T U M

I N S T I T U T F Ü R I N F O R M A T I K

The Specification Language SPECTRUM Core Language Report V1.0

Radu Grosu, Dieter Nazareth

TUM-I9429
August 1994

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-8-1994-I9429-350/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

The Specification Language Spectrum
Core Language Report V1.0*

R. Grosu, D. Nazareth

Fakultät für Informatik, Technische Universität München
80290 München, Germany

E-Mail: {grosu,nazareth}@informatik.tu-muenchen.de

*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.

Contents

Chapter 1

Introduction

This report formally describes the core syntax of the specification language Spectrum. It does not explain the particular language constructs but only gives an exact definition of the syntax. An informal introduction to Spectrum can be found in [BFG+93a, BFG+93b]. The formal semantics is given in [GR94]. Thus, in the sequel we assume that the reader is familiar with the concepts used in Spectrum. As usual this description distinguishes between the *concrete syntax* and the *abstract syntax.*

1.1 The Concrete Syntax

The concrete syntax treats the language as a *set of strings* over an alphabet of symbols. Concrete syntax is usually specified by a *context free grammar* that gives *productions* for generating strings of symbols, using auxiliary *nonterminal* symbols.

It is common practice to distinguish a *lexical level* and a *phrase level* in concrete syntax. The terminal symbols in the grammar specifying the lexical level are single characters; those in the phrase–level grammar are the *nonterminal* symbols in the lexical grammar.

For example the following rules[1]

$$\begin{array}{lll}
\langle\text{sortexp}\rangle & ::= & \langle\text{sortexp1}\rangle{\rightarrow}\langle\text{sortexp}\rangle \\
\langle\text{sortexp1}\rangle & ::= & \langle\text{alphanumid}\rangle|(\langle\text{sortexp}\rangle)
\end{array}$$

with nonterminals ⟨sortexp⟩, ⟨sortexp1⟩ and ⟨alphanumid⟩ define possibly parenthesized arrow sorts. The sort identifiers ⟨alphanumid⟩ are defined by the lexical syntax:

$$\begin{array}{lll}
\langle\text{alphanumid}\rangle & ::= & \langle\text{letter}\rangle|\langle\text{digit}\rangle|\ \_ \\
\langle\text{letter}\rangle & ::= & \mathbf{a}|\mathbf{b}|\mathbf{c}|\mathbf{d}|\mathbf{e}|\mathbf{f}|\mathbf{g}|\mathbf{h}|\mathbf{i}|\mathbf{j}|\mathbf{k}|\mathbf{l}|\mathbf{m}|\mathbf{n}|\mathbf{o}|\mathbf{p}|\mathbf{q}|\mathbf{r}|\mathbf{s}|\mathbf{t}|\mathbf{u} \\
 & | & \mathbf{v}|\mathbf{w}|\mathbf{x}|\mathbf{y}|\mathbf{z}|\mathbf{A}|\mathbf{B}|\mathbf{C}|\mathbf{D}|\mathbf{E}|\mathbf{F}|\mathbf{G}|\mathbf{H}|\mathbf{I}|\mathbf{J}|\mathbf{K}|\mathbf{L}|\mathbf{M} \\
 & | & \mathbf{N}|\mathbf{O}|\mathbf{P}|\mathbf{Q}|\mathbf{R}|\mathbf{S}|\mathbf{T}|\mathbf{U}|\mathbf{V}|\mathbf{W}|\mathbf{X}|\mathbf{Y}|\mathbf{Z} \\
\langle\text{digit}\rangle & ::= & \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}|\mathbf{6}|\mathbf{7}|\mathbf{8}|\mathbf{9}
\end{array}$$

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

[1]These rules are simplified versions of the Spectrum rules.

$[rhs]$	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
$rhs_1 \| rhs_2$	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 meta-symbols [,], {, }, (,) and | introduced above from the corresponding terminal symbols **[**, **]**, **{**, **}**, **(**, **)** and **|** printed in boldface font.

To each nonterminal we associate a *phrase sort* containing all expressions which can be inferred from that nonterminal.

Notation: For each nonterminal ⟨nonterm⟩ its phrase sort is written as Nonterm and we use *nonterm* (possibly indexed) to range over Nonterm.

1.2 The Abstract Syntax

The *abstract syntax* treats the language as a *set of trees* each representing the structure of a parsed specification text. The important thing about trees is that unlike strings, their compositional structure is *inherently* unambiguous: there is only one way of constructing a particular tree out of its (immediate) sub–trees.

Given a concrete sort expression *sortexp* or a concrete expression *exp* we use *abs*(*sortexp*) and *abs*(*exp*) respectively, to denote their corresponding abstract trees. For the other abstract syntax objects we use a mathematical notation based on set theory which is summarized below:

$A \times B$	The Cartesian product of A and B. $A^k = \underbrace{A \times \ldots \times A}_{k}$.
$A \overset{fin}{\to} B$	The set of finite mappings from A to B. A finite map will be written explicitly in the form $\{a_1 \to b_1, \ldots, a_n \to b_n\}, n \geq 0$; in particular, the empty map is $\{\}$.
$Dom\, f, Ran\, f$	The domain and range of a finite map f.
$f + g$	For finite maps f and g, $f+g$ is called f *modified* by g. It is defined by: $Dom(f+g) = Dom(f) \cup Dom(g)$; $(f+g)(a) =$ if $a \in Dom(g)$ then $g(a)$ else $f(a)$. If $Dom\, f \cap Dom\, g = \emptyset$ then $f + g = f \cup g$.
$A \cup B$	The union of A and B.
$A \mid B$	The disjoint union of A and B.
$Fin(A)$	The set of finite subsets of A.
$List(A)$	The set of lists with elements taken from A. We write *a*:*l* for a list with first element *a* and rest *l*.

For example the following declaration:

$$ct \in ConType = \bigcup_{k \geq 0} (ClassId^k \times ClassId)$$

defines the set of sort constructor signatures as *ConType* and designate the variable *ct* (possibly indexed) to range over this set. The sets of identifiers *Id*, *ClassId*, etc. are taken directly from the concrete syntax. The advantage of the mathematical notation is the automatic provision of very abstract object manipulation primitives.

1.3 The Translation Rules

In general, the meaning of an expression occurring in a specification text is context dependent (e.g. a variable declared with type integer can be used in subsequent text only on an integer position). As a consequence, expressions generated by the context free grammar of the *concrete syntax* are called *rough expressions* because among them there are also erroneous ones. The set of *well formed expressions* i.e. the set of expressions respecting the context dependencies or static semantics (e.g. the typing discipline) are filtered out by the *context sensitive syntax.* We give this syntax by using a formalism based on logic. Specifically, we define concrete expressions and their translation into abstract (or semantic) objects simultaneously using axioms and inference rules. An inference rule has the form:

$$\frac{B \vdash cexp_1 \Rightarrow aexp_1 \ldots B \vdash cexp_n \Rightarrow aexp_n}{B \vdash cexp \Rightarrow aexp} \{\text{side conditions}$$

It allows to derive the conclusion $B \vdash cexp \Rightarrow aexp$ if all the assumptions $B \vdash cexp_i \Rightarrow aexp_i$ are true. The expressions $B \vdash cexp \Rightarrow aexp$ with $B = \{id_1 \rightarrow a_1, \ldots, id_n \rightarrow a_n\}$ are called *translation assertions.* They intuitively say that if identifiers $id_1, \ldots, id_n$ have attributes $a_1, \ldots a_n$ then *cexp* is well formed and its translation is *aexp.* The *environment* or *context* B in which *cexp* is translated is the abstract representation of the *symbol table.* Although for convenience, a context B is not always represented by a *finite mapping*, we assume that no id_i occurs twice in B and use B as a finite mapping.

Our contexts are sorted i.e. they are of the form $B_1 \times B_2 \times \ldots \times B_n$. They can also be structured i.e. if $id \rightarrow a$ then a can contain itself a context. Identifiers are also called *simple semantic objects.* All other objects from the abstract syntax (including contexts) are also called *compound semantic objects.*

In the translation rules, phrases within single square brackets [ ] are called *first options,* and those within double square brackets [[ ]] are called *second options.* To reduce the number of rules we adopt the following convention:

Notation: In each instance of a rule, the first options must be either all present or all absent; similarly the second options must be either all present or all absent.

When we write $[cse \Rightarrow ase]_{def}$ then we assume that def is the default semantic value when the option is missing.

Chapter 2

Lexical Syntax

```
⟨spectext⟩         ::=  {⟨lexeme⟩|⟨whitespace⟩|⟨comment⟩}^*
⟨lexeme⟩           ::=  ⟨charconst⟩|⟨num⟩|⟨string⟩|⟨alphanumid⟩
                    |   ⟨symbolid⟩|⟨inf-alphanumid⟩|⟨inf-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⟩|⟨eof⟩
⟨nest-comment⟩     ::=  (: {⟨no-nest⟩|⟨nest-comment⟩}^* :)
⟨no-nest⟩          ::=  {⟨any⟩}^*_{{any}^* {(:|:)} {⟨any⟩}^*}
⟨comment⟩          ::=  ⟨line-comment⟩|⟨nest-comment⟩
```

Character Constants

$\langle\text{char}\rangle \quad ::= \quad \langle\text{letter}\rangle|\langle\text{digit}\rangle|\langle\text{sym}\rangle|\langle\text{spcl}\rangle|\_|\langle\textit{space}\rangle|\langle\text{escapes}\rangle$
$\langle\text{escapes}\rangle \quad ::= \quad \mathbf{\backslash n}|\mathbf{\backslash t}|\mathbf{\backslash v}|\mathbf{\backslash r}|\mathbf{\backslash f}|\mathbf{\backslash a}|\mathbf{\backslash\backslash}|\mathbf{\backslash '}|\mathbf{\backslash "}$
$\langle\text{charconst}\rangle \quad ::= \quad \text{'}\ \langle\text{char}\rangle\ \text{'}$

String Constants
$\langle\text{string}\rangle \quad ::= \quad \text{"}\ \{\langle\text{char}\rangle\}^*\ \text{"}$

Natural Numbers
$\langle\text{num}\rangle \quad ::= \quad \langle\text{digit}\rangle_{\{\mathbf{0}\}}\ \{\langle\text{digit}\rangle\}^*$

Identifiers
$\langle\text{alphanumid}\rangle \quad ::= \quad \{\langle\text{alphanum}\rangle\}^+_{\{\langle\text{char}\rangle|\langle\text{num}\rangle|\langle\text{reserved}\rangle\}}$
$\langle\text{symbolid}\rangle \quad ::= \quad \{\langle\text{sym}\rangle\}^+_{\{\{\langle\text{sym}\rangle\}^*\ --\ \{\langle\text{sym}\rangle\}^*|\langle\text{reserved}\rangle\}}$
$\langle\text{inf-alphanumid}\rangle \quad ::= \quad \mathbf{.}\langle\text{alphanumid}\rangle\mathbf{.}$
$\langle\text{inf-symbolid}\rangle \quad ::= \quad \mathbf{.}\langle\text{symbolid}\rangle\mathbf{.}$

Reserved Words
$\langle\text{reserved}\rangle \quad ::= \quad$ **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**|**!**|**[**|**]**

Chapter 3

The Core Language

SPECTRUM consists of a language “in the small” and a language “in the large”. The first one is used for the design of a single specification unit, i.e. a basic component. The latter one is intended for structuring large system descriptions by combining and adapting specifications into more complex ones.

3.1 In the Small

3.1.1 Context Free Syntax

In this section we define the raw structure of our language “in the small” by giving a context-free grammar in EBNF style.

⟨specbody⟩	::=	**{** ⟨decls⟩ **}**	
⟨decls⟩	::=	{⟨signature⟩ ;|⟨axioms⟩ ;}*	
Signatures			
⟨signature⟩	::=	**class** ⟨id⟩ [**subclass of** {⟨id⟩ // ,}$^+$]	
	|	⟨id⟩ **subclass of** {⟨id⟩ // ,}$^+$	
	|	**sort** ⟨sid⟩ { ⟨sid⟩* | **::** ⟨classexp⟩ }	
	|	{⟨sid⟩ // ,}$^+$ **::** ⟨classexp⟩	
	|	**sortsyn** {⟨sid⟩}$^+$ = ⟨sortexp⟩	
	|	**SIG (** ⟨specexp⟩ **)**	
	|	{⟨id⟩ // ,}$^+$ **:** [⟨context⟩] ⟨sortexp⟩ **to** ⟨sortexp⟩	
	|	{⟨inf-id⟩ // ,}$^+$ **:** [⟨context⟩] ⟨sortexp⟩ × ⟨sortexp⟩ **to** ⟨sortexp⟩ [⟨prio⟩]	
	|	{⟨id⟩ // ,}$^+$ **:** [⟨context⟩] ⟨sortexp⟩	
	|	{⟨inf-id⟩ // ,}$^+$ **:** [⟨context⟩] ⟨sortexp⟩ × ⟨sortexp⟩ → ⟨sortexp⟩ [⟨prio⟩]	
⟨classexp⟩	::=	[({⟨id⟩ // ,}$^+$)] ⟨id⟩	
⟨prio⟩	::=	**prio** ⟨num⟩ [**:** {**left**|**right**}]	
Sort Expressions			
⟨sortexp⟩	::=	⟨sortexp1⟩	
	|	⟨sortexp1⟩ → ⟨sortexp⟩	*(Functional Sort)*

⟨sortexp1⟩	::=	⟨sortexp2⟩	
	|	⟨sortexp2⟩ {× ⟨sortexp2⟩}$^+$	*(Product Sort)*
⟨sortexp2⟩	::=	⟨asort⟩ | ⟨sid⟩ {⟨asort⟩}$^+$	
⟨asort⟩	::=	⟨sid⟩ | (⟨sortexp⟩)	

Sort Contexts

⟨context⟩	::=	{⟨scontext⟩ // ,}$^+$ ⇒
⟨scontext⟩	::=	{⟨sid⟩ // ,}$^+$ **::** ⟨id⟩

Axioms

⟨axioms⟩	::=	**axioms** [⟨varlist⟩] {[**{** ⟨id⟩ **}**] ⟨exp1⟩ **;**}* **endaxioms**
	|	⟨simplesorts⟩ **generated by** ⟨opns⟩
⟨varlist⟩	::=	[⟨context⟩] {{$\forall$|$\forall^\perp$} ⟨opdecls⟩}$^+$ **in**
⟨opdecls⟩	::=	{⟨sopdecl⟩ // ,}$^+$ | {⟨id⟩ // ,}$^+$
⟨sopdecl⟩	::=	{⟨id⟩ // ,}$^+$ **:** ⟨sortexp⟩
⟨simplesorts⟩	::=	{⟨sid⟩ {⟨sid⟩}* // ,}$^+$

Expressions

⟨exp1⟩	::=	⟨exp2⟩
	|	{ $\forall$ | $\forall^\perp$ | $\exists$ | $\exists^\perp$ } ⟨opdecl⟩ **.** ⟨exp1⟩
	|	**λ** ⟨pat⟩ **.** ⟨exp1⟩
⟨exp2⟩	::=	⟨exp3⟩ | ⟨exp2⟩ ⟨id⟩ [**:** ⟨asort⟩] ⟨exp1⟩
⟨exp3⟩	::=	⟨aexp⟩ | ⟨exp3⟩ ⟨aexp⟩
⟨aexp⟩	::=	⟨opn⟩ | **(** ⟨exp1⟩ **)** | **(** ⟨exp1⟩ {**,** ⟨exp1⟩}$^+$ **)** | ⟨aexp⟩ **:** ⟨asort⟩
⟨pat⟩	::=	⟨id⟩ [**:** ⟨sortexp⟩]
	|	**(** ⟨id⟩ [**:** ⟨sortexp⟩] {**,** ⟨id⟩ [**:** ⟨sortexp⟩]}$^+$ **)**

Identifiers

⟨id⟩	::=	⟨alphanumid⟩ | ⟨symbolid⟩ | ⟨num⟩ | ⟨charconst⟩ | ⟨string⟩
⟨inf-id⟩	::=	⟨inf-alphanumid⟩ | ⟨inf-symbolid⟩
⟨sid⟩	::=	$\langle\text{id}\rangle_{\{\to,\ \times,\ -\,>,\ *\ \}}$
⟨opn⟩	::=	⟨id⟩ | ⟨inf-id⟩
⟨opns⟩	::=	{⟨opn⟩ // ,}$^+$

3.1.2 Semantic Objects

A basic specification consists of a signature and a set of axioms. Thus, its abstraction is defined as follows:

(Σ, Ax) or *sp* $\in$ Specification $=$ Sig $\times$ Fin(Ax)

We start with the description of the abstract signature Sig.

Signatures

A signature contains all global identifiers known to a specification. We use separate environments for the different kinds of identifiers.

(*CE, SCE, SSE, ME, FE*) or Σ $\in$ Sig $=$ ClassEnv $\times$ SConEnv $\times$ SSynEnv $\times$ MapEnv $\times$ FuncEnv

The class environment contains all declared class identifiers and the defined subclass relation. We use a finite set of tuples to describe this relation.

$$(\mathit{IdSet},\ \mathit{Subset}) \text{ or } \mathit{CE} \quad \in \quad \text{ClassEnv} \quad = \quad \text{Fin(ClassId)} \times \text{Fin(ClassId} \times \text{ClassId)}$$

In the sort constructor environment all declared sort constructor identifiers are registered. In addition, for each identifier the possibly overloaded type information is stored. It consists of the parameter classes and the result class. The number of parameter classes depends on the arity of the sort constructor.

$$\begin{array}{rcrcl} (\mathit{SConSet},\ \mathit{SConType}) \text{ or } \mathit{SCE} & \in & \text{SConEnv} & = & \text{Fin(SConId)} \times \text{Fin(SconId} \times \text{ConType)} \\ & & \text{ConType} & = & \bigcup_{k\geq 0} (\text{ClassId}^k \times \text{ClassId}) \end{array}$$

The sort synonym environment contains all defined sort synonyms together with their assigned type expression abstractions.

$$\begin{array}{rcl} \text{SSynEnv} & = & \text{Id} \xrightarrow{fin} \text{TypeAbstr} \\ \text{TypeAbstr} & = & \bigcup_{k\geq 0} \text{TypeVar}^k \times \text{Type} \end{array}$$

Finally we need environments for the declared functions and mappings. They contain the sort information for the declared functions and mappings. In addition, for infix identifiers the priority (a natural number) and associativity information (left, right or none) is stored.

$$\begin{array}{rcrcl} \mathit{FE} & \in & \text{FuncEnv} & = & \text{Fin(TypeBind} \cup \text{TypeBind} \times \text{Prio} \times \text{Assoc)} \\ \mathit{ME} & \in & \text{MapEnv} & = & \text{Fin(TypeBind} \cup \text{TypeBind} \times \text{Prio} \times \text{Assoc)} \\ & & \text{TypeBind} & = & \text{Id} \times \text{TypeScheme} \\ p & \in & \text{Prio} & = & \text{subset of } \mathbb{N} \\ a & \in & \text{Assoc} & = & \text{left} \mid \text{right} \mid \text{none} \end{array}$$

The sort information for function and mapping identifiers consists of a binding for sort variables and an abstract sort expression. The sort variable binding assigns a class identifier to sort variables.

$$\begin{array}{rcrcl} \sigma & \in & \text{TypeScheme} & = & \text{SVarEnv} \times \text{Type} \\ \mathit{SVE} & \in & \text{SVarEnv} & = & \text{TypeVar} \xrightarrow{fin} \text{ClassId} \\ \tau & \in & \text{Type} & = & \text{abs}(\langle\text{sortexp}\rangle) \end{array}$$

Well Formed Signatures

Not all elements of the set Sig are legal signatures. Signatures must be *well formed*, i.e. they have to fulfil the additional conditions given below. To define these conditions we use the following notational shortcuts:

$$\begin{array}{rcl} \mathit{MapSet} & = & \{\ \mathit{id} \mid \exists\ (\mathit{id},\ \sigma) \in \mathit{ME} \vee \exists\ ((\mathit{id}, \sigma), p, a) \in \mathit{ME}\} \\ \mathit{FuncSet} & = & \{\ \mathit{id} \mid \exists\ (\mathit{id},\ \sigma) \in \mathit{FE} \vee \exists\ ((\mathit{id}, \sigma), p, a) \in \mathit{FE}\} \\ \text{Dom}(\Sigma) & = & \mathit{IdSet} \cup \mathit{SconSet} \cup \mathit{MapSet} \cup \mathit{FuncSet} \cup \text{Dom}(\mathit{SSE}) \\ \mathit{FSV}(\tau) & = & \text{the set of sort variables occurring freely in } \tau \end{array}$$

1. ClassEnv

   - Each identifier occurring in *Subset* must be in *IdSet.* Formally:
     $\forall (x, y) \in \mathit{Subset}.\ (x \in \mathit{IdSet}) \wedge (y \in \mathit{IdSet})$

2. SConEnv

   - Each sort constructor identifier occurring in *SconType* must be in *SconSet.* Formally:
     $\forall (x, t) \in \mathit{SconType}.\ x \in \mathit{SconSet}.$
   - The arity of each sort constructor occurring in *SconType* must be unique. Formally:

$\forall x \in \mathit{SconSet}.\ (x, (c_{11}, \ldots, c_{1k}, c_1)) \in \mathit{SconType} \wedge (x, (c_{21}, \ldots, c_{2l}, c_2)) \in \mathit{SconType} \Rightarrow k = l.$

- *SconType* must be coregular. Formally[1]:
  For each $id \in \mathit{SconSet}$ and $c \in \mathit{IdSet}$ the set $\{c^k \mid \exists (c, d) \in \mathit{Subset}^*.\ (id, (c^k, d)) \in \mathit{SconType}\}$ is either $\emptyset$ or has a greatest element w.r.t. the extension of Subset^* on tuples.
- Each class identifier occurring in *SconType* must be in *IdSet*. Formally:
  $\forall (x, ((c_1, \ldots, c_k), c)) \in \mathit{SconType}.\ c \in \mathit{IdSet} \wedge \{c_i \mid 1 \leq i \leq k\} \subseteq \mathit{IdSet}$

3. SSynEnv

- The identifiers sid_i $, 1 \leq i \leq n$, must be sort variables. Formally:
  $\forall (sid, ((sid_1, \ldots, sid_n), \tau)) \in SSE.$
  $sid_i \notin \mathit{SConSet} \cup \mathit{Dom}(SSE)$ $, 1 \leq i \leq n.$
- Each sort variable occurring free in τ must be an identifier sid_i. Formally:
  $\forall (sid, ((sid_1, \ldots, sid_n), \tau)) \in SSE.$
  $FSV(\tau) \subseteq \cup_{1 \leq i \leq n} \{sid_i\}$
- Sort synonyms must not be defined recursively. Formally:
  $\forall \tau \in \mathit{Types}.\ \exists \tau' \in \mathit{Types}.\ \ \tau \rightarrow_\beta^* \tau' \not\rightarrow_\beta^+ \tau''$
  where $sid\ \tau_1 \ldots \tau_n \rightarrow_\beta \{\alpha_1 \rightarrow \tau_1, \ldots, \alpha_n \rightarrow \tau_n\}\tau$ iff
  $SSE(sid) = ((\alpha_1, \ldots, \alpha_n), \tau)$
  and $\rightarrow_\beta^*$ is the reflexive, transitive extension of $\rightarrow_\beta$ to sort expression $\tau \in \mathit{Types}$
  and $\rightarrow^+$ is the transitive extension of $\rightarrow_\beta$ to sort expression $\tau \in \mathit{Types}$
- The identifiers sid_i $, 1 \leq i \leq n$, must be pairwise disjoint. Formally:
  $\forall i, j. 1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \neq j \Rightarrow sid_i \neq sid_j.$

4. MapEnv

- *ME* must be a finite function w.r.t. the mapping identifiers. Formally[2]:
  $(i, m) \in \mathit{ME} \wedge (i, n) \in \mathit{ME} \Rightarrow m \overset{\alpha}{=} n$

5. FuncEnv

- *FE* must be a finite function w.r.t. the function identifiers. Formally:
  $(i, n) \in \mathit{FE} \wedge (i, n) \in \mathit{FE} \Rightarrow m \overset{\alpha}{=} n$

6. SVarEnv

- No identifier occurring in the signature is allowed as a bound identifier. Formally:
  $\mathit{Dom}(SVE) \cap \mathrm{Dom}(\Sigma) = \emptyset$
- Each class identifier occurring in *SVE* is in *IdSet*. Formally:
  $\forall id.\ id \in \mathrm{Cod}(SVE) \Rightarrow id \in \mathit{IdSet}$

7. Disjointness of Identifier Classes

- The sets of class identifiers, sort identifiers, mapping identifiers and function identifiers must be pairwise disjoint. Formally:
  IdSet, SconSet, MapSet, FuncSet pairwise disjoint

[1] Subset^* denotes the reflexive and transitive closure of the relation.
[2] $\overset{\alpha}{=}$ means equal up to renaming of sort variables.

Axioms

An axiom is either a formula block or a generation statement. A formula block consists of an environment and a set of formulae. The environment contains the local sort variables and variables. The generation axiom contains the constructed sorts together with their constructors.

$$\begin{array}{rcccl}
\mathrm{ax} & \in & \mathrm{Ax} & = & \mathrm{FormAx} \cup \mathrm{GenAx} \\
(\mathit{AE, formulae}) & \in & \mathrm{FormAx} & = & \mathrm{AxEnv} \times \mathrm{Fin(Formula)} \\
(\mathit{sois, cons}) & \in & \mathrm{GenAx} & = & \mathrm{Fin(SconId)} \times \mathrm{Fin(Id)} \\
(\Delta, \Gamma) & \in & \mathrm{AxEnv} & = & \mathrm{SVarEnv} \times \mathrm{VarEnv} \\
\Gamma & \in & \mathrm{VarEnv} & = & \mathrm{Id} \overset{fin}{\to} \mathrm{Type} \\
\mathit{formula} & \in & \mathrm{Formula} & = & \mathrm{abs(<exp1>)}
\end{array}$$

Well Formed Specifications

A specification is well formed if its signature is well formed and if the following condition holds:

- No identifier occurring in the signature is allowed as a bound identifier. Formally:
  $\mathrm{Dom}(\Gamma) \cap \mathrm{Dom}(\Sigma)=\emptyset$

3.1.3 Translation Rules

Now we give the translation rules for the language in the small. Context conditions that cannot be described conveniently with the rules are given separately. To increase readability we add the associated context free rules. In order to simplify the translation rules, the context free rules sometimes slightly differ from the productions given in Section 3.1.1. However, it is easy to prove that both variants always define the same language.

Specification Body

$$\begin{array}{lcl}
\langle\mathrm{specbody}\rangle & ::= & \{\ \langle\mathrm{decls}\rangle\ \} \\
\langle\mathrm{decls}\rangle & ::= & \varepsilon \mid \langle\mathrm{decl}\rangle\ \langle\mathrm{decls}\rangle \\
\langle\mathrm{decl}\rangle & ::= & \langle\mathrm{signature}\rangle\ ; \mid \langle\mathrm{axioms}\rangle\ ;
\end{array}$$

$$\frac{\mathit{decls} \Rightarrow \Sigma}{\mathit{specbody} \Rightarrow \Sigma}$$

$$\frac{\mathit{decl} \Rightarrow \Sigma_1 \quad [\mathit{decls} \Rightarrow \Sigma_2]}{\mathit{decls} \Rightarrow \Sigma_1[\cup\Sigma_2]}$$

$$\frac{}{\mathit{axioms} \Rightarrow \emptyset}$$

The signatures of the individual declarations are unified. The axioms part does not contain any signature information.

Signature Declarations

Classes

$$\begin{array}{lcl}
\langle\mathrm{signature}\rangle & ::= & \textbf{class}\ \langle\mathrm{id}\rangle\ [\textbf{subclass of}\ \{\langle\mathrm{id}\rangle\ //\ ,\}^+] \\
 & \mid & \langle\mathrm{id}\rangle\ \textbf{subclass of}\ \{\langle\mathrm{id}\rangle\ //\ ,\}^+
\end{array}$$

$$\frac{}{\textbf{class } id \Rightarrow ((\{id\}, \{(id, CPO)\}), \emptyset, \emptyset, \emptyset, \emptyset)}$$

$$\frac{}{\textbf{class } id \textbf{ subclass of } \{id_{\mathrm{i}} //,\}^{+} \Rightarrow ((\{id\}, \bigcup_{\mathrm{i}} \{(id, id_{\mathrm{i}})\}), \emptyset, \emptyset, \emptyset, \emptyset)}$$

$$\frac{}{id \textbf{ subclass of } \{id_{\mathrm{i}} //,\}^{+} \Rightarrow ((\emptyset, \bigcup_{\mathrm{i}} \{(id,\ id_{\mathrm{i}})\}), \emptyset, \emptyset, \emptyset, \emptyset)}$$

Each defined class identifier is stored in the class environment. In addition the subclass information is stored. By default, each class is a subclass of CPO.

Sort Constructors

⟨signature⟩	::=	**sort** ⟨sid⟩ { ⟨sid⟩* | **::** ⟨classexp⟩ }
	|	{⟨sid⟩ // **,**}+ **::** ⟨classexp⟩
⟨classexp⟩	::=	[({⟨id⟩ // **,**}+)] ⟨id⟩

$$\frac{sid_{\mathrm{i}} \notin \mathrm{dom}(\Sigma) \quad 1 \leq \mathrm{i} \leq \mathrm{n} \quad \mathrm{n} \geq 0}{\textbf{sort } sid\ sid_1 \ldots sid_{\mathrm{n}} \Rightarrow (\emptyset, (\{sid\}, \{(sid, (\underbrace{\mathrm{CPO}, \ldots, \mathrm{CPO}}_{\mathrm{n}}), \mathrm{CPO})\}), \emptyset, \emptyset, \emptyset)}$$

$$\frac{\mathrm{n} \geq 1}{\textbf{sort } sid\text{:: } (id_1, \ldots, id_{\mathrm{n}})id \Rightarrow (\emptyset, (\{sid\}, \{(sid, (\mathrm{CPO}, \ldots, \mathrm{CPO}), \mathrm{CPO}), (sid, (id_1, \ldots, id_{\mathrm{n}}), id)\}), \emptyset, \emptyset, \emptyset)}$$

$$\frac{}{\textbf{sort } sid :: id \Rightarrow (\emptyset, (\{sid\}, \{(sid, (), \mathrm{CPO}), (sid, (), id)\}), \emptyset, \emptyset, \emptyset)}$$

$$\frac{\mathrm{n} \geq 1}{\{sid_{\mathrm{i}} //,\}^{+}\text{::}(id_1, \ldots, id_{\mathrm{n}})id \Rightarrow (\emptyset, (\emptyset, \bigcup_{\mathrm{i}} \{(sid_{\mathrm{i}}, (id_1, \ldots, id_{\mathrm{n}}), id)\}), \emptyset, \emptyset, \emptyset)}$$

$$\frac{}{\{sid_{\mathrm{i}} //,\}^{+} :: id \Rightarrow (\emptyset, (\emptyset, \bigcup_{\mathrm{i}} \{sid_{\mathrm{i}}, (), id)\}), \emptyset, \emptyset, \emptyset)}$$

The sort constructors are stored together with their class information. Each sort constructor can be applied to all sorts because each sort is an element of class CPO. Then the result class is always CPO.

Sort Synonyms

⟨signature⟩	::=	**sortsyn** {⟨sid⟩}+ = ⟨sortexp⟩

$$\frac{}{\textbf{sortsyn } sid\ sid_1 \ldots sid_n{=}\tau \Rightarrow (\emptyset, \emptyset, \{sid \rightarrow ((sid_1, \ldots, sid_n), \tau)\}, \emptyset, \emptyset)}$$

Including a Signature

⟨signature⟩	::=	**SIG** **(** ⟨specexp⟩)

$$\frac{specexp \Rightarrow \Sigma}{\textbf{SIG } (specexp) \Rightarrow \Sigma}$$

Mapping

$$\begin{array}{lcl}\langle\text{signature}\rangle & ::= & \{\langle\text{id}\rangle\ //\ ,\}^{+}\ \textbf{:}\ [\langle\text{context}\rangle]\ \langle\text{sortexp}\rangle\ \textbf{to}\ \langle\text{sortexp}\rangle \\ & | & \{\langle\text{inf-id}\rangle\ //\ ,\}^{+}\ \textbf{:}\ [\langle\text{context}\rangle]\ \langle\text{sortexp}\rangle \times \langle\text{sortexp}\rangle\ \textbf{to}\ \langle\text{sortexp}\rangle\ [\langle\text{prio}\rangle]\end{array}$$

$$\frac{[\mathit{context} \Rightarrow \mathit{SVE}\,]_{\emptyset}}{\{\mathit{id}_{\mathrm{i}}//,\}^{+}{:}[\mathit{context}]\mathit{sortexp}_1\ \textbf{to}\ \mathit{sortexp}_2 \Rightarrow (\emptyset,\emptyset,\emptyset,\bigcup_{\mathrm{i}}\{\mathit{id}_{\mathrm{i}},\mathit{SVE},\mathit{sortexp}_1 \to \mathit{sortexp}_2)\},\emptyset)}$$

$$\frac{[\mathit{context} \Rightarrow \mathit{SVE}]_{\emptyset} \quad [\ \mathit{prio} \Rightarrow (n,a)]_{(0,\mathrm{none})}}{\{.\mathit{id}_{\mathrm{i}}.//,\}^{+}{:}[\mathit{context}]\mathit{sortexp}_1{\times}\mathit{sortexp}_2\ \textbf{to}\ \mathit{sortexp}_3[\ \mathit{prio}] \Rightarrow}$$
$$(\emptyset,\emptyset,\emptyset,\bigcup_{\mathrm{i}}\{(\mathit{id}_{\mathrm{i}},\mathit{SVE},\mathit{sortexp}_1{\times}\mathit{sortexp}_2\ \to\ \mathit{sortexp}_3,n,a)\},\emptyset)$$

In the mapping environment the syntactic **to** is replaced by the function space constructor. From a syntactic point of view there is no difference between these two identifiers. If the priority information is missing for infix identifiers we assume priority 0 and no associativity.

Functions

$$\begin{array}{lcl}\langle\text{signature}\rangle & ::= & \{\langle\text{id}\rangle\ //\ ,\}^{+}\ \textbf{:}\ [\langle\text{context}\rangle]\ \langle\text{sortexp}\rangle \\ & | & \{\langle\text{inf-id}\rangle\ //\ ,\}^{+}\ \textbf{:}\ [\langle\text{context}\rangle]\ \langle\text{sortexp}\rangle \times \langle\text{sortexp}\rangle \longrightarrow \langle\text{sortexp}\rangle\ [\langle\text{prio}\rangle]\end{array}$$

$$\frac{[\mathit{context} \Rightarrow \mathit{SVE}]_{\emptyset}}{\{\mathit{id}_{\mathrm{i}}//,\}^{+}{:}[\mathit{context}]\mathit{sortexp} \Rightarrow (\emptyset,\emptyset,\emptyset,\emptyset,\bigcup_{\mathrm{i}}\{(\mathit{id}_{\mathrm{i}},\mathit{SVE},\ \mathit{sortexp})\})}$$

$$\frac{[\mathit{context} \Rightarrow \mathit{SVE}\,]_{\emptyset} \quad [\ \mathit{prio}\ ] \Rightarrow (\mathrm{n},\ \mathrm{a})]_{(0,\mathrm{none})}}{\{.\mathit{id}_{\mathrm{i}}.//,\}^{+}{:}[\mathit{context}]\mathit{sortexp}_1{\times}\mathit{sortexp}_2\ \to\ \mathit{sortexp}_3[\ \mathit{prio}\ ] \Rightarrow}$$
$$(\emptyset,\emptyset,\emptyset,\emptyset,\bigcup_{\mathrm{i}}\{(\mathit{id}_{\mathrm{i}},\mathit{SVE},\mathit{sortexp}_1{\times}\mathit{sortexp}_2 \to \mathit{sortexp}_3,n,a)\})$$

Priority

$$\langle\text{prio}\rangle\ ::=\ \textbf{prio}\ \langle\text{num}\rangle\ [\textbf{:}\ \{\textbf{left}|\textbf{right}\}]$$

$$\frac{}{\textbf{prio}\ \mathit{num} \Rightarrow (\mathrm{val}(\mathit{num}),\mathrm{none})}$$

$$\frac{}{\textbf{prio}\ \mathit{num}{:}\textbf{left} \Rightarrow (\mathrm{val}(\mathit{num}),\mathrm{left})}$$

$$\frac{}{\textbf{prio}\ \mathit{num}{:}\textbf{right} \Rightarrow (\mathrm{val}(\mathit{num}),\mathrm{right})}$$

Sort Contexts

Sort Contexts

$$\begin{array}{lcl}\langle\text{context}\rangle & ::= & \{\langle\text{scontext}\rangle\ //\ ,\}^{+}\ \Rightarrow \\ \langle\text{scontext}\rangle & ::= & \{\langle\text{sid}\rangle\ //\ ,\}^{+}\ \textbf{::}\ \langle\text{id}\rangle\end{array}$$

$$\frac{}{\{\mathit{sid}_{\mathrm{i}}//,\}^{+}{::}\mathit{id} \Rightarrow \bigcup_{\mathrm{i}}\{\mathit{sid}_{\mathrm{i}} \mapsto \mathit{id}\}}$$

$$\frac{\mathit{scontext} \Rightarrow \mathit{SVE}_1 \quad [\mathit{context} \Rightarrow \mathit{SVE}_2]}{\mathit{scontext}[,\mathit{context}] \Rightarrow \mathit{SVE}_1[\cup \mathit{SVE}_2]} \ \{[\mathit{dom}(\mathit{SVE}_1) \cap \mathit{dom}(\mathit{SVE}_2) = \emptyset]$$

Note that the domains of the unified sort variable environments must be disjunct.

Generated By

⟨axioms⟩	::=	⟨simplesorts⟩ **generated by** ⟨opns⟩
⟨simplesorts⟩	::=	$\{$⟨sid⟩ $\{$⟨sid⟩$\}^*$ // ,$\}^+$
⟨opns⟩	::=	$\{$⟨opn⟩ // ,$\}^+$
⟨opn⟩	::=	⟨id⟩ | ⟨inf-id⟩

$$\frac{\Sigma \vdash \mathit{simplesorts} \Rightarrow \mathit{sois} \quad \Sigma \vdash \mathit{opns} \Rightarrow \mathit{cons}}{\Sigma \vdash \mathit{simplesorts}\ \textbf{generated by}\ \mathit{opns} \Rightarrow (\mathit{sois},\ \mathit{cons})} \ \{\text{see context conditions given below}$$

$$\frac{}{\Sigma \vdash \{\mathit{sid}_{\mathrm{i}}\{\mathit{sid}_{\mathrm{ij}}\}^*//,\}^+ \Rightarrow \bigcup_{\mathrm{i}} \{\mathit{sid}_{\mathrm{i}}\}}$$

$$\frac{}{\Sigma \vdash \{[.]\mathit{id}_{\mathrm{i}}[.]//,\}^+ \Rightarrow \bigcup_{\mathrm{i}}\{\mathit{id}_{\mathrm{i}}\}}$$

© The identifier $\mathit{sid}_{\mathrm{i}}$ must be defined as a sort constructor. Formally:
$\mathit{sid}_{\mathrm{i}} \in \mathit{SConSet}$

© The identifier $\mathit{sid}_{\mathrm{ij}}$ must be a sort variable. Formally:
$\mathit{sid}_{\mathrm{ij}} \notin \mathrm{dom}(\Sigma)$

© The identifier id_{i} must be defined as a function if the optional dots are missing. Formally:
$(\mathit{id}_{\mathrm{i}}, \sigma) \in \mathit{FE}$ for some σ

© The identifier id_{i} must be defined as an infix function if id_{i} is included in .'s. Formally:
$(\mathit{id}_{\mathrm{i}}, \sigma, n, a) \in \mathit{FE}$ for some σ, n, a

© The result sort of the constructors in *cons* must be a sort of interest from *sois.* Formally:
$\forall \mathit{id} \in \mathit{cons}.\ \exists \mathit{sid} \in \mathit{sois}$
$(\mathit{id} : (\mathit{SVE}, \tau \rightarrow \mathit{sid}\,\mathit{sid}_1 \ldots \mathit{sid}_{\mathrm{n}}) \in \Sigma\ \vee$
$\mathit{id} : (\mathit{SVE}, \mathit{sid}\,\mathit{sid}_1 \ldots \mathit{sid}_{\mathrm{n}}) \in \Sigma)$
$\wedge\ \mathit{sid}_{\mathrm{i}} \notin \mathit{SConSet}\ 1 \leq \mathrm{i} \leq \mathrm{n}$

© All sort variables occurring in the argument sort of a constructor from *cons* must occur in the result sort of the constructor. Formally:
$\forall \mathit{id} \in \mathit{cons}.$
$(\mathit{id} : (\mathit{SVE}, \tau_1 \rightarrow \tau_2)) \in \Sigma$
$\Rightarrow \mathrm{FSV}(\tau_1) \subseteq \mathrm{FSV}(\tau_2)$

© There must be a constructor in *cons* for each sort of interest from *sois.* Formally:
$\forall \mathit{sid} \in \mathit{sois}.\ \exists \mathit{id} \in \mathit{cons}$
$(\mathit{id} : (\mathit{SVE},\ \tau \rightarrow \mathit{sid}\ \mathit{sid}_1 \ldots \mathit{sid}_{\mathrm{n}}) \in \Sigma\ \vee$
$\mathit{id} : (\mathit{SVE},\ \mathit{sid}\ \mathit{sid}_1 \ldots \mathit{sid}_{\mathrm{n}}) \in \Sigma)$
$\wedge\ \mathit{sid}_{\mathrm{i}} \notin \mathit{SConSet}\ 1 \leq \mathrm{i} \leq \mathrm{n}$

Ⓒ Each sort of interest from *sois* is only allowed to occur at the uppermost level of the argument sort of a constructor. Formally:

$\forall id \in cons\ .\forall sid \in sois$
$(id : (SVE, \tau_1 \times \ldots \times \tau_n \to \tau)) \in \Sigma, n \geq 1$
$\Rightarrow ((\exists \alpha_1, \ldots, \alpha_m.\tau_i = sid\ \alpha_1 \ldots \alpha_m \wedge \alpha_j \notin SConSet\ 1 \leq j \leq m)$
$\vee \neg occurs(sid, \tau_i))\ \ 1 \leq i \leq n$

3.1.4 The Sort Inference

SPECTRUM has a static sort system. However, a context free description as given in Section 3.1.1 cannot completely define a statically sorted language. Sorting constraints are typically context sensitive. Therefore we will define the well-sorted axioms using a logical calculus. To simplify the description of well-sortedness for axioms blocks we translate axioms blocks to equivalent ones in the following way:

axioms $[context]\ \{\forall|\forall^\perp\}\ opdecls_1 \ldots \{\forall|\forall^\perp\}\ opdecls_n$ **in**
$[\{id_1\}]\ exp_1;$
$\vdots$
$[\{id_m\}]\ exp_m;$
endaxioms

is translated to

axioms $[context]$ **in** $\{\forall|\forall^\perp\}\ opdecls_1. \ldots .\{\forall|\forall^\perp\}\ opdecls_n.$
$exp_1 \wedge \ldots \wedge exp_m;$
endaxioms

Ⓒ Each axioms block must be well-sorted. Formally:
- **axiom in** *exp* **endaxioms**: for some SVE. $SVE, \emptyset \rhd_\Sigma exp :: \text{Bool}$
- **axiom** *context* **in** *exp* **endaxioms**: $SVE, \emptyset \rhd_\Sigma exp :: \text{Bool}$
  where $context \Rightarrow SVE$

Note that we use σ both for sort variable substitutions and for their extensions to sort expressions. *FI(exp)* denotes the set of identifiers occurring freely in expression *exp*.

$$\frac{}{\Delta, \Gamma \rhd_\Sigma id :: \Gamma(id)}\ (var)$$

$$\frac{\forall \alpha \in dom(SVE).\ \Delta \vdash_\Sigma \sigma(\alpha) : SVE(\alpha)}{\Delta, \Gamma \rhd_\Sigma id :: \sigma(\tau)}\ (const) \begin{cases} FE(id) = (SVE, \tau) \\ \sigma : FSV(\tau) \overset{fin}{\to} Type \end{cases}$$

$$\frac{\forall \alpha \in dom(SVE).\ \Delta \vdash_\Sigma \sigma(\alpha) : SVE(\alpha)}{\Delta, \Gamma \rhd_\Sigma .id. :: \sigma(\tau)}\ (iconst) \begin{cases} FE(id) = ((SVE, \tau), n, a) \\ \sigma : FSV(\tau) \overset{fin}{\to} Type \end{cases}$$

$$\frac{\Delta, \Gamma \rhd_\Sigma exp :: \tau}{\Delta, \Gamma \rhd_\Sigma (exp) :: \tau}\ (paranthesis)$$

$$\frac{\Delta, \Gamma \rhd_\Sigma exp_i :: \tau\ \ 2 \leq i \leq n}{\Delta, \Gamma \rhd_\Sigma (exp_1, \ldots, exp_n) :: \tau_1 \times \ldots \times \tau_n}\ (tuple)$$

$$\frac{\Delta, \Gamma \rhd_\Sigma exp :: \tau}{\Delta, \Gamma \rhd_\Sigma exp : \tau :: \tau}\ (constrained)$$

$$\frac{\Delta, \Gamma \rhd_\Sigma exp_1 :: \tau_1 \to \tau_2 \quad \Delta, \Gamma \rhd_\Sigma exp_2 :: \tau_1}{\Delta, \Gamma \rhd_\Sigma exp_1\ exp_2 :: \tau_2}\ (appl)$$

$$\frac{\Delta,\Gamma \rhd_\Sigma exp_1 :: \tau_1 \quad \Delta,\Gamma \rhd_\Sigma exp_2 :: \tau_2}{\Delta,\Gamma \rhd\ exp_1\ id[:\sigma(\tau)]\ exp_2 :: \tau_3}\ (infix) \begin{cases} ME(id) = ((SVE,\tau),n,a) \\ \sigma : FSV(\tau) \overset{fin}{\to} Type \\ \forall \alpha \in dom(SVE). \\ \Delta \vdash_\Sigma \sigma(\alpha) : SVE(\alpha) \\ \tau = \tau_1 \times \tau_2 \to \tau_3 \end{cases}$$

$$\frac{\Delta,\Gamma.id \to \tau_1 \rhd_\Sigma exp :: \tau_2}{\Delta,\Gamma \rhd\ \lambda\ id[:\tau_1].\ exp :: \tau_1 \to \tau_2}\ (abstr1) \left\{\ FI(exp) \cap Dom(ME) = \emptyset \right.$$

$$\frac{\Delta,\Gamma.id_1 \to \tau_1.\ \ldots id_n \to \tau_n \rhd_\Sigma exp :: \tau}{\Delta,\Gamma \rhd_\Sigma \lambda\ (id_1[:\tau_1],\ldots,id_n[:\tau_n]).\ exp :: \tau_1 \times \ldots \times \tau_n \to \tau}\ (abstr2) \begin{cases} id_i\ pairwise\ disjoint \\ n \geq 2 \\ FI(exp) \cap Dom(ME) = \emptyset \end{cases}$$

$$\frac{\Delta,\Gamma.\ id_1 \to \tau_1,\ldots,id_n \to \tau_n \rhd_\Sigma exp :: Bool}{\Delta,\Gamma \rhd_\Sigma Q\ id_1.\ \ldots\ .id_n.\ exp :: Bool}\ (quant_1) \begin{cases} Q \in \{\forall,\forall^\perp,\exists,\exists^\perp\} \\ id_i\ pairwise\ disjoint \end{cases}$$

$$\frac{\Delta,\Gamma.\ id_1 \to \tau_1.\ \ldots\ .id_{n_1} \to \tau_1.\ \ldots\ .id_m \to \tau_m.\ \ldots\ .id_{n_m} \to \tau_m \rhd_\Sigma exp :: Bool}{\Delta,\Gamma \rhd_\Sigma Q\ id_1,\ldots,id_{n_1} : \tau_1,\ldots,id_m,\ldots,id_{n_m} : \tau_m.\ exp :: Bool}\ (quant_2)$$

$$\begin{cases} Q \in \{\forall,\forall^\perp,\exists,\exists^\perp \\ id_{ij}\ pairwise\ disjoint \end{cases}$$

$$\frac{\Delta,\Gamma \rhd_\Sigma exp :: \sigma(\tau_1)}{\Delta,\Gamma \rhd_\Sigma id\ exp :: \sigma(\tau_2)}\ (mappl) \begin{cases} ME(id) = (SVE,\tau) \\ \sigma : FSV(\tau) \overset{fin}{\to} Type \\ \forall \alpha \in Dom(SVE). \\ \Delta \vdash_\Sigma \sigma(\tau) : SVE(\alpha) \\ \tau = \tau_1 \to \tau_2 \end{cases}$$

$$\frac{\Delta,\Gamma \rhd_\Sigma exp :: \sigma(\tau_1)}{\Delta,\Gamma \rhd_\Sigma .id.\ exp :: \sigma(\tau_2)}\ (imappl) \begin{cases} ME(id) = (SVE,\tau) \\ \sigma : FSV(\tau) \overset{fin}{\to} Type \\ \forall \alpha \in Dom(SVE). \\ \Delta \vdash_\Sigma \sigma(\tau) : SVE(\alpha) \\ \tau = \tau_1 \to \tau_2 \end{cases}$$

$$\frac{\Delta,\Gamma \rhd_\Sigma exp_1 :: \sigma(\tau_1) \quad \Delta,\Gamma \rhd_\Sigma exp_2 :: \sigma(\tau_2)}{\Delta,\Gamma \rhd_\Sigma exp_1\ id[:\sigma(\tau)]\ exp_2 :: \sigma(\tau_3)}\ (minfix) \begin{cases} ME(id) = ((SVE,\tau),n,a) \\ \sigma : FSV(\tau) \overset{fin}{\to} Type \\ \forall \alpha \in dom(SVE). \\ \Delta \vdash_\Sigma \sigma(\alpha) : SVE(\alpha) \\ \tau = \tau_1 \times \tau_2 \to \tau_3 \end{cases}$$

$$\frac{\Delta,\Gamma \rhd_\Sigma exp :: \tau_1}{\Delta,\Gamma \rhd_\Sigma exp :: \tau_2}\ (sortsyn) \left\{\ \tau_1 \to_\beta^* \tau_2 \right.$$

The side condition in rules *(abstr1)* and *(abstr2)* ensures that no mapping is used in the body of a λ-abstraction. Otherwise a λ-abstraction may denote a non-continuouse function. This is, however, not allowed in SPECTRUM.

3.1.5 Class Inference

In SPECTRUM sorts are classified by classes. Therefore we need a second calculus to define the judgement $\tau : C$, stating that sort τ belongs to class C.

$$\frac{}{\Delta \vdash_\Sigma \alpha : C}\ (svar) \left\{\ C \in \Delta(\alpha) \right.$$

$$\frac{\Delta \vdash_\Sigma \tau_i : C_i \quad 1 \leq i \leq n}{\Delta \vdash_\Sigma sid\ \tau_1 \ldots \tau_n : C}\ (sconappl) \begin{cases} n \geq 0 \\ (sid, ((C_1, \ldots, C_n), C) \in SconType \end{cases}$$

$$\frac{\Delta \vdash_\Sigma \tau : C_1}{\Delta \vdash_\Sigma \tau : C_2}\ (subclass) \{\ (C_1, C_2) \in Subset$$

3.1.6 Infix Symbols

The context free syntax given in Section 3.1.1 is ambiguous with respect to expressions because of the user definable infix identifiers. Thus, parsing an expression possibly yields a set of semantic objects. The following context condition uniquely selects the object with the desired binding.

Ⓒ The binding of infix identifiers must be correct with respect to the given priority and associativity. Formally:

$$\begin{array}{l} \forall\ exp \in Formula\ .\ \ \exists n \in Prio \cup \{\infty\},\ a \in Assoc \\ exp \Rightarrow (n, a) \end{array}$$

where $exp \Rightarrow (n, a)$ is inductively defined on the structure of exp in the following way:

$$\frac{exp_1 \Rightarrow (n_1, a_1) \quad exp_2 \Rightarrow (n_2, a_2)}{exp_1\ id\ exp_2 \Rightarrow (n, a)}$$

iff $\begin{array}{l} (FE(id) = (\sigma, n, a) \vee ME(id) = (\sigma, n, a)) \wedge \\ (n_1 > n \wedge n_2 > n \vee \\ n_1 = n \wedge n_2 > n \wedge a_1 = a = left \vee \\ n_1 > n \wedge n_2 = n \wedge a_2 = a = right \vee \\ n_1 = n = n_2 \wedge a_1 = left \wedge a = none \wedge a_2 = right) \end{array}$

$$\frac{exp_1 \Rightarrow (n_1, a_1) \ldots exp_m \Rightarrow (n_m, a_m)}{exp \Rightarrow (\infty, none)} \begin{cases} \text{for all other inductive cases,} \\ \text{where } exp_1 \ldots exp_m,\ m \geq 0 \\ \text{are all direct} \\ \text{subexpressions of } exp \end{cases}$$

3.2 In the Large

3.2.1 Context Free Syntax

⟨system⟩	::=	⟨syspart⟩ [system]
⟨syspart⟩	::=	⟨alphanumid⟩ = { ⟨specexp⟩ | ⟨sigmorph⟩ | ⟨specabstr⟩ }

Signature Morphisms

⟨sigmorph⟩	::=	[⟨rename-list⟩]
⟨rename-list⟩	::=	⟨rename⟩ [, ⟨rename-list⟩]
⟨rename⟩	::=	⟨inf-id⟩ **to** ⟨inf-id⟩ | ⟨id⟩ **to** ⟨id⟩ | ⟨sid⟩ **to** ⟨sid⟩
⟨morph⟩	::=	⟨alphanumid⟩ | ⟨sigmorph⟩

Structured Specifications

⟨specexp⟩	::=	⟨aspecexp⟩ [+ ⟨specexp⟩]
	|	**abstr** (⟨arg-lst⟩)
⟨arg-lst⟩	::=	⟨specexp⟩ [**via** ⟨morph⟩] [, ⟨arg-lst⟩]
⟨aspecexp⟩	::=	**rename** ⟨specexp⟩ **by** ⟨morph⟩
	|	(⟨specexp⟩)
	|	**hide** ⟨sigel-set⟩ **in** ⟨aspecexp⟩
	|	**export** ⟨sigel-set⟩ **in** ⟨aspecexp⟩
	|	**{ enriches** ⟨specexp⟩; ⟨decls⟩ **}**
	|	⟨aspecexp⟩ | ⟨specbody⟩
⟨sigel-set⟩	::=	⟨sigel⟩ [, ⟨sigel-set⟩]
⟨sigel⟩	::=	⟨opn⟩ | ⟨sid⟩ | **SIG** (⟨specexp⟩)

Parameterized Specifications

⟨abstr⟩	::=	⟨alphanumid⟩ | ⟨specabstr⟩
⟨specabstr⟩	::=	**param** ⟨spec-list⟩**body** ⟨aspecexp⟩
⟨spec-list⟩	::=	⟨alphanumid⟩ = ⟨specexp⟩ [, ⟨spec-list⟩]

3.2.2 Compound Semantic Objects

A specification text consists of a set of (parameterized) specifications and signature morphisms (the system parts) together with their declared names. Its abstraction is therefore a *specification environment*

$c \in SpecEnv = Alphanumid \to SysPart$

i.e. a finite mapping from identifiers to system parts.

The abstraction ρ of a signature morphism is again a finite mapping, from identifiers to identifiers. Abstract signature morphisms are used not only to rename specifications but also to implement hiding. A symbol to be hidden is renamed to a fresh name, not available at the user level. As a consequence, this symbol acts as if it was existentially bound: it cannot be used in any enclosing specification and does not collide with any other hidden symbol. Fresh names are taken from the set of identifiers *HiddenId* which is disjoint from the set of concrete syntax identifiers *VisibleId*. The set of all identifiers is *AllId*.

To generate fresh hidden symbols we keep track in the base

$(HIS, C) \in Base = HiddenIdSet \times SpecEnv$

of the set *HIS* of hidden symbols generated so far in the specification text. A hidden identifier *hid* is then fresh if $hid \notin HIS$. For each specification SP, the function $hids(SP)$ returns the set of hidden symbols in SP.

The abstraction of a specification was defined in the previous section. It is a pair consisting of a signature and a set of axiom blocks (a flat specification). The abstraction of a parameterized specification

$SA \in PSpecification = SpecParam \times Specexp$

is a tuple consisting of the list

$SpP \in SpecParam = List(Alphanumid \times Specification)$

of formal parameter specifications and the specification expression of the body. We use a list instead of finite mapping for the formal parameters because the order of the actual parameter list

$SpA \in SpecArgs = List(Sigmorph \times Specification)$

must match the order of the formal parameters. Actual parameters are required to match syntactically the formal parameters modulo renaming i.e.

if $(\rho_i, SpA_i) \in SpA$ then $\rho_i(\Sigma(SpP_i)) \subseteq \Sigma(SpA_i)$.

The definition of compound semantic objects for structured specifications is summarized below:

$$\begin{array}{rcll}
(HIS, C) \text{ or } B & \in & Base & = HiddenIdSet \times SpecEnv \\
C & \in & SpecEnv & = Alphanumid \overset{fin}{\to} SysPart \\
 & & SysPart & = SigMorph \cup Specification \cup PSpecification \\
\rho & \in & SigMorph & = AllId \overset{fin}{\to} AllId \\
SA & \in & PSpecification & = SpecParam \times Specexp \\
SpP & \in & SpecParam & = List(Alphanumid \times Specification) \\
SpA & \in & SpecArgs & = List(Sigmorph \times Specification) \\
 & & & \\
VId & \in & VisibleId & = Sid \cup Op \\
HId & \in & HiddenId & = \text{a set disjoint from } VisibleId \\
AId & \in & AllId & = VisibleId \cup HiddenId \\
 & & & \\
VIS & \in & VisibleIdSet & = Fin(VisibleId) \\
HIS & \in & HiddenIdSet & = Fin(HiddenId)
\end{array}$$

We are now ready to give the translation rules. To increase their readability, each rule is given together with its associated context free production. Context checks which cannot be described conveniently with the rules are given separately.

3.2.3 Translation Rules

System Specification

⟨system⟩ ::= ⟨syspart⟩ [system]

⟨syspart⟩ ::= ⟨alphanumid⟩ = { ⟨specexp⟩ | ⟨sigmorph⟩ | ⟨specabstr⟩ }

$$\frac{B \cup SY \vdash syspart \Rightarrow P \quad [B \cup P \vdash system \Rightarrow SY]}{B \vdash syspart\ [system] \Rightarrow P\ [\ \cup\ SY]} \left\{ \begin{array}{l} \text{no cyclic reference occurs} \\ \text{in } syspart\ [system] \end{array} \right.$$

The system part *syspart* can contain forward references to system part names from *system*. Therefore it is necessary to add the abstract representation SY of *system* to the current environment when translating *system*. However, references are not allowed to be cyclic.

$$\frac{B \vdash specexp \Rightarrow SP}{B \vdash alphanumid = specexp \Rightarrow (hids(SP), \{alphanumid \to SP\})} \{alphanumid \notin dom(B)$$

This rule, together with the previous one, assures the propagation of the hidden symbols into the base.

$$\frac{B \vdash sigmorph \Rightarrow \rho}{B \vdash alphanumid = sigmorph \Rightarrow (\emptyset, \{alphanumid \to \rho\})} \{alphanumid \notin dom(B)$$

$$\frac{B \vdash specabstr \Rightarrow SA}{B \vdash alphanumid = specabstr \Rightarrow (\emptyset, \{alphanumid \to SA\})} \{alphanuid \notin dom(B)$$

Signature Morphisms

⟨sigmorph⟩	::=	[⟨rename-list⟩]
⟨rename-list⟩	::=	⟨rename⟩ [, ⟨rename-list⟩]
⟨rename⟩	::=	⟨inf-id⟩ **to** ⟨inf-id⟩ | ⟨id⟩ **to** ⟨id⟩ | ⟨sid⟩ **to** ⟨sid⟩
⟨morph⟩	::=	⟨alphanumid⟩ | ⟨sigmorph⟩

$$\frac{B \vdash \mathit{rename\text{-}list} \Rightarrow \rho}{B \vdash [\,\mathit{rename\text{-}list}\,] \Rightarrow \rho}$$

$$\frac{B \vdash \mathit{rename} \Rightarrow \rho_1 \quad [B \vdash \mathit{rename\text{-}list} \Rightarrow \rho_2]}{B \vdash \mathit{rename}\ [,\mathit{rename\text{-}list}] \Rightarrow \rho_1\ [\ \cup\ \rho_2]}\ \{[\mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset]$$

Abstractly, a signature morphism is a finite map. This motivates the side condition.

$$\frac{}{B \vdash \mathit{inf\text{-}id}_1\ \textbf{to}\ \mathit{inf\text{-}id}_2 \Rightarrow \{\mathit{inf\text{-}id}_1 \to \mathit{inf\text{-}id}_2\}}$$

$$\frac{}{B \vdash \mathit{id}_1\ \textbf{to}\ \mathit{id}_2 \Rightarrow \{\mathit{id}_1 \to \mathit{id}_2\}}$$

$$\frac{}{B \vdash \mathit{sid}_1\ \textbf{to}\ \mathit{sid}_2 \Rightarrow \{\mathit{sid}_1 \to \mathit{sid}_2\}}$$

$$\frac{\mathit{alphanumid} \in \mathrm{dom}\,(B)}{B \vdash \mathit{alphanumid} \Rightarrow B(\mathit{alphanumid})}$$

The name of a system part is translated to its definition.

Plus

⟨specexp⟩	::=	⟨aspecexp⟩ [+ ⟨specexp⟩]
⟨aspecexp⟩	::=	⟨alphanumid⟩ | ⟨specbody⟩

$$\frac{B \vdash \mathit{aspecexp} \Rightarrow SP_1 \quad [B \vdash \mathit{specexp} \Rightarrow SP_2]}{B \vdash \mathit{aspecexp}\ [\ +\ \mathit{specexp}] \Rightarrow SP_1\ [\ \cup\ SP_2]}\ \{SP_1[\cup SP_2]\ \text{is well formed}$$

The sum of two flat specifications is again a flat specification with signature and axioms the union of the signatures and axioms of the component specifications. Clearly, this union has to be well formed; otherwise it is rejected.

For *alphanumid* one uses the rule given in the previous section.

Rename

$$\langle\text{specexp}\rangle \quad ::= \quad \textbf{rename}\ \langle\text{specexp}\rangle\ \textbf{by}\ \langle\text{morph}\rangle$$

Abstractly, a signature morphism ρ is a finite map from identifiers to identifiers. If $id \notin dom(\rho)$ we consider that $\rho(id) = id$. A finite map ρ is extended to expressions, sort expressions and environments in a trivial way which we do not further describe here. When a signature morphism ρ is applied to a specification SP, it is possible that $dom(\rho)$ contains identifiers outside the domain $dom\ \Sigma(SP)$ of signature identifiers. To avoid both the collision of these identifiers with variables bound in SP and the renaming of identifiers occurring in the standard signature[3] Σ_S we restrict ρ to $\Sigma \setminus \Sigma_S$ before applying it to SP[4]. We ambiguously write this as $\rho|_\Sigma$.

$$\frac{C \vdash specexp \Rightarrow (\Sigma, Ax) \quad C \vdash morph \Rightarrow \rho}{C \vdash\ \textbf{rename}\ specexp\ \textbf{by}\ morph \Rightarrow \rho|_\Sigma(\Sigma, Ax)}\ \{\rho|_\Sigma(\Sigma, Ax)\ \text{is well formed}$$

The renamed specification has to be well formed. This is a very strong condition which excludes all erroneous renamings which occur when changing the attributes of an identifier (e.g. a sort identifier is renamed to an identifier which was declared as a function in the same specification or a function identifier is renamed to a function which occurs in the same specification with another signature).

Hide

$$\langle\text{aspecexp}\rangle \quad ::= \quad \textbf{hide}\ \langle\text{sigel-set}\rangle\ \textbf{in}\ \langle\text{aspecexp}\rangle$$

$$\langle\text{sigel-set}\rangle \quad ::= \quad \langle\text{sigel}\rangle\ [\textbf{,}\ \langle\text{sigel-set}\rangle]$$

$$\langle\text{sigel}\rangle \quad ::= \quad \langle\text{opn}\rangle \mid \langle\text{sid}\rangle \mid \textbf{SIG}\ \textbf{(}\ \langle\text{specexp}\rangle\ \textbf{)}$$

As we already mentioned an identifier is hidden by renaming it to a fresh identifier from $HiddenId$. This fresh renaming is abstractly realized with an injective function

$$(.)_H : VisibleId \rightarrow HiddenId$$

where $H \in HiddenIdSet$ and such that

$$id \in VisibleId \quad \Rightarrow \quad id_H \in (HiddenId \setminus H).$$

Given a set of visible identifiers VIS and a set of hidden identifiers HIS, we define their associated renaming morphism ρ^{VIS}_{HIS} as follows:

$$\rho^{VIS}_{HIS} = \{id \rightarrow id_{HIS} | id \in VIS\}$$

The visible identifiers of a signature have to build again a signature. As a consequence, hiding a class has to automatically hide all sorts, sort synonyms, maps and functions whose signatures contain this class. Similarly, when hiding a sort constructor and a sort synonym. Given an identifier id and a signature Σ we denote by $scons(id, \Sigma)$, $ssyns(id, \Sigma)$, $maps(id, \Sigma)$ and $fncs(id, \Sigma)$ the set of sort constructors, the set of sort synonyms, the set of maps and respectively the set of functions which contain id in their signatures.

[3]See [BFG+93a, BFG+93b] for the definition of the standard signature.

[4]Remember, that the set of bound identifiers in a specification SP is disjoint from the set of identifiers declared in the signature or SP.

If $\Sigma = (CE, SCE, SSE, ME, FE)$ then the above sets are defined as follows:

$$scons, ssyns, fncs, maps : VisibleId \times Sign \rightarrow Fin(VisibleId)$$

$$\begin{array}{lcl} scons(id,\Sigma) & = & \{scid \mid id\ occurs\ in\ SCE(scid)\} \\ ssyn(id,\Sigma) & = & \{ssid \mid id\ occurs\ in\ SSE(ssid)\} \\ fncs(id,\Sigma) & = & \{fid \mid id\ occurs\ in\ FE(fid)\} \\ maps(id,\Sigma) & = & \{mid \mid id\ occurs\ in\ ME(mid)\} \end{array}$$

The set of all identifiers depending on id — the downward closure of id — is written as $d(id, \Sigma)$. It is defined as follows:

$$\begin{array}{lll} id \in IdSet & \Rightarrow d(id,\Sigma) = & \{id\} \cup fncs(id,\Sigma) \cup maps(id,\Sigma) \cup d(scons(id,\Sigma),\Sigma) \\ id \in SconSet & \Rightarrow d(id,\Sigma) = & \{id\} \cup fncs(id,\Sigma) \cup maps(id,\Sigma) \cup d(ssyns(id,\Sigma),\Sigma) \\ id \in dom(SSE) & \Rightarrow d(id,\Sigma) = & \{id\} \cup fncs(id,\Sigma) \cup maps(id,\Sigma) \cup d(ssyns(id,\Sigma),\Sigma) \\ id \in dom(FE) & \Rightarrow d(id,\Sigma) = & \{id\} \\ id \in dom(ME) & \Rightarrow d(id,\Sigma) = & \{id\} \end{array}$$

This function is extended point-wise to sets of identifiers. We also forbid to hide identifiers from the standard signature Σ_S.

Note: The relation $\leq$ between classes can be considered as an axiom. So hiding a class automatically hides this axiom but it *does not affect* the other classes. Similarly hiding a function does not affect the other functions which use the hidden one in their definition.

$$\frac{HIS, C \vdash \textit{sigel-set} \Rightarrow VIS \quad HIS, C \vdash \textit{aspecexp} \Rightarrow (\Sigma, Ax)}{HIS, C \vdash\ \textbf{hide}\ \textit{sigel-set}\ \textbf{in}\ \textit{aspecexp} \Rightarrow \rho_{HIS}^{d(VIS,\Sigma)}(\Sigma, Ax)}\ \{d(VIS,\Sigma) \subseteq dom(\Sigma \setminus \Sigma_S)$$

$$\frac{B \vdash \textit{sigel} \Rightarrow VIS_1 \quad [B \vdash \textit{sigel-set} \Rightarrow VIS_2]}{B \vdash \textit{sigel}[,\ \textit{sigel-set}] \Rightarrow VIS_1[\cup VIS_2]}$$

$$\frac{}{B \vdash \textit{opn} \Rightarrow \{\textit{opn}\}}$$

$$\frac{}{B \vdash \textit{sid} \Rightarrow \{\textit{sid}\}}$$

$$\frac{B \vdash \textit{specexp} \Rightarrow (\Sigma, Ax)}{B \vdash\ \textbf{SIG}(\textit{specexp}) \Rightarrow dom(\Sigma)}$$

Export

⟨aspecexp⟩ ::= **export** ⟨sigel-set⟩ **in** ⟨aspecexp⟩

Export is similar to hiding. However, in this case the visible identifiers are given in the export list. As before, we must close this list, to a valid, visible signature. This closure operation is written as $u(id, \Sigma)$ — $\underline{u}$pward closure of id with respect to Σ — and it is defined as below.

Export–Closure Building

$$\begin{array}{rl}
id \in IdSet & \Rightarrow u(id,\Sigma) = \{id\} \\
id \in SConSet & \Rightarrow u(id,\Sigma) = \{id\} \cup flat(CE(id)) \\
id \in dom(SSE) & \Rightarrow u(id,\Sigma) = \{id\} \cup u(flat(SSE(id)),\Sigma) \\
(id,t) \in FE & \Rightarrow u(id,\Sigma) = \{id\} \cup u(flat(t),\Sigma) \\
(id,t) \in ME & \Rightarrow u(id,\Sigma) = \{id\} \cup u(flat(t),\Sigma)
\end{array}$$

Given an expression t, then $flat(t)$ builds the set containing all the class identifiers, sort constructors and sort synonyms which occur in t. Sort variables are ignored. Denote by $dom(\Sigma)$ the set of all identifiers occurring in the signature. Then we define the complement set of $u(VIS,\Sigma)$ as follows:

$$c(VIS,\Sigma) = dom(\Sigma) \setminus (dom(\Sigma_S) \cup u(VIS,\Sigma))$$

The semantic rule for export is then as follows:

$$\frac{HIS, C \vdash \textit{sigel-set} \Rightarrow VIS \quad HIS, C \vdash \textit{aspecexp} \Rightarrow (\Sigma, Ax)}{HIS, C \vdash \ \textbf{export}\ \textit{sigel-set}\ \textbf{in}\ \textit{aspecexp} \Rightarrow \rho_{HIS}^{c(VIS,\Sigma)}(\Sigma, Ax)} \ \{VIS \subseteq dom(\Sigma \setminus \Sigma_S)$$

Parameterized Specifications

⟨abstr⟩	::=	⟨alphanumid⟩ | ⟨specabstr⟩
⟨specabstr⟩	::=	**param** ⟨spec-list⟩ **body** ⟨aspecexp⟩
⟨spec-list⟩	::=	⟨alphanumid⟩ = ⟨specexp⟩ [**,** ⟨spec-list⟩]
⟨specexp⟩	::=	⟨abstr⟩ **(** ⟨arg-lst⟩ **)**
⟨arg-lst⟩	::=	⟨specexp⟩ [**via** ⟨morph⟩] [**,** ⟨arg-lst⟩]

A parameterized specification (SpP, *specexp*) can be understood as a "macro definition" where

$$SpP = \{id_1 \to Sp_1\} : \ldots : \{id_n \to Sp_n\}$$

is the list of evaluated formal parameters and *specexp* is the unevaluated body.

This specification can be applied to a list of actual parameters

$$(\rho_1, SP'_1) : \ldots : (\rho_n, SP'_n)$$

only if:

- the signatures of the actual parameter include (modulo renaming) the signatures of the formal parameters i.e. if $(\rho_i|_{\Sigma_i})(\Sigma_i) \subseteq \Sigma'_i$
- the union $\rho = (\rho_1|_{\Sigma_1}) \cup \ldots \cup (\rho_n|_{\Sigma_n})$ is again a signature morphism

In that case, the renamed body $\rho(\textit{specexp})$ is evaluated in a context where the actual parameters replace the formal ones.

$$\frac{B \vdash \textit{spec-list} \Rightarrow SpP \quad B + SpP \vdash \textit{aspecexp} \Rightarrow SP}{B \vdash \ \textbf{param}\ \textit{spec-list}\ \textbf{body}\ \textit{aspecexp} \Rightarrow (SpP, \textit{aspecexp})}$$

$$\frac{B \vdash \textit{specexp} \Rightarrow SP \quad [B \vdash \textit{spec-list} \Rightarrow SpP]}{B \vdash \textit{alphanumid} = \textit{specexp}[\,,\ \textit{spec-list}] \Rightarrow \{\textit{alphanumid} \to SP\}[:\ SpP]} \ \left\{ \begin{array}{l} [\textit{alphanumid} \notin \\ dom(SpP)] \end{array} \right.$$

Note: The environment SpP overrides B. We therefore use $B + SpP$ instead of $B \cup SpP$.

$$\frac{\begin{array}{l} B \vdash abstr \Rightarrow \{id_1 \rightarrow SP_1 : \ldots : id_n \rightarrow SP_n, aspecexp) \\ B \vdash arg\text{-}lst \Rightarrow (\rho_1, SP_1') : \ldots : (\rho_n, SP_n') \\ B + \{id_1 \rightarrow SP_1' : \ldots : id_n \rightarrow SP_n'\} \\ \vdash (\rho_1|_{\Sigma_1} \cup \ldots \cup \rho_n|_{\Sigma_n})(aspecexp) \Rightarrow SP \end{array}}{B \vdash abstr(arg\text{-}lst) \Rightarrow SP} \left\{ \begin{array}{l} (\rho_i|_{\Sigma_i})(\Sigma_i) \subseteq \Sigma_i' \\ id \in dom(\rho_i) \cap dom(\rho_j) \\ \quad \Rightarrow \rho_i(id) = \rho_j(id) \end{array} \right.$$

$$\frac{B \vdash specexp \Rightarrow SP \quad [B \vdash morph \Rightarrow \rho] \quad [[B \vdash arglst \Rightarrow SpA]]}{B \vdash specexp[\ \mathbf{via}\ morph][[\ ,\ arglst]] \Rightarrow (\{\}[+\rho], SP)[[: SpA]]}$$

Acknowledgments

The authors would like to thank all colleagues that were concerned with the development of Spectrum. In particular we thank Rudi Hettler, Franz Regensburger and Oscar Slotosch for stimulating discussions and for their comments to this report. Special thanks go to Manfred Broy for supervising the whole development of Spectrum.

Bibliography

[BFG$^+$93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.

[BFG$^+$93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.

[GR94] Radu Grosu and Franz Regensburger. The Logical Framework of SPECTRUM. Technical Report TUM-I9402, Institut für Informatik, Technische-Universität München, 1994.

Appendix A

Context Free Syntax

A.1 In the Small

In this section we define the raw structure of our language "in the small" by giving a context-free grammar in EBNF style.

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

Signatures
⟨signature⟩   ::=  class ⟨id⟩ [subclass of {⟨id⟩ // ,}+]
                |  ⟨id⟩ subclass of {⟨id⟩ // ,}+
                |  sort ⟨sid⟩ { ⟨sid⟩* | :: ⟨classexp⟩ }
                |  {⟨sid⟩ // ,}+ :: ⟨classexp⟩
                |  sortsyn {⟨sid⟩}+ = ⟨sortexp⟩
                |  SIG ( ⟨specexp⟩ )
                |  {⟨id⟩ // ,}+ : [⟨context⟩] ⟨sortexp⟩ to ⟨sortexp⟩
                |  {⟨inf-id⟩ // ,}+ : [⟨context⟩] ⟨sortexp⟩ × ⟨sortexp⟩ to ⟨sortexp⟩ [⟨prio⟩]
                |  {⟨id⟩ // ,}+ : [⟨context⟩] ⟨sortexp⟩
                |  {⟨inf-id⟩ // ,}+ : [⟨context⟩] ⟨sortexp⟩ × ⟨sortexp⟩ → ⟨sortexp⟩ [⟨prio⟩]
⟨classexp⟩    ::=  [( {⟨id⟩ // ,}+ )] ⟨id⟩
⟨prio⟩        ::=  prio ⟨num⟩ [: {left|right}]

Sort Expressions
⟨sortexp⟩     ::=  ⟨sortexp1⟩
                |  ⟨sortexp1⟩ → ⟨sortexp⟩                    (Functional Sort)
⟨sortexp1⟩    ::=  ⟨sortexp2⟩
                |  ⟨sortexp2⟩ {× ⟨sortexp2⟩}+                   (Product Sort)
⟨sortexp2⟩    ::=  ⟨asort⟩ | ⟨sid⟩ {⟨asort⟩}+
⟨asort⟩       ::=  ⟨sid⟩ | ( ⟨sortexp⟩ )

Sort Contexts
⟨context⟩     ::=  {⟨scontext⟩ // ,}+ ⇒
```

⟨scontext⟩	::=	{⟨sid⟩ // ,}$^+$ **::** ⟨id⟩

Axioms

⟨axioms⟩	::=	**axioms** [⟨varlist⟩] {[**{** ⟨id⟩ **}**] ⟨exp1⟩ **;**}* **endaxioms**
	|	⟨simplesorts⟩ **generated by** ⟨opns⟩
⟨varlist⟩	::=	[⟨context⟩] {{$\forall$|$\forall^{\perp}$} ⟨opdecls⟩}$^+$ **in**
⟨opdecls⟩	::=	{⟨sopdecl⟩ // ,}$^+$ | {⟨id⟩ // ,}$^+$
⟨sopdecl⟩	::=	{⟨id⟩ // ,}$^+$ **:** ⟨sortexp⟩
⟨simplesorts⟩	::=	{⟨sid⟩ {⟨sid⟩}* // ,}$^+$

Expressions

⟨exp1⟩	::=	⟨exp2⟩
	|	{ $\forall$ | $\forall^{\perp}$ | $\exists$ | $\exists^{\perp}$ } ⟨opdecl⟩ **.** ⟨exp1⟩
	|	$\boldsymbol{\lambda}$ ⟨pat⟩ **.** ⟨exp1⟩
⟨exp2⟩	::=	⟨exp3⟩ | ⟨exp2⟩ ⟨id⟩ [**:** ⟨asort⟩] ⟨exp1⟩
⟨exp3⟩	::=	⟨aexp⟩ | ⟨exp3⟩ ⟨aexp⟩
⟨aexp⟩	::=	⟨opn⟩ | **(** ⟨exp1⟩ **)** | **(** ⟨exp1⟩ {**,** ⟨exp1⟩}$^+$ **)** | ⟨aexp⟩ **:** ⟨asort⟩
⟨pat⟩	::=	⟨id⟩ [**:** ⟨sortexp⟩]
	|	**(** ⟨id⟩ [**:** ⟨sortexp⟩] {**,** ⟨id⟩ [**:** ⟨sortexp⟩]}$^+$ **)**

Identifiers

⟨id⟩	::=	⟨alphanumid⟩ | ⟨symbolid⟩ | ⟨num⟩ | ⟨charconst⟩ | ⟨string⟩
⟨inf-id⟩	::=	⟨inf-alphanumid⟩ | ⟨inf-symbolid⟩
⟨sid⟩	::=	⟨id⟩$_{\{\rightarrow, \times, -\,>, *\}}$
⟨opn⟩	::=	⟨id⟩ | ⟨inf-id⟩
⟨opns⟩	::=	{⟨opn⟩ // ,}$^+$

A.2 In the Large

⟨system⟩	::=	⟨syspart⟩ [system]
⟨syspart⟩	::=	⟨alphanumid⟩ = { ⟨specexp⟩ | ⟨sigmorph⟩ | ⟨specabstr⟩ }

Signature Morphisms

⟨sigmorph⟩	::=	[⟨rename-list⟩]
⟨rename-list⟩	::=	⟨rename⟩ [, ⟨rename-list⟩]
⟨rename⟩	::=	⟨inf-id⟩ **to** ⟨inf-id⟩ | ⟨id⟩ **to** ⟨id⟩ | ⟨sid⟩ **to** ⟨sid⟩
⟨morph⟩	::=	⟨alphanumid⟩ | ⟨sigmorph⟩

Structured Specifications

⟨specexp⟩	::=	⟨aspecexp⟩ [+ ⟨specexp⟩]
	|	**abstr (** ⟨arg-lst⟩ **)**
⟨arg-lst⟩	::=	⟨specexp⟩ [**via** ⟨morph⟩] [, ⟨arg-lst⟩]
⟨aspecexp⟩	::=	**rename** ⟨specexp⟩ **by** ⟨morph⟩
	|	**(** ⟨specexp⟩ **)**
	|	**hide** ⟨sigel-set⟩ **in** ⟨aspecexp⟩
	|	**export** ⟨sigel-set⟩ **in** ⟨aspecexp⟩

	|	**{ enriches** ⟨specexp⟩; ⟨decls⟩ **}**
	|	⟨aspecexp⟩ | ⟨specbody⟩
⟨sigel-set⟩	::=	⟨sigel⟩ [, ⟨sigel-set⟩]
⟨sigel⟩	::=	⟨opn⟩ | ⟨sid⟩ | **SIG (** ⟨specexp⟩ **)**

Parameterized Specifications

⟨abstr⟩	::=	⟨alphanumid⟩ | ⟨specabstr⟩
⟨specabstr⟩	::=	**param** ⟨spec-list⟩**body** ⟨aspecexp⟩
⟨spec-list⟩	::=	⟨alphanumid⟩ = ⟨specexp⟩ [, ⟨spec-list⟩]