely, every identifier occurring in a specification is visible in a + operation to the other specification. We can restrict the scope of an identifier to the body of a specification by hiding that identifier outside the specification.

As with renaming, it is allowed to hide/export every sort synonym and every identifier occurring in a signature (i.e. sort classes, sorts, and functions). In order to obtain a well defined signature and well defined sort synonym declarations, hiding a sort class, a sort or a sort synonym automatically hides every sort constructor, sort synonym or function which used them in their declaration.

In the following example:

```
NAT* = export Nat, zero, succ, .*. in NAT'
```

we restrict the scope of all identifiers except zero, succ and $*$ to the body of NAT*. The same result could have been obtained also by writing:

NAT* = **hide** pred, $.\leq.$, $.<.$, $.+.$, $.-.$, div, mod **in** NAT'

The choice between these operators depends only on the number of identifiers which have to remain visible.

Sometimes it is useful to hide or export an entire signature. For such purposes **SIG**(SP) is also allowed in the position of an identifier. **SIG**(SP) is also allowed in a basic specification.

Hiding identifiers with **hide** is normally performed when *using* a specification. However, it is often desirable to explicitly declare identifiers or entire specifications as auxiliary when *writing* a specification. For example:

NAT* = **{**
−−The NAT signature;
pred, $.\leq.$, $.<.$, $.+.$, $.-.$, div, mod **hidden**;
−−The NAT axioms
}

is equivalent to the above specification. Every identifier or signature which can be hidden by using **hide** can be declared **hidden**. The formal treatment of **hide**, **export** and **hidden** is analogous to the one adopted in PLUSS [Bid89]. Instead of removing the identifiers from a specification, hiding merely renames them by fresh identifiers not accessible to the specifier[3]. This renaming not only restricts the use of the identifiers but also avoids unintended clashes with other visible or hidden identifiers. This is analogous to an existential binding. Moreover, like existentially quantified formulas, axioms involving hidden identifiers can be used to reason about the enclosing specifications. In this way we avoid the technical problems implied by using a calculus on structured specifications as for e.g. in [SW83].

Finally, note that the algebraic (or logical) semantics of a specification is not changed by hiding. More precisely, the visibility control mechanism is relegated from the algebraic to the static semantics of the SPECTRUM language.

7.4 Enrichment

The process of building specifications hierarchically, by adding new sort and function symbols together with their corresponding axioms to a given specification, can be expressed as follows:

SP' = SP + **{** **SIG**(SP); ... new spec. text ... **}**

[3]As a consequence, the user could not have written directly the specification by using only **rename**.

Since this not very user friendly, SPECTRUM offers the following shorthand:

SP' = { **enriches** SP; ... new spec. text ... }

Because the new spec. text alone is not a specification, the enrich sentence is written as the first line *between* the curly brackets.

For example we can obtain the theory of *total orders* using the above theory of partial orders:

```
TOrder = {
    enriches rename POrder by [ PO to TO ];
    axioms α :: TO ⇒ ∀ a, b : α in
        a ≤ b ∨ b ≤ a;
    endaxioms;
}
```

We could have reused the code of POrder also by declaring the sort class TO to be a subclass of PO i.e. by establishing a semantical relationship between the two sort classes:

```
TOrder' = {
    enriches POrder;
    class TO subclass of PO;
    axioms α :: TO ⇒ ∀ a, b : α in
        a ≤ b ∨ b ≤ a;
    endaxioms;
}
```

By using **enriches** and **hidden** in conjunction it is possible to declare entire specifications as auxiliary. For example, the specification:

```
SP = {
    enriches SP1 + SP2;
    SIG(SP1) hidden;
    ...
}
```

uses the specifications SP_1 and SP_2 but does not export SP_1.

7.5 Parameterization

7.5.1 Parameterization with Sort Classes

A very important abstraction mechanism when writing specifications is sort abstraction. A rough or untyped version of this kind of abstraction is given by

parametric polymorphism. When combined with user defined sort constructors it gives in many cases an elegant alternative to the well known parameterization mechanisms from the algebraic community.

An example of this kind of abstraction was the polymorphic list specification from Section 5.2. In this specification, the sort variable α ranges over all possible sorts. As a consequence, every specification importing LIST, has automatically for each sort s a list List s together with the corresponding functions for lists. No renaming and no multiple copies of the list specification are necessary. In fact, the sort constructor List, taking sorts into sorts, is a more intuitive explanation of parameterization as the pushout or the functorial semantics. Moreover, a specification parameterized in this way is a usual specification and no special mechanism is needed in order to give it a meaning.

Although very simple and elegant, parametric polymorphism alone is rarely desired in practice. Normally, we do not want to abstract (or quantify) over all possible sorts, but only over those which satisfy some (interface) requirements.

For example, suppose we want to define and implement a predicate $.\in.$ on lists. Since $.\in.$ is not even monotonic on non flat sorts we would like to restrict its domain.

But sort restriction is exactly what sort classes are doing for us. This observation was also used to control the degree of overloading. By adding sort classes we shift from an "unsorted" to a "sorted parametric polymorphism". For example the LIST specification becomes:

```
LIST' = {
      enriches LIST;
      .∈. : α :: EQ ⇒ α × List α ⟶ Bool;
      .∈. strict total;
      axioms α :: EQ ⇒ ∀a, x : α, l : List α in
            ¬(a ∈ []);
            a ∈ cons(x,l) ⇔ (a == x) ∨ a ∈ l;
      endaxioms;
}
```

We use here the sort class EQ to restrict the range of the sort variable α. Which sorts actually belong to the sort class EQ is explicitly controlled by the specifier. Consequently, the specifier has control over the possible instances of the polymorphic list operations. In the following example:

```
ListUser = {
      enriches LIST' + CHAR + NAT ;
      Char :: EQ;
      Nat  :: EQ;
      Bool :: EQ;
}
```

the .$\in$. predicate is available for List Char, List Nat and List Bool beside the usual functions on lists. If one of the sorts Char, Nat or Bool was already declared to belong to the sort class EQ the corresponding declaration in ListUser would have been superfluous.

Since Char, Nat and Bool are all flat sorts owning a boolean equality function the specification ListUser is consistent. The generation of proof obligations is not part of the language but part of the methodology. When supporting the language with an interactive theorem prover like ISABELLE for example, proof obligations can be automatically generated and added to the list of goals to be proven.

The parameterization mechanism described above is very flexible. For example we can now refine the list specification to lists with a minimum function easily by writing:

MinLIST = {
  enriches TOrder + LIST';
  min : α :: TO $\Rightarrow$ List $\alpha \longrightarrow \alpha$;
  min **strict**;
  axioms α :: TO $\Rightarrow$ $\forall$ e : α, s : List α **in**
    s $\neq$ [] $\Rightarrow$ min(s) $\in$ s $\wedge$ (e $\in$ s $\Rightarrow$ min(s) $\leq$ e);
  endaxioms;
}

In this example it is important that TO is a subclass of EQ i.e. that every sort in TO is also contained in EQ[4]. Consequently, the polymorphic list functions also work properly when restricted to the sorts in TO i.e. we can reuse them. On the other hand, the function min is only useful on lists over total orders. We therefore restrict its domain.

In conclusion, the addition of sort classes and of a subclass relation significantly improves the parametric polymorphism and increases the degree of specification reuse.

7.5.2 Classical Parameterization

Parametric polymorphism combined with sort classes i.e. "sorted parametric polymorphism" is powerful enough to model most of the classical examples for parameterized specifications. In all these examples sort abstraction is done only over one sort or in other words the theories have only one sort of interest.

However, sometimes we want to abstract not only from one sort but from n–tuples of sorts. A typical example is the theory of vector spaces. Here we abstract from tuples of sorts of the form (α, β) :: scalars $\times$ vectors which satisfy the vector space properties. There are also cases in which we want to abstract from sort constructors. A typical example is the container specification:

[4]If this is not the case, then a subclass declaration has to be written.

```
CONTAINER = {
    sort Elem, Container;
    cons : Elem × Container ⟶ Container;
    nth  : Container × Nat ⟶ Elem;
    len  : Container ⟶ Nat;
        --Container axioms
}
```

This specification has as instances for example lists and vectors.

For such cases, in order to keep the sort class mechanism simple, we provide another abstraction mechanism: specification abstraction. The basic idea is to designate some specifications occurring in a specification expression as parameters (similar to λ–abstraction for functions). Usually these parameters are the requirement theories. The specification abstraction obtained, is called "parameterized specification". We are now allowed to instantiate the parameterized specification with actual specifications which "fit" the formal parameters.

More precisely, a parameterized specification written in SPECTRUM consists of two parts: the *formal parameters part* describing the required properties for the actual parameters and the *body* describing how the actual parameters are used to build the result.

For example, a parameterized specification of the container theory is the following one:

```
PCONTAINER =
    param
        X = {sort Elem};
    body {
        enriches X;
        sort Container;
        cons : Elem × Container ⟶ Container;
        nth : Container × Nat ⟶ Elem;
        len : Container ⟶ Nat;
        --Container axioms
}
```

PCONTAINER can subsequently be applied to any specification if it is also indicated which sort in this specification corresponds to Elem. In general one has to give a mapping (a renaming list) from the signature of the formal parameter to the signature of the actual parameter if this is not an inclusion. Its form is identical to the one given in a renaming operation. As an example we can instantiate PCONTAINER with NAT as follows:

```
NatPcont = PCONTAINER(NAT via [Elem to Nat])
```

For specifications with more than one formal parameter, the actual parameters must satisfy the following consistency condition: the concatenation of their renaming lists must be a valid renaming list. The result of the above instantiation is:

NatContainer = PCONTAINER_body[NAT/X, Nat/Elem]

where PCONTAINER_body[NAT/X, Nat/Elem][5] is the body of PCONTAINER with NAT substituted for each occurrence of X and Nat substituted for each occurrence of Elem.

Using the parameterized container theory we can write the following parameterized specification:

```
MAP =
    param
        X = { sort X_Elem };
        Y = { sort Y_Elem };
        Z = PCONTAINER;
    body{
        enriches X + Y
                  + rename Z(X) by [ Container to X_Container,
                                     cons to X_cons,
                                     nth to X_nth, len to X_len ]
                  + rename Z(Y) by [ Container to Y_Container,
                                     cons to Y_cons,
                                     nth to Y_nth, len to Y_len ];
        map : (X_Elem ⟶ Y_Elem) × X_Container ⟶ Y_Container;
        --Map axioms
    }
```

We can subsequently instantiate this specification with actual specifications for X, Y and Z. It is important to note that the instances of parameterized specifications have to be legal specification expressions. As a consequence, we do not allow partial instantiations. Suppose we have given the parameterized specifications LIST, VECT of lists and vectors both having at least the PCONTAINER functions and satisfying its axioms. Then we could build for example the following instances:

```
ListMap = MAP(NAT via [X_Elem to Nat],
              NAT via [Y_Elem to Nat],
              LIST via [Container to List])

VectMap = MAP(NAT Via [X_Elem to Nat],
              NAT Via [Y_Elem to Nat],
              VECT via [Container to Vector])
```

[5] This is not SPECTRUM syntax.

Because NAT satisfies the trivial theory and LIST, VECT satisfy the container theory the instances have the properties we expect them to have. As with sort classes, the proof obligations are treated at the methodology level.

As probably noted, classical parameterization alone involves a lot of complications. First, we have to allow parameterized specifications as parameters and then we have to take care of name clashes. Furthermore, classical parameterization alone does not take advantage of the new facilities of SPECTRUM by requiring the specifications LIST and VECT to be parameterized in a classical way.

However, when combined with sort classes, parameterization becomes a simple and very powerful mechanism. For example, we can write a polymorphic version of containers as follows:

```
PCONTAINER' = {
    sort Container α;
    cons : α × Container α ⟶ Container α;
    nth  : Container α × Nat ⟶ α;
    len  : Container α ⟶ Nat;
    --Container axioms
}
```

We can now use PCONTAINER' to define the parameterized specification:

```
MAP' =
    param
        X = PCONTAINER';
    body{
        enriches X
        map : (α ⟶ β) × Container α ⟶ Container β;
        --Map axioms
    }
```

Subsequently, MAP' can be instantiated with the polymorphic versions of LIST and VECTOR easily by writing:

```
ListMap = MAP'(LIST via [Container to List])

VectMap = MAP'(VECT via [Container to Vector])
```

Obviously, the second version is easier to write and to understand as the first one. It also takes advantage of the new facilities of the language.

Finally, as with derived operators we do not give any autonomous semantics to parameterized specifications. Instead, we check if they are correctly applied and explain their meaning by substituting the arguments for the formal parameters in the specification expression representing the body.

Chapter 8

Executable Sublanguage

In SPECTRUM programs are developed by first giving an abstract requirement specification, which in general is purely descriptive. Using the development method of [Bro91] this specification is refined until we obtain a specification completely written in the so called *executable sublanguage* of SPECTRUM. In the following we refer to such specifications as *executable specifications*.

Executable specifications can be understood as functional programs[1]. More precisely this means that there is a (automatic) translation from an executable specification to a functional program whose denotational semantics lie in the specification's model class.

Functional programming languages can be divided into two main categories.

- In languages with eager evaluation strategy (such as ML, OPAL) all functions are strict.
- In languages with lazy evaluation strategy (such as LML, HASKELL) all functions are nonstrict.

In order to cope with both kinds of functional languages we have to give two somewhat different definitions of an executable specification. Of course, every functional target language requires a specific translator which respects the syntax of this language.

In the following we sketch the notion of executable sublanguage for both evaluation strategies. In this informal introduction we restrict ourselves to the case of basic specifications.

[1] Remember that the design of SPECTRUM is oriented towards the specification and development of functional programs.

8.1 Executable Sublanguage for Strict Target Languages

As an example for a strict target language in this section we choose the language ML. A specification is executable wrt ML if it has the following syntactic properties.

- All functions in the signature are declared to be strict.
- All sorts are defined using the **data** construct (with strict constructors).
- Every function f is defined using only axioms of the form $f(t_1, \ldots, t_n) = E$ where
  - the t_i are constructor patterns
  - the patterns in the axioms of one function do not overlap
  - E is a term containing no quantifiers, λ abstractions and mappings. Since λ abstraction is nonstrict in SPECTRUM it is not allowed in the executable sublanguage for strict target languages.
- The only sort classes that are used are CPO and EQ

As an example we look at a part of the specification NAT' in Section 3.1. We focus on the definition of $.\leq.$ (we call the prefix symbol le) and pred:

```
LENAT = {

    data Nat  =   zero
              |   succ(!Nat);

    strict;
    pred : Nat ⟶ Nat;
    le : Nat × Nat ⟶ Bool;

    axioms ∀ n, m : Nat in

        le( zero , n ) = true ;
        le( succ n , zero ) = false ;
        le( succ n , succ m ) = le( n , m ) ;
        pred( succ n ) = n ;

    endaxioms;
```

This specification may be translated into the following ML program. The help functions (is_zero, is_succ, ...) introduced by the expansion of **data** are omitted.

```
datatype  Nat  =   zero
              |   succ of Nat ;

fun  le( zero , n )     =    true
|    le( succ n , zero )     =    false
|    le( succ n , succ m )        =    le( n, m ) ;

fun pred( succ n ) = n ;
```

That the definition of pred is not total causes a good ML-compiler to generate a warning like "matches are not exhaustive". A call of the function pred(zero) will raise an exception.

8.2 Executable Sublanguage for Lazy Target Languages

As an example for a lazy target language we chose in this section the language HASKELL. A specification is executable wrt HASKELL if it has the following syntactic properties.

- No function in the signature is declared to be strict.
- All sorts are defined using the **data** construct and none of the constructors is declared to be strict.
- Every function f is defined using only axioms of the form $f(t_1, \ldots, t_n) = E$ where
  - the t_i are constructor patterns
  - the patterns in the axioms of one function do not overlap
  - E is a term containing no quantifiers and no mappings

Note that SPECTRUM's nonstrict λ abstraction is allowed in the axioms since the abstraction mechanism in HASKELL is nonstrict, too.

The above rules for the executable sublanguage for a lazy target language are less restrictive as those of Section 8.1. Therefore the class of specifications that are executable in a lazy language is bigger than the one for strict languages.

Chapter 9

Acknowledgement

The language presented here is influenced by the work of many researchers. In particular we thank the following colleagues for stimulating contributions: H. D. Ehrich, H. Ehrig, J. Guttag, J. Goguen, J. Horning, S. Jähnichen, B. Krieg-Brückner, J. Loeckx, M. Löwe, B. Möller, A. Poigné, P. Pepper, D. Sannella, H. Schwichtenberg and A. Tarlecki.
For comments on draft versions and stimulating discussions we like to thank S. Dick, H. Ehler, M. Gogolla, W. Grieskamp, J. Horning, C. Klein, J. Loeckx, M. Löwe, B. Möller, F. Nickl, C. Prehofer, W. Reif, B. Reus, B. Rumpe, V. Sabelfeld, D. Sannella, T. Santen, K. Spies, R. Steinbrüggen, W. Stephan, T. Streicher, A. Tarlecki, U. Wolter and J. Zeyer.
Thanks also go to the whole research group of M. Broy in Munich.

Bibliography

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, and Krieg-Brückner. *The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L.*, volume 183 of *L.N.C.S.* Springer, 1985.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification (ACM Press Frontier Series)*. Addison-Wesley, 1989.

[Bid89] M. Bidoit. *PLUSS, un langage pour le développement de spécifications algébriques modulaires.* PhD thesis, Université Paris-Sud, Orsay, 1989.

[Bro88] M. Broy. Requirement and design specification for distributed systems. *LNCS*, 335:33–62, 1988.

[Bro91] M. Broy. *Informatik und Mathematik.* Springer, 1991.

[BW82] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18:47–64, 1982.

[Cam89] Juanito Camilleri. The HOL system description, version 1 for HOL 88.1.10. Technical report, Cambridge Research Center, 1989.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constrains.* Springer, 1990.

[Gal86] Jean Gallier. *Logig for Computer Science.* Harper and Row, 1986.

[Gau86] M.-C. Gaudel. Towards structured algebraic specifications. *ESPRIT '85', Status Report of Continuing Work (North-Holland)*, pages 493–510, 1986.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical report, Digital, Systems Research Center, Paolo Alto, California, 1985.

[GM87] J.A. Goguen and J. Meseguer. Order–sorted algebra solves the constructor–selector, multiple representation and coercion problems. In *Logic in Computer Science*, IEEE, 1987.

[Gog76] M. Gogolla. Partially ordered sorts in algebraic specifications. In B. Courcelle, editor, *Proc. 9th CAAP 1984, Bordeaux*. Cambridge University Press, 1976.

[GR93] Radu Grosu and Franz Regensburger. The logical framework of SPECTRUM. Draft, 1993.

[Gro91] OPAL Language Group. The programming language OPAL. Technical Report 91-10, Technische Universität Berlin, 1991.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, May 1992.

[HMM86] R.W. Harper, D.B. MacQueen, and R.G. Milner. Standard ML. *Report ECS-LFCS-86-2, Univ. Edinburgh*, 1986.

[JKKM88] J.-P. Jouannaud, C. Kirchner, H. Kirchner, and A. Megrelis. OBJ: Programming with equalities, subsorts, overloading and parameterization. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *Algebraic and Logic Programming*, pages 41–53. Akademie-Verlag Berlin, 1988.

[KBH91] B. Krieg-Brükner and B. Hoffmann, editors. *PROgram development by SPECification and TRAnsformation: Vol. I: Methodology, Vol II: Language Family, Vol III: System.* PROSPECTRA *Report M.1.1.S3-R-55.2, -56.2, -57.2.* (to appear in LNCS), 1991. Universität Bremen (1990).

[LL88] T. Lehmann and J. Loeckx. The specification language OBSCURE. In D. Sanella and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, pages 131–153. LNCS 332, 1988.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[Mit90] John C. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. Elsevier Science Publisher, 1990.

[Möl82] B. Möller. *Unendliche Objekte und Geflechte*. PhD thesis, Technische Universität München, September 1982.

[Nip91] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[Pau87] L.C. Paulson. *Logic and Computation, Interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[SNGM89] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In *Resolution of Equations in Algebraic Structures*. Academic Press, 1989.

[Sok89] S. Sokolowski. Applicative high-order programming or Standard ML in the battlefield. Technical Report MIP 8908, Universität Passau, Lehrstuhl für Informatik, February 1989.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[Str67] C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, 1967.

[SW83] D.T. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. Technical Report CSR-131-83, University of Edinburgh, Edinburgh EH9 3JZ, September 1983.

[Tur85] D. A. Turner. Miranda — a non-strict functional language with polymorphic types. In Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer Verlag, 1985.

[WB89] Philip Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[Wir86] M. Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 43:123–250, 1986.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.*, chapter 13, pages 675–788. North-Holland, Amsterdam, 1990.

Appendix A

Concrete Syntax of SPECTRUM

A.1 Notational Conventions

The concrete syntax of SPECTRUM is presented as an EBNF-like grammar. The notations used are summed up below:

$[rhs]$	rhs is optional
$\{rhs\}^*$	zero or more repetitions of rhs
$\{rhs \;//\; sep\}^*$	zero or more repetitions of rhs separated by sep
$\{rhs\}^+$	one or more repetitions of rhs
$\{rhs \;//\; sep\}^+$	one or more repetitions of rhs separated by sep
$\{rhs\}$	grouping
$rhs_1 \mid rhs_2$	choice
$rhs_{\{\overline{rhs}\}}$	difference: elements generated by rhs except those generated by $\overline{rhs}$
terminal	terminal syntax is given in boldface
⟨nonterminal⟩	nonterminals are enclosed in angle brackets
⟨*nonterminal*⟩	emphasized nonterminals are not defined in the grammar but represent a non-printable letter of the ASCII character set or are given informally

Remark: It is important to distinguish the metasymbols [,], {, }, (,) and | introduced above from the corresponding terminal symbols **[**, **]**, **{**, **}**, (,) and | printed in boldface font.

A.2 Lexical Syntax

⟨spectext⟩ ::= {⟨lexeme⟩|⟨whitespace⟩|⟨comment⟩}*
⟨lexeme⟩ ::= ⟨charconst⟩|⟨num⟩|⟨string⟩|⟨alphanumid⟩
| ⟨symbolid⟩|⟨special⟩|⟨reserved⟩

Whitespace
⟨whitespace⟩ ::= ⟨*space*⟩|⟨*newline*⟩|⟨*carriage return*⟩|⟨*tab*⟩
| ⟨*formfeed*⟩|⟨*vtab*⟩

Syntactic Categories
⟨letter⟩ ::= **a**|**b**|**c**|**d**|**e**|**f**|**g**|**h**|**i**|**j**|**k**|**l**|**m**|**n**|**o**|**p**|**q**|**r**|**s**|**t**|**u**
| **v**|**w**|**x**|**y**|**z**|**A**|**B**|**C**|**D**|**E**|**F**|**G**|**H**|**I**|**J**|**K**|**L**|**M**
| **N**|**O**|**P**|**Q**|**R**|**S**|**T**|**U**|**V**|**W**|**X**|**Y**|**Z**
⟨digit⟩ ::= **0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**
⟨sym⟩ ::= **[]**|**!**|**#**|**%**|**&**|**$**|*****|**+**|**−**|**/**|**<**|**>**|**=**|**?**|**@**|**^**|**~**|**||**|**`**
| \ {⟨letter⟩}+
⟨spcl⟩ ::= **{**|**}**|**(**|**)**|**.**|**;**|**,**|**:**|**”**|****
⟨graph-spcl⟩ ::= ∀|∀⊥|∃|∃⊥|**λ**
⟨graph-sym⟩ ::= **δ**|⊥|⊑|¬|∨|∧|⟶|×|⇒|⇔|≠
⟨ext-graph-sym⟩ ::= ⟨*additional graphic symbols*⟩
⟨symbol⟩ ::= ⟨sym⟩|⟨graph-sym⟩|⟨ext-graph-sym⟩
⟨special⟩ ::= ⟨spcl⟩|⟨graph-spcl⟩
⟨alphanum⟩ ::= ⟨letter⟩|⟨digit⟩|´|_
⟨any⟩ ::= ⟨alphanum⟩|⟨symbol⟩|⟨special⟩|⟨*space*⟩|⟨*tab*⟩

Comments
⟨line-comment⟩ ::= −− {⟨any⟩}* ⟨line-end⟩
⟨line-end⟩ ::= ⟨*newline*⟩|⟨*carriage return*⟩|⟨*formfeed*⟩|⟨*vtab*⟩
⟨nest-comment⟩ ::= **(:** {⟨no-nest⟩|⟨nest-comment⟩}* **:)**
⟨no-nest⟩ ::= {⟨any⟩}*{{any}* {**(:**|**:)**} {⟨any⟩}*}
⟨comment⟩ ::= ⟨line-comment⟩|⟨nest-comment⟩

Character Constants
⟨char⟩ ::= ⟨letter⟩|⟨digit⟩|⟨sym⟩|⟨spcl⟩|_|⟨*space*⟩|⟨escapes⟩
⟨escapes⟩ ::= **\n**|**\t**|**\v**|**\r**|**\f**|**\a**|\\|\’|\”
⟨charconst⟩ ::= ´ ⟨char⟩ ´

String Constants

⟨string⟩ ::= " {⟨char⟩}* "

Natural Numbers

⟨num⟩ ::= ⟨digit⟩$_{\{\mathbf{0}\}}$ {⟨digit⟩}*

Identifiers

⟨alphanumid⟩ ::= {⟨alphanum⟩}$^{+}_{\{\langle char\rangle|\langle num\rangle|\langle reserved\rangle\}}$

⟨symbolid⟩ ::= {⟨symbol⟩}$^{+}_{\{\{\langle symbol\rangle\}^{*} -- \{\langle symbol\rangle\}^{*}|\langle reserved\rangle\}}$

⟨id⟩ ::= ⟨alphanumid⟩|⟨symbolid⟩|⟨num⟩|⟨charconst⟩|⟨string⟩

Reserved Words

⟨reserved⟩ ::= **enriches**|**export**|**in**|**hide**|**rename**|**SIG**|**to**|**let**
| **letrec**|**endlet**|**ALL**|**ALL+**|**EX**|**EX+**|**LAM**
| **if**|**then**|**else**|**endif**|**and**|**axioms**|**endaxioms**
| **data**|**strong**|**strict**|**total**|**freely**|**generated**|**by**
| **prio**|**sortsyn**|**class**|**subclass**|**of**|**sort**|**hidden**
| **param**|**body**|**via**|**!**|**[**|**]**

Remarks on implementations: In order to increase readability, the design of SPECTRUM makes use of nonstandard graphic symbols (i.e. symbols that are not defined in the ASCII standard) like ∀, ∃, ... Thus, an implementation of SPECTRUM has to provide at least all the graphic symbols used in the language definition. However, as we do not want to exclude (nongraphic) ASCII implementations completely[1], we allow for so-called *restricted implementations* (in contrast to the above mentioned *full implementations*) according to the following concept:

- A restricted implementation may safely leave out the symbols defined under ⟨graph-sym⟩, ⟨graph-spcl⟩ and ⟨ext-graph-sym⟩. Instead of the (not implemented) graphic symbols their ASCII representations according to table A.1 can be used.
- Any full implementation has to implement the symbols of ⟨graph-sym⟩ and ⟨graph-spcl⟩. In addition, it may provide arbitrarily many additional graphic symbols. In the above grammar, those symbols are represented by ⟨ext-graph-sym⟩. It is completely up to the implementor which additional graphic symbols are provided. The only restriction is that they have to be different from the ASCII symbols and the symbols defined in ⟨graph-sym⟩ and ⟨graph-spcl⟩.

[1]For example for hardware without graphic capabilities.

Graphic Symbol	ASCII-Representation
$\neg$	~
$\wedge$	&
$\vee$	|
$\Rightarrow$	=>
$\Leftrightarrow$	<=>
$\neq$	#
$\forall$	ALL
$\exists$	EX
$\forall^{\perp}$	ALL+
$\exists^{\perp}$	EX+
δ	DEF
$\sqsubseteq$	<<
$\perp$	UU
$\longrightarrow$	->
$\times$	*
$\boldsymbol{\lambda}$	LAM

Table A.1: ASCII-representation of graphic symbols

A.3 Contextfree Syntax

Specifying in the Large

⟨system⟩ ::= {⟨sys-part⟩}*
⟨sys-part⟩ ::= ⟨specid⟩ = ⟨specexp⟩
| ⟨morphid⟩ = ⟨sigmorph⟩
| ⟨abstrid⟩ = ⟨specabstr⟩

Signature Morphisms
⟨morph⟩ ::= ⟨morphid⟩ | ⟨sigmorph⟩
⟨sigmorph⟩ ::= [{⟨rename⟩ // ,}+]
⟨rename⟩ ::= ⟨opn⟩ **to** ⟨opn⟩
| ⟨sortcon⟩ **to** ⟨sortcon⟩
| ⟨sortsyn⟩ **to** ⟨sortsyn⟩
| ⟨classid⟩ **to** ⟨classid⟩
⟨morphid⟩ ::= ⟨id⟩

Structured Specifications

⟨specexp⟩ ::= ⟨aspecexp⟩
| ⟨aspecexp⟩ **+** ⟨specexp⟩ *(Union)*
| ⟨abstr⟩ **(** {⟨specexp⟩ [**via** ⟨morph⟩] // **,**}$^+$ **)** *(Application)*

⟨aspecexp⟩ ::= ⟨specid⟩
| ⟨specbody⟩ *(Basic Specification)*
| **{ enriches** ⟨specexp⟩ **;** ⟨decls⟩ **}** *(Enrichment)*
| **export** {⟨sigel⟩ // **,**}$^+$ **in** ⟨aspecexp⟩ *(Export)*
| **hide** {⟨sigel⟩ // **,**}$^+$ **in** ⟨aspecexp⟩ *(Hiding)*
| **rename** ⟨specexp⟩ **by** {⟨morphid⟩|⟨sigmorph⟩} *(Renaming)*
| **(** ⟨specexp⟩ **)**

⟨sigel⟩ ::= ⟨opn⟩
| ⟨sortcon⟩
| ⟨sortsyn⟩
| ⟨classid⟩
| **SIG (** ⟨specexp⟩ **)**

⟨specid⟩ ::= ⟨id⟩

Parameterized Specifications

⟨abstr⟩ ::= ⟨abstrid⟩ | ⟨specabstr⟩

⟨specabstr⟩ ::= **param** {⟨specid⟩ **=** ⟨specexp⟩ // **,**}$^+$
body ⟨aspecexp⟩

⟨abstrid⟩ ::= ⟨id⟩

Specifying in the Small

⟨specbody⟩ ::= **{** ⟨decls⟩ **}**

⟨decls⟩ ::= {⟨signature⟩ **;**|⟨axioms⟩ **;**}*

Signatures

⟨signature⟩ ::= **class** ⟨classid⟩ [**subclass of** {⟨classid⟩ // **,**}$^+$]
| ⟨classid⟩ **subclass of** {⟨classid⟩ // **,**}$^+$
| **sort** ⟨sortcon⟩{{⟨sortvar⟩}*|**::** ⟨classexp⟩}
| {⟨sortcon⟩ // **,**}$^+$ **::** ⟨classexp⟩
| **sortsyn** ⟨sortsyn⟩ {⟨sortvar⟩}* **=** ⟨sortexp⟩
| **SIG (** ⟨specexp⟩ **)**
| {⟨sigel⟩ // **,**}$^+$ **hidden**
| ⟨opns⟩ **:** [⟨context⟩] ⟨sortexp⟩ **to** ⟨sortexp⟩ [⟨prio⟩]
| ⟨opns⟩ **:** [⟨context⟩] ⟨sortexp⟩ [⟨prio⟩]

⟨classexp⟩ ::= [({⟨classid⟩ // ,}$^+$)] ⟨classid⟩
⟨prio⟩ ::= **prio** ⟨num⟩ [: {**left**|**right**}]

Sort Expressions

⟨sortexp⟩ ::= ⟨sortexp1⟩
| ⟨sortexp1⟩ → ⟨sortexp⟩ *(Functional Sort)*
⟨sortexp1⟩ ::= ⟨sortexp2⟩
| ⟨sortexp2⟩ {× ⟨sortexp2⟩}$^+$ *(Product Sort)*
⟨sortexp2⟩ ::= ⟨asort⟩
| ⟨sortcon⟩ {⟨asort⟩}$^+$ *(Applied Sort Constructor)*
| ⟨sortsyn⟩ {⟨asort⟩}$^+$ *(Applied Sort Synonym)*
⟨asort⟩ ::= ⟨sortvar⟩
| ⟨sortcon⟩ *(Basic Type)*
| ⟨sortsyn⟩
| **(** ⟨sortexp⟩ **)**

Sort Contexts

⟨context⟩ ::= {⟨scontext⟩ // ,}$^+$ ⇒
⟨scontext⟩ ::= {⟨sortvar⟩ // ,}$^+$ :: ⟨classid⟩

Axioms

⟨axioms⟩ ::= **axioms** [⟨varlist⟩] {[⟨axid⟩] ⟨exp1⟩ ;}* **endaxioms**
| **data** [⟨context⟩] {⟨datadecl⟩ // **and**}$^+$
| [⟨opns⟩] ⟨attrdecl⟩
| ⟨simplesorts⟩ [**freely**] **generated by** ⟨opns⟩
⟨axid⟩ ::= **{** ⟨id⟩ **}**
⟨simplesorts⟩ ::= {⟨sortcon⟩ {⟨sortvar⟩}* // ,}$^+$
⟨varlist⟩ ::= [⟨context⟩] {{∀|∀$^\perp$} ⟨opdecls⟩}$^+$ **in**
⟨opdecls⟩ ::= {{⟨id⟩ // ,}$^+$: ⟨sortexp⟩ // ,}$^+$
| {⟨id⟩ // ,}$^+$

Data Types

⟨datadecl⟩ ::= ⟨sortcon⟩ {⟨sortvar⟩}* = {⟨product⟩ // ‖}$^+$
⟨product⟩ ::= ⟨id⟩
| ⟨opn⟩ **(** {[!] [⟨id⟩ :] ⟨sortexp⟩ // ,}$^+$ **)** [⟨prio⟩]

Totality, Strictness and Strongness Axioms

⟨attrdecl⟩ ::= **strong**
| {**total** | ⟨strictdecl⟩}$^+$
⟨strictdecl⟩ ::= **strict** [**in** {⟨num⟩ | (⟨num⟩ {, ⟨num⟩}$^+$)}]

Terms

⟨exp1⟩ ::= ⟨exp2⟩
| ∀ ⟨opdecls⟩ . ⟨exp1⟩ *(∀-Quantification)*
| ∀$^\perp$ ⟨opdecls⟩ . ⟨exp1⟩ *(∀$^\perp$-Quantification)*
| ∃ ⟨opdecls⟩ . ⟨exp1⟩ *(∃-Quantification)*
| ∃$^\perp$ ⟨opdecls⟩ . ⟨exp1⟩ *(∃$^\perp$-Quantification)*
| **λ** ⟨pat⟩ . ⟨exp1⟩ *(**λ**-Abstraction)*

⟨exp2⟩ ::= ⟨exp3⟩
| ⟨exp2⟩ ⟨id⟩ [: ⟨asort⟩] ⟨exp1⟩ *(Infix-Application)*

⟨exp3⟩ ::= ⟨aexp⟩
| ⟨exp3⟩ ⟨aexp⟩ *(Prefix-Application)*

⟨aexp⟩ ::= ⟨opn⟩
| **if** ⟨exp1⟩ **then** ⟨exp1⟩ **else** ⟨exp1⟩ **endif**
| **let** ⟨defs⟩ **in** ⟨exp1⟩ **endlet**
| **letrec** ⟨defs⟩ **in** ⟨exp1⟩ **endlet**
| (⟨exp1⟩ {, ⟨exp1⟩}$^+$) *(Tuples)*
| (⟨exp1⟩) *(Grouping)*
| [{⟨exp1⟩ // ,}*] *(Lists)*
| ⟨aexp⟩ : ⟨asort⟩ *(Sorted Expression)*

⟨defs⟩ ::= {⟨id⟩ = ⟨exp1⟩ // **and**}$^+$

⟨pat⟩ ::= ⟨id⟩ [: ⟨sortexp⟩]
| (⟨id⟩ [: ⟨sortexp⟩] {, ⟨id⟩ [: ⟨sortexp⟩]}$^+$)

Identifiers

⟨classid⟩ ::= ⟨id⟩
⟨sortcon⟩ ::= ⟨id⟩$_{\{\times|*|\rightarrow|->\}}$
⟨sortsyn⟩ ::= ⟨id⟩$_{\{\times|*|\rightarrow|->\}}$
⟨sortvar⟩ ::= ⟨id⟩$_{\{\times|*|\rightarrow|->\}}$
⟨opn⟩ ::= ⟨id⟩ | .⟨id⟩.
⟨opns⟩ ::= {⟨opn⟩ // ,}$^+$

Appendix B

Models and Logic

The purpose of this appendix is to give an introduction to the semantics and the underlying logic of SPECTRUM. For a detailed description of the logical framework of SPECTRUM see [GR93].

From an abstract point of view SPECTRUM is nothing but a notation for predicate logic. More specifically SPECTRUM is a many sorted LCF–like logic (see [Pau87]) that allows functions as elements in carrier sets and supports polymorphism. Since SPECTRUM was designed as a specification language there are some conceptual details, motivated by methodological aspects, experimental ideas and also by personal taste, that influenced not only the syntax but also the semantics of SPECTRUM (e.g. identification of boolean terms and formulae lead to three–valued logic, concrete implementations of functional programming languages motivated the lifting of the function space). Therefore the logic of SPECTRUM differs in a few technical details from $PP\lambda$, the logic of computable functions as introduced by Dana Scott [Sto77]. Besides these minor differences there is one concept in SPECTRUM that is beyond the expressiveness of $PP\lambda$, that of sort classes. In the concrete language of SPECTRUM we use the notion 'sort class' but for the description of the semantics we adopt the convention of type theory and use the notion 'kind'.

This appendix on the semantics and logic of SPECTRUM is organized as follows. First we introduce polymorphic signatures with partially ordered kinds (B.1) and well–formed terms (B.2). Then we sketch the mathematical structures, the algebras we use for the interpretation of terms and formulae (B.3). Next we define the interpretation of terms in those algebras and conclude with the notion of satisfaction and models (B.4).

B.1 Signatures

As an abstraction from the concrete syntax a specification $S = (\Sigma, E)$ is a pair where $\Sigma = (\Omega, F, O)$ is a polymorphic signature and E is a set of Σ-formulae. We

now sketch the definitions and refer to [GR93] for a detailed presentation.

Definition 2.1 Sort Signature:

A sort–signature $\Omega = (K, \leq, SC)$ is an order sorted signature[1], where

- $(K, \leq)$ is a partial order on *kinds*,
- $SC = \{SC_{w,k}\}_{w \in (K \setminus \{map\})^*, k \in K}$ is an indexed set of *sort constructors* with monotonic functionalities i.e.:

$$(sc \in SC_{w,k} \cap SC_{w',k'}) \wedge (w \leq w') \Rightarrow (k \leq k')$$

A sort–signature must satisfy the following additional constraints:

- It is *regular, coregular* and *downward complete*. These properties[2] guarantee the existence of principal kinds.
- It includes the standard sort–signature (see below).
- All kinds except *map* and *cpo*, which are in the standard signature, are below *cpo* with respect to $\leq$. In other words, *cpo* is the top kind for all kinds a user may introduce.

□

Definition 2.2 The standard (predefined) sort–signature

The standard sort–signature:

$$\begin{array}{lll} \Omega_{standard} = & (& \{cpo, map\}, \emptyset, \\ & & \{\{\mathsf{Bool}\}_{cpo}, \\ & & \ \{\rightarrow\}_{cpo\ cpo,\ cpo}, \\ & & \ \{\mathbf{to}\}_{cpo\ cpo,\ map}, \\ & & \ \{\times_n\}_{\underbrace{cpo \ldots cpo}_{n\ times},\ cpo} \\ & & \} \\ &) & \end{array}$$

contains two kinds and four sort constructors (actually, we have for every natural number n a sort constructor $\times_n$):

- *cpo* represents the kind of *all complete partial orders*, *map* represents the kind of *all full function spaces*,

[1] Order kinded would be more precise; see [GM87, Gog76] for order sorted algebras.
[2] See [SNGM89] for a definition.

- Bool is the sort of booleans, $\longrightarrow$ is the constructor for lifted continuous function spaces, **to** is the constructor for full function spaces and $\times_n$ for $n \geq 2$ is the constructor for cartesian product spaces.

□

The sort–signatures together with a disjoint family χ of sort variables indexed by kinds (a *sort context*) allows us to define the set of sort terms.

Definition 2.3 Sort Terms:

$T_\Omega(\chi)$ is the freely generated order kinded term algebra over χ. □

Example 2.1 Some sort terms

Let $\mathsf{Set} \in SC_{cpo,\ cpo}$. Then:
$\mathsf{Set}\ \alpha \longrightarrow \mathsf{Bool},\ \mathsf{Bool} \times \mathsf{Bool} \in T_\Omega(\{\alpha\}_{cpo})$ □

The idea behind polymorphic elements is to describe families of non-polymorphic elements. In the semantics this is represented with the concept of the generalized cartesian product. For the syntax however there are several techniques to indicate this fact. E.g. in HOL ([Cam89]) the sort of a polymorphic constant in the signature is treated as a template that may be arbitrarily instantiated to build terms. This technique is also used for the concrete syntax of SPECTRUM. In the technical paper [GR93] we decided to make this mechanism explicit in the syntax too and introduced a binding operator Π for sorts and an application mechanism on the syntactic level. For a system with simple predicative polymorphism this is just a matter of taste. For a language with local polymorphic elements (ML–polymorphism) or even deep polymorphism such binding mechanisms are essential.

Definition 2.4 Π – Sort Terms:

$\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n.e \in T_\Omega^\Pi$ if:

- $e \in T_\Omega(\chi)$
- $\mathsf{Free}(e) \subseteq \{\alpha_1, \ldots, \alpha_n\}$
- $k_i \leq cpo \qquad$ for $k_i \in K$, $1 \leq i \leq n$

□

Note that the third condition rules out bound sort variables of kind *map*.

Example 2.2 Some Π – Sort Terms:

$\Pi\alpha : cpo.\mathbf{Set}\ \alpha \longrightarrow \mathsf{Bool} \in T_\Omega^\Pi$ □

The idea of the template and its instantiation is made precise by Π–abstraction and application of such Π–sorts to non–polymorphic sorts $s \in T_\Omega(\chi)$.

In a signature every constant or mapping will have a sort without free sort variables. This motivates the following definition.

Definition 2.5 Closed Sort Terms:

$$T_\Omega = T_\Omega(\emptyset)$$
$$T_\Omega^{closed} = T_\Omega \cup T_\Omega^\Pi$$

□

Note that $T_{\Omega,cpo}^{closed}$ will contain valid sorts for constants while $T_{\Omega,map}^{closed}$ will contain valid sorts for mappings.
Now we are able to define polymorphic signatures.

Definition 2.6 Polymorphic Signature:

A polymorphic signature $\Sigma = (\Omega, F, O)$ is a triple where:

- $\Omega = (K, \leq, SC)$ is a sort–signature.
- $F = \{F_\mu\}_{\mu \in T_{\Omega,cpo}^{closed}}$ is an indexed set of constant symbols.
- $O = \{O_\nu\}_{\nu \in T_{\Omega,map}^{closed}}$ is an indexed set of mapping symbols.

It must include the standard signature
$\Sigma_{standard} = (\Omega_{standard}, F_{standard}, O_{standard})$ which is defined as follows:

- Predefined Constants ($F_{standard}$):
  - $\{\mathsf{true}, \mathsf{false}\} \subseteq F_{\mathsf{Bool}}$, $\{\neg\} \subseteq F_{\mathsf{Bool} \to \mathsf{Bool}}$, $\{\wedge, \vee, \Rightarrow\} \subseteq F_{\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}}$ are the boolean constants and connectives.
  - $\{\bot\} \subseteq F_{\Pi\alpha : cpo.\ \alpha}$ is the polymorphic bottom symbol.
  - $\{\mathsf{fix}\} \subseteq F_{\Pi\alpha : cpo.\ (\alpha \to \alpha) \to \alpha}$ is the polymorphic fixed point operator.
- Predefined mappings ($O_{standard}$):
  - $\{=, \sqsubseteq\} \subseteq O_{\Pi\alpha : cpo.\ \alpha \times \alpha\ \mathbf{to}\ \mathsf{Bool}}$ are the polymorphic equality and approximation predicates.
  - $\{\delta\} \subseteq O_{\Pi\alpha : cpo.\ \alpha\ \mathbf{to}\ \mathsf{Bool}}$ is the polymorphic definedness predicate.

□

B.2 The Language of Terms

Next we introduce terms over a polymorphic signature. The language we define serves as a core language for specifications in the small written in the concrete syntax of SPECTRUM. All constructs, except the **generated by** phrases which are treated semantically, may be translated into this language.

B.2.1 Context Free Language (Pre-Terms)

<term> ::=	ψ	(*Variables*)
	<id>	(*Constants*)
	<Πid> $\underline{[}${<sortexp> $//\underline{,}\}^{+}\underline{]}$	(*Polyconstant-Inst*)
	<map> <term>	(*Mapping application*)
	<Πmap> $\underline{[}${<sortexp> $//\underline{,}\}^{+}\underline{]}$<term>	(*Polymapping-Inst*)
	$\underline{\langle}\{$<term> $//\underline{,}\}^{2+}\underline{\rangle}$	(*Tuple* $n \geq 2$)
	$\underline{\lambda}$ <pattern> $\underline{.}$ <term>	(λ-*abstraction*)
	<term> <term>	(*Application*)
	$\underline{\mathbf{Q}}$ <tid> $\underline{.}$ <term>	($Q \in \{\forall^{\perp}, \exists^{\perp}\}$)
	$\underline{(}$ <term> $\underline{)}$	(*Priority*)

<tid> ::= ψ $\underline{:}$ <sortexp>

<sortexp> ::= $T_{\Omega}\ (\chi)$

<pattern> ::= <tid> | $\underline{\langle}\{$<tid> $//\underline{,}\}^{2+}\underline{\rangle}$

<id> ::= $F_{T_{\Omega},\, cpo}$

<map> ::= $O_{T_{\Omega},\, map}$

<Πid> ::= $F_{T^{\Pi}_{\Omega},\, cpo}$

<Πmap> ::= $O_{T^{\Pi}_{\Omega},\, map}$

Note that object variables $x \in \psi$ are all different from sort variables $\alpha \in \chi$ and that all variables are different from identifiers in F and O.

B.2.2 Context Sensitive Language

With the pre–terms at hand we can now define the well-formed terms. We use a technique similar to [Mit90] and give a calculus of formation rules. Since for sort variables there is only a binding mechanism in the language of sort terms but not in the language of object terms, we need no dynamic context for sort variables. The disjoint family χ of sort variables (the sort context) carries enough information. For the object variables however there are several binders and therefore we need an explicit variable context.

Definition 2.7 Sort Assertions

The set of sort assertions $\rhd$ is a set of tuples (χ, Γ, e, τ) where:

- χ is a sort context.
- $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is a set of sort assumptions (a variable context), such that $\tau_i \in T_{\Omega,cpo}(\chi)$ and no x_i occurs twice in the sort assumptions contained in Γ (valid context condition). This prohibits overloading of variables in one scope.
- e is the pre–term to be sorted.
- $\tau \in T_{\Omega,cpo}(\chi)$ is the derived sort for e.

$(\chi, \Gamma, e, \tau) \in \rhd$ if and only if there is a finite proof tree D for this fact according to the natural deduction system below. □

When we write $\Gamma \rhd_\chi e :: \tau$ in the text we actually mean that there is a proof tree (sort derivation) for $(\chi, \Gamma, e, \tau) \in \rhd$. If we want to refer to a special derivation D we write $D : \Gamma \rhd_\chi e :: \tau$. The intuitive meaning of the sort assertion (χ, Γ, e, τ) with $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is that if the variables $x_1, \ldots, x_n$ have sorts $\tau_1, \ldots, \tau_n$ then the pre–term e is well-formed and has sort τ.

Axioms:

$$\text{(var)}\frac{}{x : \tau \rhd_\chi x :: \tau} \qquad \text{(const)}\ \frac{}{\emptyset \rhd_\chi c :: \tau}\Big\{\ c \in F_\tau$$

$$\text{(}\Pi\text{-inst)}\frac{}{\emptyset \rhd_\chi f[s(\alpha_1), \ldots, s(\alpha_n)] :: s^\star(\tau)}\left\{\begin{array}{l} f \in F_{\Pi\alpha_1:k_1,\ldots,\alpha_n:k_n.\tau} \\ s : \xi \rightarrow T_\Omega(\chi) \quad \text{where} \\ s = \{s_k : \xi_k \rightarrow T_{\Omega,\,k}(\chi)\}_{k \in K} \\ \text{with} \quad \alpha_i \in \xi_{k_i} \end{array}\right.$$

Note that it is essential to use an order kinded family of functions $s = \{s_k : \xi_k \rightarrow T_{\Omega,\,k}(\chi)\}_{k \in K}$ for the instantiation because the bound sort variables α_i of kind k_i have to be instantiated with a sort term of appropriate kind. Due to order sorted notation $s : \xi \rightarrow T_\Omega(\chi)$ on the level of sort terms this kind-correct instantiation comes for free. Note that the above definition also guarantees shallow polymorphism [Sok89].

Inference Rules:

$$\text{(weak)}\frac{\Gamma \rhd_\chi e :: \tau}{\Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \rhd_\chi e :: \tau} \qquad \text{(Note valid context condition!)}$$

$$\text{(map-appl)}\ \frac{\Gamma \rhd_\chi e :: \tau_1}{\Gamma \rhd_\chi oe :: \tau_2}\left\{ o \in \mathrm{O}_{\tau_1 \,\mathbf{to}\, \tau_2} \right.$$

$$\text{(}\Pi\text{map-appl)}\frac{\Gamma \rhd_\chi e :: s^\star(\tau_1)}{\Gamma \rhd_\chi o[s(\alpha_1), \ldots, s(\alpha_n)]e :: s^\star(\tau_2)}\left\{ \begin{array}{l} o \in \mathrm{O}_{\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n \,.\, \tau_1 \,\mathbf{to}\, \tau_2} \\ s : \xi \longrightarrow T_\Omega(\chi) \quad \text{where} \\ s = \{s_k : \xi_k \longrightarrow T_{\Omega,\, k}(\chi)\}_{k \in K} \\ \text{with} \quad \alpha_i \in \xi_{k_i} \end{array} \right.$$

$$\text{(tuple)}\ \frac{\Gamma \rhd_\chi e_1 :: \tau_1 \ldots \Gamma \rhd_\chi e_n :: \tau_n}{\Gamma \rhd_\chi \langle e_1, \ldots, e_n \rangle :: \tau_1 \times \ldots \times \tau_n}\left\{ n \geq 2 \right.$$

$$\text{(abstr)}\frac{\Gamma, x : \tau_1 \rhd_\chi e :: \tau_2}{\Gamma \rhd_\chi \lambda x : \tau_1 . e :: \tau_1 \longrightarrow \tau_2}\left\{ \begin{array}{l} x \text{ is not free on a} \\ \text{mapping's argument} \\ \text{position. No abstraction} \\ \text{over mappings!} \end{array} \right.$$

$$\text{(patt-abstr)}\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \rhd_\chi e :: \tau}{\Gamma \rhd_\chi \lambda\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle . e :: \tau_1 \times \ldots \times \tau_n \longrightarrow \tau}\left\{ \begin{array}{l} \text{No } x_i \text{ is free on} \\ \text{a mappings's} \\ \text{argument} \\ \text{position.} \end{array} \right.$$

$$\text{(appl)}\frac{\Gamma \rhd_\chi e_1 :: \tau_1 \longrightarrow \tau_2 \qquad \Gamma \rhd_\chi e_2 :: \tau_1}{\Gamma \rhd_\chi e_1 e_2 :: \tau_2}$$

$$\text{(quantifier)}\ \frac{\Gamma, x : \tau \rhd_\chi e :: \mathsf{Bool}}{\Gamma \rhd_\chi Qx : \tau . e :: \mathsf{Bool}}\left\{ Q \in \{\forall^\perp, \exists^\perp\} \right.$$

$$\text{(priority)}\frac{\Gamma \rhd_\chi e :: \tau}{\Gamma \rhd_\chi (e) :: \tau}$$

Example 2.3 Restricted λ-abstraction

The side conditions of rules **(abstr)** and **(patt-abstr)** prohibits the building of terms like:

$$\lambda x : \mathsf{Bool} . = [\mathsf{Bool}]\langle x, x \rangle$$

The reason for this restriction is that we want λ-terms always to denote continuous functions. On the other hand the interpretation of a mapping symbol may be a non-continuous mapping in our semantics. In the example above, the mapping's interpretation is the polymorphic identity which is by definition not monotonic. If we allow the above expression as component in a well-formed term its interpretation would have to be a non-monotonic function. □

B.2.3 Well-formed Terms and Sentences

With the context-sensitive syntax of the previous paragraph we are now able to define the notion of well-formed terms over a polymorphic signature. Since we use an explicitly sorted system, a well–formed term is a pre–term e together with a sort context χ, a variable context Γ and a sort τ. Erasing all sort information from the pre–terms yields terms in an implicit sort system without explicit typing for bound variables and no explicit instantiation of polymorphic objects. This leads to the well known problem of relating an explicitly sorted system with its implicit version (ref. [Gun92]). We do not address this problem here and go on with our explicit sort system.

Definition 2.8 Well-formed terms

Let Σ be a polymorphic signature. The set of well-formed terms over Σ in sort context χ and variable context Γ with sort τ is defined as follows:

$$T_{\Sigma,\tau}(\chi,\Gamma) = \{(\chi,\Gamma,e,\tau) \mid \Gamma \rhd_\chi e :: \tau\}$$

The set of all well-formed terms in context (χ,Γ) is defined to be the family

$$T_\Sigma(\chi,\Gamma) = \{T_{\Sigma,\tau}(\chi,\Gamma)\}_{\tau \in T_{\Omega(\chi)}}$$

In addition we define the following abbreviations:

$$T_\Sigma(\chi) = T_\Sigma(\chi,\emptyset) \quad \text{(closed object terms)}$$

$$T_\Sigma = T_\Sigma(\emptyset) \quad \text{(non-polymorphic closed object terms)}$$

□

Next we define formulae $\mathbf{Form}(\Sigma,\chi,\Gamma)$ and sentences $\mathbf{Sen}(\Sigma,\chi)$ over a polymorphic signature Σ and sort context χ. In SPECTRUM the set of formulae $\mathbf{Form}(\Sigma,\chi,\Gamma)$ is the set of well-formed terms in context (χ,Γ) of sort Bool. This leads to a three-valued logic. Sentences are as usual closed formulae.

Definition 2.9 Formulae and Sentences

$$\mathbf{Form}(\Sigma,\chi,\Gamma) = T_{\Sigma,\mathsf{Bool}}(\chi,\Gamma)$$

$$\mathbf{Sen}(\Sigma,\chi) = \mathbf{Form}(\Sigma,\chi,\emptyset) \quad \text{(closed formulae are sentences)}$$

$$\mathbf{Sen}(\Sigma) = \mathbf{Sen}(\Sigma,\emptyset) \quad \text{(non-polymorphic sentences)}$$

□

Example 2.4

$$\forall^{\perp} x : \alpha . = [\alpha]\langle x, x\rangle \in \mathbf{Sen}(\Sigma, \{\alpha\})$$
$$\forall^{\perp} x : \mathbf{Nat} . = [\mathbf{Nat}]\langle x, x\rangle \in \mathbf{Sen}(\Sigma)$$

□

Definition 2.10 Specification

A polymorphic specification $S = (\Sigma, E)$ is a pair where $\Sigma = (\Omega, F, O)$ is a polymorphic signature and $E \subset \mathbf{Sen}(\Sigma, \chi)$ is a set of sentences for some sort context χ □

B.3 Algebras

The following definitions are standard definitions of domain theory (see [Gun92]). We include them here to get a self contained presentation.

Definition 2.11 Partial Order

A partial order $\mathcal{A}$ is a pair $(A, \leq)$ where A is a set and $(\leq) \subseteq A \times A$ is a reflexive, transitive and antisymmetric relation. □

Definition 2.12 Chain Complete Partial Order

A partial order $\mathcal{A}$ is (countably) chain complete **iff** every chain $a_1 \leq \ldots \leq a_n \leq \ldots,\ n \in \mathbb{N}$ has a least upper bound in $\mathcal{A}$. We denote it by $\sqcup_{i \in \mathbb{N}} a_i$. □

Definition 2.13 Pointed Chain Complete Partial Order (PCPO)

A chain complete partial order $\mathcal{A}$ is pointed **iff** it has a least element. In the sequel we denote this least element by uu_A. □

Definition 2.14 Monotonic Functions

Let $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ be two PCPOs. A function[3] $f \in B^A$ is *monotonic* **iff**

$$d \leq_A d' \ \Rightarrow\ f(d) \leq_B f(d')$$

□

[3] We write B^A for all functions from A to B.

Definition 2.15 Continuous Functions:

A function between PCPOs $\mathcal{A}$ and $\mathcal{B}$ is continuous (and therefore monotonic) **iff** for every chain $a_1 \leq \ldots \leq a_n \leq \ldots$ in $\mathcal{A}$:

$$f(\bigsqcup_{i \in \mathbb{N}} a_i) = \bigsqcup_{i \in \mathbb{N}} f(a_i)$$

□

Definition 2.16 Product PCPO

If $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ are two PCPOs then the product PCPO $\mathcal{A} \times \mathcal{B} = (A \times B, \leq_{A \times B})$ is defined as follows:

- $A \times B$ is the usual cartesian product of sets,
- $(d, e) \leq_{A \times B} (d', e')$ **iff** $(d \leq_A d) \wedge (e \leq_B e')$,
- $uu_{A \times B} = (uu_A, uu_B)$

This definitions may be generalized to n-ary products in a straight forward way.

□

Definition 2.17 Function PCPO

If $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ are two PCPOs then the function PCPO $\mathcal{A} \xrightarrow{c} \mathcal{B} = (A \xrightarrow{c} B, \leq_{A \xrightarrow{c} B})$ is defined as follows:

- $A \xrightarrow{c} B$ is the set of all continuous functions from A to B,
- $f \leq_{A \xrightarrow{c} B} g$ **iff** $\forall a \in A. f(a) \leq_B g(a)$,
- $uu_{A \xrightarrow{c} B} = \lambda x : A. uu_B$

□

Definition 2.18 Lift PCPO

If $\mathcal{A} = (A, \leq_A)$ is a PCPO then the lifted PCPO $\mathcal{A}\,\mathbf{lift} = (A\,\mathbf{lift}, \leq_{A\,\mathbf{lift}})$ is defined as follows:

- $A\,\mathbf{lift} = (A \times \{0\}) \cup \{uu_{A\,\mathbf{lift}}\}$ where $uu_{A\,\mathbf{lift}}$ is a new element which is not a pair.
- $(x, 0) \leq_{A\,\mathbf{lift}} (y, 0)$ **iff** $x \leq_A y$

  $\forall z \in A\,\mathbf{lift}. uu_{A\,\mathbf{lift}} \leq_{A\,\mathbf{lift}} z$

- We also define an extraction function $\downarrow$ from A **lift** to A such that

$$\downarrow uu_{A\,\mathbf{lift}} = uu_A \qquad ; \qquad \downarrow (x, 0) = x$$

□

We will call the PCPOs also domains (note that in the literature domains are usually algebraic directed complete po's [Gun92]).

B.3.1 The Sort Algebras

Definition 2.19 Sort–Algebras

Let $\Omega = (K, \leq, SC)$ be a sort–signature. An Ω–algebra $\mathcal{SA} = (\mathcal{K}, \subseteq, \mathcal{DC})$ is an order sorted algebra[4] of domains i.e.:

- For each kind $k \in K$ with $k \leq cpo$ we have a set of domains $k^{\mathcal{SA}} \subseteq \mathcal{K}$. For the kind $map \in K$ we have a set of full functions spaces $map^{\mathcal{SA}}$.
- For all kinds $k_1, k_2 \in K$ with $k_1 \leq k_2$ we have $k_1^{\mathcal{SA}} \subseteq k_2^{\mathcal{SA}}$.
- $\mathcal{DC} = \{\mathcal{DC}_{k_1 \ldots k_n, k}\}_{k, k_i \in K}$ is an indexed set of domain constructors with:

$$\mathcal{DC}_{k_1 \ldots k_n, k} = \{sc^{\mathcal{SA}} : k_1^{\mathcal{SA}} \times \ldots \times k_n^{\mathcal{SA}} \to k^{\mathcal{SA}} \mid sc \in SC_{k_1 \ldots k_n, k}\}$$

  such that if $sc \in SC_{w,s} \cap SC_{w',s'}$ and $w \leq w'$ then

$$sc^{\mathcal{SA}}_{w',s'} \mid_{w^{\mathcal{SA}}} = sc^{\mathcal{SA}}_{w,s}$$

  In other words overloaded domain constructors must be equal on the smaller domain $w^{\mathcal{SA}} = k_1^{\mathcal{SA}} \times \ldots \times k_n^{\mathcal{SA}}$ where $w = k_1 \ldots k_n$.

We further require the following interpretation for the sort constructors occurring in the standard sort–signature:

- $\mathsf{Bool}^{\mathcal{SA}} = (\{uu_{\mathsf{Bool}}, \mathit{ff}, \mathit{tt}\}, \leq_{\mathsf{Bool}})$ is the flat three–valued boolean domain.
- For $\times_n^{\mathcal{SA}} \in cpo^{\mathcal{SA}} \times \ldots \times cpo^{\mathcal{SA}} \to cpo^{\mathcal{SA}}$:

$$\times_n^{\mathcal{SA}}(d_1, \ldots, d_n) = d_1 \times \ldots \times d_n, \; n \geq 2$$

  is the n-ary cartesian product of domains.
- For $\rightarrow^{\mathcal{SA}} \in cpo^{\mathcal{SA}} \times cpo^{\mathcal{SA}} \to cpo^{\mathcal{SA}}$:

$$\rightarrow^{\mathcal{SA}}(d_1, d_2) = (d_1 \overset{c}{\to} d_2)\mathbf{lift}$$

  is the lifted domain of continuous functions. We lift this domain because we want to distinguish between $\perp$ and $\lambda x.\, \perp$.

[4] See [GM87, Gog76].

- For $\mathbf{to}^{\mathcal{SA}} \in cpo^{\mathcal{SA}} \times cpo^{\mathcal{SA}} \to map^{\mathcal{SA}}$

$$\mathbf{to}^{\mathcal{SA}}(d_1, d_2) = d_2^{d_1}$$

  is the full function space between d_1 and d_2.

□

Definition 2.20 Interpretation of sort terms

Let $\nu : \chi \to \mathcal{SA}$ be a sort environment and $\nu^* : T_\Omega(\chi) \to \mathcal{SA}$ its homomorphic extension. Then $\mathcal{SA}_{[\![.]\!]}\nu$ is defined as follows:

- $\mathcal{SA}[\![e]\!]\nu = \nu^*(e)$ **if** $e \in T_\Omega(\chi)$
- $\mathcal{SA}\left[\!\left[\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n.e\right]\!\right] =$
  $\{f \mid f[\nu(\alpha_1), \ldots, \nu(\alpha_n)] \in \mathcal{SA}[\![e]\!]\nu$ **for all** $\nu\}$

For closed terms we write for $\mathcal{SA}[\![e]\!]$ also $e^{\mathcal{SA}}$. □

Sort terms in T_Ω^Π are interpreted as generalized cartesian products (dependent products). By using *n-ary* dependent products we can interpret Π–terms in one step. This leads to simpler models as the ones for the polymorphic λ–calculus.

Polymorphic Algebras

Definition 2.21 Polymorphic Algebra

Let $\Sigma = (\Omega, F, O)$ be a polymorphic signature with $\Omega = (K, \leq, SC)$ the sort–signature. A polymorphic Σ–algebra $\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ is a triple where:

- $\mathcal{SA} = (\mathcal{K}, \subseteq, \mathcal{DC})$ is an Ω sort algebra,
- $\mathcal{F} = \{\mathcal{F}_\mu\}_{\mu \in T^{closed}_{\Omega,cpo}}$ is an indexed set of constants (or functions), with:

$$\mathcal{F}_\mu = \{f^{\mathcal{A}} \in \mu^{\mathcal{SA}} \mid f \in F_\mu\}$$

- $\mathcal{O} = \{\mathcal{O}_\nu\}_{\nu \in T^{closed}_{\Omega,map}}$ is an indexed set of mappings, with:

$$\mathcal{O}_\nu = \{o^{\mathcal{A}} \in \nu^{\mathcal{SA}} \mid o \in O_\nu\}$$

We further require a fixed interpretation for the symbols in the standard signature. In order to simplify notation we will write $f^{\mathcal{A}}_{d_1,\ldots,d_n}$ for the instance $f^{\mathcal{A}}[d_1, \ldots, d_n]$ of a polymorphic function and $o^{\mathcal{A}}_{d_1,\ldots,d_n}$ for the instance $o^{\mathcal{A}}[d_1, \ldots, d_n]$ of a polymorphic mapping.

- Predefined Mappings ($O_{standard}$):
  - $\{=, \sqsubseteq\} \subseteq O_{\Pi\alpha:cpo.\,\alpha\times\alpha \text{ to Bool}}$ are interpreted as *identity* and *partial order*. More formally, for every domain $d \in cpo^{\mathcal{A}}$ and $x, y \in d$:

$$x =_d^{\mathcal{A}} y := \begin{cases} tt & \text{if } x \text{ is identical to } y \\ ff & \text{otherwise} \end{cases}$$

$$x \sqsubseteq_d^{\mathcal{A}} y := \begin{cases} tt & \text{if } x \leq_d y \\ ff & \text{otherwise} \end{cases}$$

  - $\{\delta\} \subseteq O_{\Pi\alpha:cpo.\,\alpha \text{ to Bool}}$ is the polymorphic definedness predicate. For every $d \in cpo^{\mathcal{A}}$ and $x \in d$:

$$\delta_d^{\mathcal{A}}(x) := \begin{cases} tt & \text{if } x \text{ is different from } uu_d \\ ff & \text{otherwise} \end{cases}$$

- Predefined Constants ($F_{standard}$):
  - $\{\mathsf{true}, \mathsf{false}\} \subseteq F_{\mathsf{Bool}}$ are interpreted in the $\mathsf{Bool}^{\mathcal{SA}}$ domain as follows:

$$\mathsf{true}^{\mathcal{A}} = tt \qquad ; \qquad \mathsf{false}^{\mathcal{A}} = ff$$

  - The interpretations of $\{\neg\} \subseteq F_{\mathsf{Bool}\to\mathsf{Bool}}$, $\{\wedge, \vee, \Rightarrow\} \subseteq F_{\mathsf{Bool}\times\mathsf{Bool}\to\mathsf{Bool}}$ are pairs in the lifted function spaces such that the function components behave like three–valued Kleene connectives on $\mathsf{Bool}^{\mathcal{SA}}$ as follows:

x	y	$(\downarrow \neg^{\mathcal{A}})(x)$	$x(\downarrow \wedge^{\mathcal{A}})y$	$x(\downarrow \vee^{\mathcal{A}})y$	$x(\downarrow \Rightarrow^{\mathcal{A}})y$
tt	tt	ff	tt	tt	tt
tt	ff	ff	ff	tt	ff
ff	tt	tt	ff	tt	tt
ff	ff	tt	ff	ff	tt
uu	tt	uu	uu	tt	tt
uu	ff	uu	ff	uu	uu
uu	uu	uu	uu	uu	uu
tt	uu	ff	uu	tt	uu
ff	uu	tt	ff	uu	tt

  - $\{\bot\} \subseteq F_{\Pi\alpha:cpo.\,\alpha}$ is interpreted in each domain as the least element of this domain. For every $d \in cpo^{\mathcal{SA}}$:

$$\bot_d^{\mathcal{A}} := uu_d$$

  - $\{\mathsf{fix}\} \subseteq F_{\Pi\alpha:cpo.\,(\alpha\to\alpha)\to\alpha}$ is interpreted for each domain d as a pair $\mathsf{fix}_d^{\mathcal{A}} \in (d \to^{\mathcal{SA}} d) \to^{\mathcal{SA}} d$ such that the function component behaves as follows:

$$(\downarrow \mathsf{fix}_d^{\mathcal{A}})(f) := \bigsqcup_{i \in \mathbb{N}} f^n(uu_d)$$

where:

$$f^0(uu_d) := uu_d$$

$$f^{n+1}(uu_d) := (\downarrow f)(f^n(uu_d))$$

Note that $\downarrow uu_{(d \overset{c}{\to} d)\,\mathbf{lift}} = uu_{(d \overset{c}{\to} d)}$ and therefore the above definition is sound.

□

B.4 Models

B.4.1 Interpretation of Sort Assertions

In this section we define the interpretation of well-formed terms. The interpretation of $(\chi, \Gamma, e, \tau) \in T_{\Sigma,\tau}(\chi, \Gamma)$ is defined inductively on the structure of **a** sort derivation $D : \Gamma \rhd_\chi e :: \tau$. The technique used is again due to [Mit90].

Definition 2.22 Satisfaction of a variable context

Let $\Sigma = (\Omega, F, O)$ be a polymorphic signature with $\Omega = (K, \leq, SC)$ and let $\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ be a a polymorphic Σ–algebra with $\mathcal{SA} = (\mathcal{K}, \subseteq, \mathcal{DC})$.

If Γ is a variable context and

$$\begin{array}{ll} \nu = \{\nu_k : \chi_k \to k^{\mathcal{SA}}\}_{k \in K \setminus \{map\}} & \text{sort environment (order-sorted)} \\ \eta : \psi \to \bigcup\limits_{d \in cpo^{\mathcal{SA}}} d & \text{object environment (unsorted)} \end{array}$$

then η satisfies Γ in sort environment ν (in symbols $\eta \models_\nu \Gamma$) **iff**

$$\eta \models_\nu \Gamma \Leftrightarrow \text{for all } x : \tau \in \Gamma . \eta(x) \in \nu^\star(\tau)$$

□

Definition 2.23 Update of object environments

$$\eta[a/x](y) := \begin{cases} a & \text{if } x = y \\ \eta(y) & \text{otherwise} \end{cases}$$

□

Now we define an order-sorted total meaning function $\mathcal{A}[\![\cdot]\!]_{\nu,\eta}$ that maps sort derivations $D : \Gamma \rhd_\chi e :: \tau$ to elements in $\mathcal{A}$. It may be proved that the meaning of $(\chi, \Gamma, e, \tau) \in \rhd$ is independent of the derivation $D : \Gamma \rhd_\chi e :: \tau$ we choose. This leads to a total meaning function $\mathcal{A}[\![\cdot]\!]_{\nu,\eta} : T_\Sigma(\chi, \Gamma) \to \mathcal{A}$.

Definition 2.24 Meaning of a sort derivation

The meaning of a sort derivation $D : \Gamma \rhd_\chi e :: \tau$ in a polymorphic algebra $\mathcal{A}$ in sort context ν and variable context Γ such that $\eta \models_\nu \Gamma$ is $\mathcal{A}[\![D : \Gamma \rhd_\chi e :: \tau]\!]_{\nu,\eta}$ which is recursively defined on the structure of D. The defining clauses are given below. □

Base cases:

(var) $\mathcal{A}[\![x : \tau \rhd_\chi x :: \tau]\!]_{\nu,\eta} = \eta(x)$ (const) $\mathcal{A}[\![\emptyset \rhd_\chi c :: \tau]\!]_{\nu,\eta} = c^{\mathcal{A}}$

(Π-inst)

$$\mathcal{A}[\![\emptyset \rhd_\chi f[s(\alpha_1), \ldots, s(\alpha_n)] :: s^\star(\tau)]\!]_{\nu,\eta} = f^{\mathcal{A}}[\nu^\star(s(\alpha_1)), \ldots, \nu^\star(s(\alpha_n))]$$

Inductive cases:
(weak)

$$\mathcal{A}[\![\Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \rhd_\chi e :: \tau]\!]_{\nu,\eta} = \mathcal{A}[\![\Gamma \rhd_\chi e :: \tau]\!]_{\nu,\eta}$$

(map-appl)

$$\mathcal{A}[\![\Gamma \rhd_\chi oe :: \tau_2]\!]_{\nu,\eta} = o^{\mathcal{A}}(\mathcal{A}[\![\Gamma \rhd_\chi e :: \tau_1]\!]_{\nu,\eta})$$

(Πmap-appl)

$$\mathcal{A}[\![\Gamma \rhd_\chi o[s(\alpha_1), \ldots, s(\alpha_n)]e :: s^\star(\tau_2)]\!]_{\nu,\eta} = o^{\mathcal{A}}[\nu^\star(s(\alpha_1)), \ldots, \nu^\star(s(\alpha_n))](\mathcal{A}[\![\Gamma \rhd_\chi e :: s^\star(\tau_1)]\!]_{\nu,\eta})$$

(tuple)

$$\mathcal{A}[\![\Gamma \rhd_\chi \langle e_1, \ldots, e_n\rangle :: \tau_1 \times \ldots \times \tau_n]\!]_{\nu,\eta} = (\mathcal{A}[\![\Gamma \rhd_\chi e_1 :: \tau_1]\!]_{\nu,\eta}, \ldots, \mathcal{A}[\![\Gamma \rhd_\chi e_n :: \tau_n]\!]_{\nu,\eta})$$

(abstr)

$$\mathcal{A}[\![\Gamma \rhd_\chi \lambda x : \tau_1.e :: \tau_1 \longrightarrow \tau_2]\!]_{\nu,\eta} = \text{the } \underline{\text{unique}} \text{ pair } (f, 0) \in \nu^\star(\tau_1 \longrightarrow \tau_2) \text{ with } \forall a \in \nu^\star(\tau_1).f(a) = \mathcal{A}[\![\Gamma, x : \tau_1 \rhd_\chi e :: \tau_2]\!]_{\nu,\eta[a/x]}$$

(patt-abstr)

$$\begin{aligned}&\mathcal{A}[\![\Gamma \rhd_\chi \lambda\langle x_1:\tau_1,\ldots,x_n:\tau_n\rangle.e :: \tau_1\times\ldots\times\tau_n\longrightarrow\tau]\!]_{\nu,\eta} =\\ &\quad \text{the } \underline{\text{unique}} \text{ pair } (f,0)\in\nu^\star(\tau_1\times\ldots\times\tau_n\longrightarrow\tau) \text{ with}\\ &\quad \forall a_1\in\nu^\star(\tau_1),\ldots,a_n\in\nu^\star(\tau_n).f((a_1,\ldots,a_n)) =\\ &\quad \mathcal{A}[\![\Gamma, x_1:\tau_1,\ldots,x_n:\tau_n \rhd_\chi e :: \tau]\!]_{\nu,\eta[a_1/x_1,\ldots,a_n/x_n]}\end{aligned}$$

(appl)[5]

$$\mathcal{A}[\![\Gamma \rhd_\chi e_1e_2 :: \tau_2]\!]_{\nu,\eta} =\downarrow (\mathcal{A}[\![\Gamma \rhd_\chi e_1 :: \tau_1\longrightarrow\tau_2]\!]_{\nu,\eta})(\mathcal{A}[\![\Gamma \rhd_\chi e_2 :: \tau_1]\!]_{\nu,\eta})$$

(universal quantifier)

$$\mathcal{A}[\![\Gamma \rhd_\chi \forall^\perp x:\tau.e :: \mathsf{Bool}]\!]_{\nu,\eta} =$$

$$= \begin{cases} tt & \text{if } \forall a\in\nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = tt)\\ ff & \text{if } \exists a\in\nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = ff)\\ uu & \text{otherwise}\end{cases}$$

(existential quantifier)

$$\mathcal{A}[\![\Gamma \rhd_\chi \exists^\perp x:\tau.e :: \mathsf{Bool}]\!]_{\nu,\eta} =$$

$$= \begin{cases} tt & \text{if } \exists a\in\nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = tt)\\ ff & \text{if } \forall a\in\nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = ff)\\ uu & \text{otherwise}\end{cases}$$

B.4.2 Satisfaction and Models

In this subsection we define the satisfaction relation for boolean terms and sentences (closed boolean terms) and also the notion of a model.

Definition 2.25 Satisfaction

Let

$\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$	Σ-Algebra
$\nu = \{\nu_k : \chi_k \to k^{\mathcal{SA}}\}_{k\in K\setminus\{map\}}$	sort environment (order-sorted)
$\eta : \psi \to \bigcup_{d\in cpo^{\mathcal{SA}}} d$	object environment (unsorted)

[5] Note that we use an implicit apply function for the application of elements from the lifted function space. Our algebras are a variation of what Mitchell calls *type frames* [Mit90].

and Γ a variable context with $\eta \models_\nu \Gamma$ then:

$\mathcal{A}$ satisfies $(\chi, \Gamma, e, \mathsf{Bool}) \in \mathbf{Form}(\Sigma, \chi, \Gamma)$ wrt sort environment ν and object environment η (in symbols $\mathcal{A} \models_{\nu,\eta} (\chi, \Gamma, e, \mathsf{Bool})$) **iff**

$$\mathcal{A} \models_{\nu,\eta} (\chi, \Gamma, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A}[\![\Gamma \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta} = tt$$

A special case of the above definition is the satisfaction of sentences. Let $(\chi, \emptyset, e, \mathsf{Bool}) \in \mathbf{Sen}(\Sigma, \chi)$ and η_0 an arbitrary environment, then:

$$\mathcal{A} \models_\nu (\chi, \emptyset, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A}[\![\emptyset \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta_0} = tt$$

$$\mathcal{A} \models (\chi, \emptyset, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A} \models_\nu (\chi, \emptyset, e, \mathsf{Bool}) \text{ for every } \nu$$

□

Now we are able to define models $\mathcal{A}$ of specifications $S = (\Sigma, E)$.

Definition 2.26 Models

Let $S = (\Sigma, E)$ be a specification. A polymorphic Σ-algebra $\mathcal{A}$ is a model of S (in symbols $\mathcal{A} \models S$) **iff**

$$\mathcal{A} \models S \Leftrightarrow \forall p \in E.\ \mathcal{A} \models p$$

□

Appendix C

Generation Principles and Induction

In this appendix we explain the semantics of the **generated by** phrases. We begin with a very technical definition of *reachable algebras* that is based on [Wir90]. We extend the notion of *algebra $\mathcal{A}$ is reachable by Cons* to polymorphic algebras with kinds.

C.1 Reachable Algebras

We assume here all definitions in Appendix B about polymorphic signatures and algebras and let $\Sigma = (\Omega, F, O)$ be a polymorphic signature with $\Omega = (K, \leq, SC)$ and let $\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ be a a polymorphic Σ–algebra with $\mathcal{SA} = (\mathcal{K}, \subseteq, \mathcal{DC})$. Further let $GS \subseteq SC$ be a sub–family of sort constructors and $CF \subseteq F$ a sub–family of function symbols. We call GS the sort constructors for the generated sorts and CF the constructor functions.
We now define the notion of primitive sorts and sorts of interest.

Definition 3.1 Primitive Sorts and Sorts of Interest

Given a sort context χ we define the set of primitive sorts P_χ to be

$$P_\chi = \{\tau \mid \tau \in T_{(K,\leq,SC\setminus GS)}(\chi)\}$$

The sorts of interest are defined as

$$SI_\chi = \{\tau \mid \tau = gs_n(\alpha_1, \ldots, \alpha_{l_n}); gs_n \in GS; n, l_n \in \mathbb{N}; 1 \leq i \leq l_n; \alpha_i \in \chi\}$$

Note that the definition of SI_χ also covers sort constructors with arity 0 (basic sorts). □

We now define the reachable sorts in an algebra with respect to families GS and CF. In this definition we allow elements in carriers that are not representable by terms but may be approximated by a sequence of terms.

Definition 3.2 *GS* **is reachable in** $\mathcal{A}$ **with** *CF*

GS is reachable in Σ–algebra $\mathcal{A}$ with CF **iff** for arbitrary sort context χ and sort environment ν, given a sort of interest $\tau \in SI_\chi$ and an element $a \in \mathcal{SA}[\![\tau]\!]\nu$ there exists

- a sequence of well–formed terms $(t_i)_{i\in\mathbb{N}}$ with

$$t_i = (\chi, \Theta, e_i, \tau) \in T_{(\Omega, CF\cup Fs, \emptyset), \tau}(\chi, \Theta)$$

  where

$$Fs = \{\bot\}_{\Pi\alpha\,:\,cpo.\alpha} \text{ and } \sigma \in P_\chi \text{ for all } x{:}\sigma \in \Theta$$

- an object environment η with $\eta \models_\nu \Theta$

such that

$$(\mathcal{A}[\![t_i]\!]_{\nu,\eta})_{i\in\mathbb{N}} \text{ is a chain, and } \bigsqcup_{i\in\mathbb{N}} \mathcal{A}[\![t_i]\!]_{\nu,\eta} = a$$

The sorts $\tau \in SI_\chi$ and the carriers $\mathcal{SA}[\![\tau]\!]\nu$ are called reachable (with CF). The terms t_i are called constructor terms. Note that $(\Omega, CF \cup Fs, \emptyset)$ is only a pseudo signature since it doesn't contain the entire standard signature $F_{standard}$. □

Definition 3.3 Finite and infinite elements

An element a in a reachable carrier set is called finite **iff** there is a single constructor term the interpretation of which is a. An element is called infinite (limit element) **iff** it is not finite. □

The above definition of reachable carriers τ ensures that the elements $a \in \mathcal{SA}[\![\tau]\!]\nu$ are reachable by chains whose elements are representable by terms. Note however that these terms don't need to have any structure in common. Often one wants to talk about an infinite element that is approximated by finite elements in a "canonical" way. Since the standard signature $F_{standard}$ contains the polymorphic fixed point operator $\mathsf{fix}{:}\Pi\alpha : cpo.\ (\alpha{\longrightarrow}\alpha){\longrightarrow}\alpha$, this is no problem. Using fix the only chains we get are those that result from the iteration of functions due to the semantics of fix. This restriction is not covered in the present definition of *reachable* (see [Möl82] for the notion of fixed point algebras).

C.2 Induction on Constructor Terms

The definitions in C.1 give rise to an induction principle on the structure of the constructor terms. Since we allow also limits of chains in reachable sorts the induction principle has to be formulated with some care. First we define the notion of a *predicate* which is not obvious since we use a three–valued logic.

Definition 3.4 Predicates

Suppose we have a signature Σ, a Σ–algebra $\mathcal{A}$, a formula $f[x:\sigma] \in \mathbf{Form}(\Sigma, \chi, \Theta)$ and a sorted variable $x:\sigma \in \Theta$.

The formula $f[x:\sigma]$ is called the characteristic function in x of the predicate

$$p_f[x:\sigma] = \{p_f[x:\sigma]_{\nu,\eta}\} \text{ were } p_f[x:\sigma]_{\nu,\eta} = \{a \mid \mathcal{A}[\![f]\!]_{\nu,\eta[a/x]} = tt\}$$

If there arises no confusion we will not distinguish between $f[x:\sigma]$ (syntax) and $p_f[x:\sigma]$ (semantics) and will call $f[x:\sigma]$ a predicate in x. □

Note that a condition

$$\forall \nu, \eta, a \in \mathcal{SA}[\![\sigma]\!]\nu.\ \mathcal{A}[\![f]\!]_{\nu,\eta[a/x]} \neq uu_{\mathbf{Bool}}$$

is not needed here. Definedness is a problem of logical implication. If we try to formulate an inductive step via logical implication, we have to ensure that our chain of inductive steps doesn't break down (compare the induction rules in Chapter 3.4).

The above definition says that predicates are subsets of carriers. The next definition is on closure properties of predicates with respect to limit elements.

Definition 3.5 Admissibility

A predicate $f[x:\sigma] \in \mathbf{Form}(\Sigma, \chi, \Theta)$ is called admissible in x **iff** $p_f[x:\sigma]_{\nu,\eta}$ is chain complete for every ν and η. □

Now we have all we need to define the principle of structural induction on reachable sorts.

Definition 3.6 Structural induction on reachable sorts

Suppose GS is reachable in $\mathcal{A}$ with CF. If a predicate $f[x:\sigma]$ is admissible in x, $\sigma \in SI_\chi$ and we want to establish

$$\forall \nu, \eta.\ p_f[x:\sigma]_{\nu,\eta} = \mathcal{SA}[\![\sigma]\!]\nu$$

it is enough to prove

$$\mathcal{A}[\![t]\!]_{\nu,\eta} \in p_f[x:\sigma]_{\nu,\eta}$$

for every constructor term

$$t = (\chi, \Theta, e, \sigma)$$

by induction on the structure of the constructor terms. Since reachability is defined with respect to a familiy GS, the above definition may easily be extended to simultaneous structural induction. □

C.3 Semantics of generated by

As was indicated in Appendix B, every specification in the concrete syntax of SPECTRUM may be translated in a specification formulated in the core language that was formalized in Appendix B. Since the process of translation is not presented in this paper we can only sketch the semantics of **generated by**.

Definition 3.7 Semantics of generated by (sketch)

Suppose we have a specification S in the concrete SPECTRUM syntax that contains the pseudo axiom

GS **generated by** CF

and let $S = (\Sigma, E)$ be the translated version of S in the core language, then a Σ–algebra $\mathcal{A}$ is a model of S **iff**

$$\mathcal{A} \models S \text{ and } GS \text{ is reachable in } \mathcal{A} \text{ by } CF$$

□

Turning it the other way round, GS **generated by** CF means that simultaneous induction on constructor terms with respect to GS and CF is required to be sound.

In [GR93] we present the explicitly sorted core language and its implicit compagnon in full detail. There we also define the logical calculus for SPECTRUM that is formulated in the implicit sort system. The induction rules, that originate from **generated by** phrases, along with various context conditions are described in this paper, too.

Appendix D

SPECTRUM's Predefined Specification

In SPECTRUM a number of symbols have a fixed semantics (see Appendix B). These symbols are defined in the following predefined signature which is part of every SPECTRUM specification.

Predefined_Signature = {

−−Top sort class CPO
class CPO;

−−Sort Bool
sort Bool;

−−Functions on Bool
true, false: Bool;
$\neg$: Bool $\longrightarrow$ Bool;
.$\Rightarrow$.: Bool $\times$ Bool $\longrightarrow$ Bool **prio 1:right**;
.$\vee$.: Bool $\times$ Bool $\longrightarrow$ Bool **prio 2:left**;
.$\wedge$.: Bool $\times$ Bool $\longrightarrow$ Bool **prio 3:left**;

−−polymorphic undefined element
$\perp$: α;

−−polymorphic fixed point function
fix: $(\alpha \longrightarrow \alpha) \longrightarrow \alpha$;

−−strong polymorphic equality
.=.: $\alpha \times \alpha$ **to** Bool **prio 5:left**;

```
    --polymorphic definedness predicate
    δ: α to Bool;

    --polymorphic less defined relation
    .⊑.: α × α to Bool                    prio 4:left;
}
```

In Predefined_Specification a number of additional functions are specified. This specification is also part of every SPECTRUM specification.

```
Predefined_Specification = {

    --Equivalence
    .⇔.: Bool × Bool ⟶ Bool               prio 1:right;
    .⇔. total strict;

    --strong inequality
    .≠.: α × α to Bool                    prio 5:left;
    .≠. strong;

    axioms x, y in
  {eqv}  (x ⇔ y) = ((x ⇒ y) ∧ (y ⇒ x));

  {neq}  (x ≠ y) = ¬(x = y);
    endaxioms;

    --definition of a weak equality
    class EQ;

    Bool :: EQ;

    .==. : α :: EQ ⇒ α × α ⟶ Bool;        prio 5:left;
    .==. strict total;
    axioms α :: EQ ⇒ ∀ a, b, c : α in

{weak_eq}  (a == b) = (a = b);

    endaxioms;

    --built-in polymorphic if_then_else_endif
    if_then_else_endif : Bool × α × α ⟶ α;
    if_then_else_endif total;
```

```
    axioms ∀⊥ e1, e2 in
    {if1}  if_then_else_endif (⊥, e1, e2) = ⊥;
    {if2}  if_then_else_endif (true, e1, e2) = e1;
    {if3}  if_then_else_endif (false, e1, e2) = e2;
    endaxioms;
}
```

Appendix E

Standard Library

This appendix provides a small library of frequently used specifications. The specifications given here are (unlike the Predefined_Specification of Appendix D) not part of every SPECTRUM specification. If needed, they have to be included via an appropriate **enriches** statement.

```
Standard_Lib = Character + List + String + Ordering +
               Numericals + Naturals;

Character = {
    data Char = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' |
                'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' |
                'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' |
                'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
                'Y' | 'Z' | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |
                '[' | ']' | '!' | '#' | '%' | '&' | '$' | '*' | '+' | '-' | '/' | '<' |
                '>' | '=' | '?' | '@' | '^' | '~' | '|' | '`' | '{' | '}' | '(' | ')' | '.' |
                ';' | ',' | ':' | '_' | ' ' | '\n' | '\t' | '\v' | '\b' | '\r' | '\f' | '\a' |
                '\\' | '\'' | '\"';

    Char :: EQ;
}

List = {

    data List α = [ ] | cons(!first:α, !rest:List α);

    List :: (EQ)EQ;

    .++. : List α × List α → List α        prio 10:left;
    .++. strict total;
```

```
    axioms ∀ s,s' : List α ∀ e : α in
        [] ++ s = s;
        cons(e,s') ++ s = cons(e,s'++s);
    endaxioms;
}

String = { enriches Character + List;
    --String is only an abbreviation for lists of characters
    sortsyn String = List Char;
}

Ordering = {

    class PO subclass of EQ;

    .≤. : α :: PO ⇒ α × α → Bool        prio 6;

    axioms α :: PO ⇒ ∀ x,y,z : α in
   {refl}  x ≤ x;
  {trans}  x ≤ y ∧ y ≤ z ⇒ x ≤ z;
    {ant}  x ≤ y ∧ y ≤ x ⇒ x == y;
    endaxioms;

    class TO subclass of PO;

    axioms α :: TO ⇒ ∀ x,y : α in
    {tot}  x ≤ y ∨ y ≤ x;
    endaxioms;
};

Numericals = { enriches Ordering;

    class NUM subclass of TO;

    --functions for sort class NUM
    .+.: α :: NUM ⇒ α × α → α        prio 6:left;
    .−.: α :: NUM ⇒ α × α → α        prio 6;
    .∗.: α :: NUM ⇒ α × α → α        prio 7:left;
    ./.: α :: NUM ⇒ α × α → α        prio 7;


    .+., .−., .∗., ./. strict;
    .+., .∗. total;
```

```
    axioms α :: NUM ⇒ ∀ a, b, c : α in
        --Associativity
        (a + b) + c = a + (b + c);
        (a * b) * c = a * (b * c);

        --Commutativity
        a + b = b + a;
        a * b = b * a;
    endaxioms;
}
Naturals = {enriches Numericals;

    data Nat = 0 | succ(!pred:Nat);
    Nat :: NUM;

    .mod.: Nat × Nat ⟶ Nat                    prio 7;
    .mod. strict;

    axioms ∀ n, m : Nat in

        --Addition
        n + 0 = n;
        n + succ m = succ (n + m);

        --Subtraction
        δ (n − m) ⇔ m ≤ n;
        (n + m) − m = n;

        --Multiplication
        n * 0 = 0;
        n * succ m = n + n * m;

        --Division
        δ(n/m) ⇔ m ≠ 0;
        m ≠ 0 ⇒ n mod m ≤ m ∧ n mod m ≠ m;
        m ≠ 0 ⇒ n = (n/m) * m + n mod m;

        --Ordering
        n ≤ succ n;
    endaxioms;
}
```