CASL

The Common Algebraic Specification Language

Summary

CoFI Document: CASL/Summary
Version: 1.0.1 25 March 2001

by The CoFI Task Group on Language Design
E-mail address for comments: cofi-language@brics.dk
CoFI: The Common Framework Initiative
http://www.brics.dk/Projects/CoFI

A list of the changes[1] *relative to the previous version of this document is available.*

This document is available in various formats from the CoFI *archives.*[2]

[1] `http://www.brics.dk/Projects/CoFI/Documents/CASL/v1.0.1/Summary-Changes/index.html`

[2] `ftp://ftp.brics.dk/Projects/CoFI/Documents/CASL/Summary/`

Abstract

The language CASL is central to CoFI, the Common Framework Initiative for algebraic specification and development. It is a reasonably expressive algebraic language for specifying requirements and design for conventional software. From CASL, simpler languages (e.g., for interfacing with existing tools) are to be obtained by restriction, and CASL is to be incorporated in more advanced languages (e.g., higher-order). CASL strikes a balance between simplicity and expressiveness. The main features of its design are as follows:

Many-sorted *basic specifications* in CASL denote classes of many-sorted partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed. Axioms are first-order formulae built from equations and definedness assertions. Sort generation constraints can be stated. Datatype declarations are provided for concise specification of sorts together with some constructors and (optional) selectors. Subsorted basic specifications provide moreover a simple treatment of subsorts, interpreting subsort inclusion as embedding.

Structured specifications allow translation, reduction, union, and extension of specifications. Extensions may be required to be free; initiality constraints are a special case. A simple form of generic specifications is provided, together with instantiation involving parameter-fitting translations and views.

Architectural specifications express that the specified software is to be composed from separately-developed, reusable units with clear interfaces.

Finally, *specification libraries* allow the (distributed) storage and retrieval of named specifications.

This document gives a detailed summary of the syntax and intended semantics of CASL. It is intended for readers who are already familiar with the main concepts of algebraic specifications.

Contents

About this document

This document gives a detailed summary of the syntax and intended semantics of CASL. It is intended for readers who are already familiar with the main concepts of algebraic specification. In general, it does not attempt to motivate the design choices that have been taken; a rationale for the design has been published separately [Mos97], as has a full exposition of architectural specifications [BST00, BST98]. The formal semantics of CASL is available [CoF99].

Structure

Part I (Chapters 1, 2, 3, 4) deals with *basic specifications*—first many-sorted, then subsorted.

Part II (Chapters 5, 6) provides *structured specifications*, together with *specification definitions*, *instantiations*, and *views*.

Part III (Chapters 7, 8) summarizes so-called *architectural and unit specifications*, which, in contrast to structured specifications, prescribe the separate development of composable, reusable implementation units.

Finally, Part IV (Chapters 9, 10) considers *specification libraries*.

In each part, a chapter summarizing the main *semantic concepts* underlying the kind of specification concerned is followed by a chapter presenting the (concrete and abstract) syntax of the associated CASL *language constructs* and indicating their intended semantics.

The Index may facilitate locating the places in this document where terminology is explained.

Appendix A provides a complete grammar for the *abstract syntax* of the language, collecting the fragments that are given in the semantics summary. Appendix B provides an *abbreviated grammar* (for the *same* abstract syntax).

Appendix C provides a complete grammar for the *concrete syntax* of the language, determining how CASL specifications are to be input. (The relationship between concrete and abstract syntax is mostly rather straightforward, and left implicit here.) Appendix D summarizes the intended *display format* for CASL, showing how CASL specifications appear when displayed after parsing.

Appendix E illustrates the syntax of CASL by giving some simple examples. (A systematic library of useful specifications is available separately [RM00].)

Versions

Version 0.95 of this document was the summary of the complete CASL Tentative Design [CoF96], available since December 1996.

CoFI Note S-1 [CoF97c] extended the cited Tentative Design Language Summary with annotations concerning some questions and doubts raised by the Semantics task group, in connection with their development of a formal semantics for CASL, see CoFI Note S-4 [CoF97d]; CoFI Note S-1 has since been updated with an indication of how the issues were expected to be resolved.

Version 0.96 was the first draft of the Summary of the CASL Proposed Design, resolving almost all the issues that had been raised concerning the Tentative Design.

Version 0.97 [CoF97b] incorporated some relatively minor enhancements to the Proposed Design, suggested shortly after version 0.96 became available (April 1997). CoFI Note S-6 [CoF97e] provided a completed draft semantics for this version. Version 0.97 was reviewed by IFIP WG1.3 [IFI97], resulting in tentative approval of the design.

Version 0.98 showed just which bits of CASL were subject to reconsideration, in view of the referees' comments [CoF97a] and the recommendations made by the CoFI Semantics Task Group [CoF97e].

Version 0.99 summarized what was almost the final CASL Design, now also incorporating concrete syntax (the syntax of views and architectural specifications was, however, still somewhat tentative).

CoFI Note M-4 [BST00] provided an updated rationale for the design of architectural specifications in CASL.

The Summary of CASL version 1.0 reflected various minor adjustments to the details of CASL, arising mainly from the work on the formal semantics of CASL and on the implementation of parsers. It also incorporated a significant revision of the treatment of *views*. The original release was

in October 1998; a subsequent release in July 1999 incorporated numerous minor clarifications and corrections, and was accompanied by the complete formal semantics of CASL v1.0 [CoF99] (also available as Semantics Note S-9).

The present Summary of CASL version 1.0.1 reflects some changes that have been made concerning the concrete syntax of CASL v1.0 (facilitating the use of bulleted lists of axioms and local variable declarations, and affecting the form and position of comments and annotations). The abstract syntax and semantics of CASL v1.0.1 are the same as for CASL v1.0, apart from a relaxation concerning operations declared both as total and partial, and an adjustment of the semantics of architectural specifications (concerning whether sharing analysis affects well-formedness). Further clarifications have been made to the wording of Summary. A list all of the changes[3] relative to the Summary of CASL v1.0 is available.

The preliminary response by the Language Design task group to the IFIP WG1.3 referees [CoF97a] has been expanded to clarify the extent to which the referees' recommendations have been followed [CoF00].

Contributors

The CoFI Language Design Task Group was formed at the founding meeting of the Common Framework Initiative, in Oslo, September 1995. The working meetings held in Paris (November 1995), Munich (January 1996), Oxford (March 1996), Paris (May 1996), Munich (July 1996), Edinburgh (November 1996), Paris (January and April 1997), Amsterdam (September 1997), Bremen (January 1998), and finally Lisbon (April 1998) helped to guide the subsequent design of CASL. The following persons have participated in some or all of these meetings: Egidio Astesiano, Hubert Baumeister, Jan Bergstra, Gilles Bernot, Didier Bert, Mohammed Bettaz, Michel Bidoit, Mark van den Brand, Maria Victoria Cengarle, Maura Cerioli, Christine Choppy, Ole-Johan Dahl, Hans-Dieter Ehrich, Hartmut Ehrig, José Fiadeiro, Marie-Claude Gaudel, Chris George, Joseph Goguen, Radu Grosu, Magne Haveraaen, Anne Haxthausen, Jim Horning, Hélène Kirchner, Kolyang, Hans-Jörg Kreowski, Bernd Krieg-Brückner, Pierre Lescanne, Christoph Lüth, Tom Maibaum, Grant Malcolm, Karl Meinke, Till Mossakowski, Peter D. Mosses, Peter Padawitz, Fernando Orejas, Olaf Owe, Gianna Reggio, Horst Reichel, Don Sannella, Giuseppe Scollo, Amilcar Sernadas, Andrzej Tarlecki, Eelco Visser, Frédéric Voisin, Eric Wagner, Michał Walicki, and Martin Wirsing.

The acronym CASL for the Common Algebraic Specification Language was

[3] `http://www.brics.dk/Projects/CoFI/Documents/CASL/v1.0.1/Summary-Changes/index.html`

originally proposed by Christine Choppy.

This document has been developed by Peter D. Mosses, originally on the basis of the design proposals and notes made available before the Munich meeting in July 1996 and the agreements reached during that meeting. Its preparation has been greatly assisted by the timely production of the minutes of several meetings by Christine Choppy.

Subsequent versions of this document have attempted to incorporate the improvements suggested in various comments and notes from the following persons: Egidio Astesiano, Hubert Baumeister, Jan Bergstra, Gilles Bernot, Didier Bert, Michel Bidoit, Pietro Cenciarelli, Maria Victoria Cengarle, Maura Cerioli, Christine Choppy, Ole-Johan Dahl, Marie-Claude Gaudel, Chris George, Joseph Goguen, Radu Grosu, Anne Haxthausen, Jim Horning, Hélène Kirchner, Kolyang, Hans-Jörg Kreowski, Bernd Krieg-Brückner, Christoph Lüth, Till Mossakowski, Peter D. Mosses, Olaf Owe, Gianna Reggio, Markus Roggenbach, Erik Saaman, Don Sannella, Andrzej Tarlecki, Christophe Tronche, Eelco Visser, Frédéric Voisin, Michał Walicki, Bjarke Wedemeijer, Martin Wirsing, Uwe Wolter, and Alexandre Zamulin.

The design of the abstract syntax and semantics of CASL has been much influenced by the work of the CoFI Semantics task group on the formal semantics of CASL, which has been produced mainly by Hubert Baumeister, Maura Cerioli, Anne Haxthausen, Till Mossakowski, Don Sannella, and Andrzej Tarlecki.

The concrete syntax (input syntax and display format) of CASL has been designed initially by Michel Bidoit, Christine Choppy, Bernd Krieg-Brückner, and Frédéric Voisin, and coordinated by Peter D. Mosses. Feedback from the development of various prototype parsers for CASL by Hubert Baumeister, Mark van den Brand, Kolyang, Till Mossakowski, Markus Roggenbach, Axel Schairer, Christophe Tronche, Frédéric Voisin, and Bjarke Wedemeijer has also contributed significantly to the final concrete syntax design.

The coordinator of the Language Design task group is Bernd Krieg-Brückner.

Part I

Basic Specifications

Chapter 1

Basic Concepts

First, before considering the particular concepts underlying CASL, here is a brief reminder of how specification frameworks in general may be formalized in terms of so-called ***institutions*** [GB92] (some category-theoretic details are omitted) and *proof systems*.

A *basic specification framework* may be characterized by:

- a class **Sig** of ***signatures*** Σ, each determining the set of ***symbols*** $|\Sigma|$ whose intended interpretation is to be specified, with ***morphisms*** between signatures;
- a class $\mathbf{Mod}(\Sigma)$ of ***models***, with ***homomorphisms*** between them, for each signature Σ;
- a set $\mathbf{Sen}(\Sigma)$ of ***sentences*** (or ***axioms***), for each signature Σ;
- a relation $\models$ of ***satisfaction***, between models and sentences over the same signature; and
- a ***proof system***, for inferring sentences from sets of sentences.

A ***basic specification*** consists of a signature Σ together with a set of sentences from $\mathbf{Sen}(\Sigma)$. The signature provided for a particular declaration or sentence in a specification is called its ***local environment***. It may be a restriction of the entire signature of the specification, e.g., determined by an order of ***presentation*** for the signature declarations and the sentences with ***linear visibility***, where symbols may not be used before they have been declared; or it may be the entire signature, reflecting ***non-linear visibility***.

The (loose) ***semantics*** of a basic specification is the class of those models in $\mathbf{Mod}(\Sigma)$ which satisfy all the specified sentences. A specification is said to be ***consistent*** when there are some models that satisfy all the sentences, and

inconsistent when there are no such models. A sentence is a ***consequence*** of a basic specification if it is satisfied in all the models of the specification.

A ***signature morphism*** $\sigma : \Sigma \to \Sigma'$ determines a ***translation*** function $\mathbf{Sen}(\sigma)$ on sentences, mapping $\mathbf{Sen}(\Sigma)$ to $\mathbf{Sen}(\Sigma')$, and a ***reduct*** function $\mathbf{Mod}(\sigma)$ on models, mapping $\mathbf{Mod}(\Sigma')$ to $\mathbf{Mod}(\Sigma)$.[1] Satisfaction is required to be preserved by translation: for all $S \in \mathbf{Sen}(\Sigma), M' \in \mathbf{Mod}(\Sigma')$,

$$\mathbf{Mod}(\sigma)(M') \models S \iff M' \models \mathbf{Sen}(\sigma)(S).$$

The proof system is required to be sound, i.e., sentences inferred from a specification are always consequences; moreover, inference is to be preserved by translation.

Sentences of basic specifications may include ***constraints*** that restrict the class of models, e.g., to reachable ones.

The rest of this chapter considers many-sorted basic specifications of the CASL specification framework, and indicates the underlying signatures, models, and sentences.[2] Consideration of the extra features concerned with subsorts is deferred to Chapter 3.

The syntax of the language constructs used for expressing many-sorted basic specifications is described in Chapter 2; subsorting constructs are deferred to Chapter 4. The abstract syntax of any well-formed basic specification determines a signature and a set of sentences, the models of which provide the semantics of the basic specification.

1.1 Signatures

A ***many-sorted signature*** $\Sigma = (S, TF, PF, P)$ consists of:

- a set S of ***sorts***;
- sets $TF_{w,s}$, $PF_{w,s}$, of ***total function symbols***, respectively ***partial function symbols***, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each *function* ***profile*** (w, s) consisting of a sequence of ***argument sorts*** $w \in S^*$ and a ***result sort*** $s \in S$ (***constants*** are treated as functions with no arguments);
- sets P_w of *predicate* symbols, for each *predicate profile* consisting of a sequence of argument sorts $w \in S^*$.

[1] In fact **Sig** is a category, and **Sen**(.) and **Mod**(.) are functors. The categorial aspects of the semantics of CASL are emphasized in its formal semantics [CoF97e].

[2] The choice of a particular proof system for CASL has been investigated, but not yet decided.

Constants and functions are also referred to as ***operations***, following the traditions of algebraic specification.

Note that symbols used to identify sorts, operations, and predicates may be ***overloaded***, occurring in more than one of the above sets. To ensure that there is no ambiguity in sentences at this level, however, function symbols f and predicate symbols p are always ***qualified*** by profiles when used, written $f_{w,s}$ and p_w respectively. (The language considered in Chapter 2 allows the omission of such qualifications when these are unambiguously determined by the context.)

A ***many-sorted signature morphism*** $\sigma : (S, TF, PF, P) \to (S', TF', PF', P')$ consists of a mapping from S to S', and for each $w \in S^*, s \in S$, a mapping between the corresponding sets of function, resp. predicate symbols. A partial function symbol may be mapped also to a total function symbol, but not vice versa.

1.2 Models

For a many-sorted signature $\Sigma = (S, TF, PF, P)$ a ***many-sorted model*** $M \in \mathbf{Mod}(\Sigma)$ is a ***many-sorted first-order structure*** consisting of a ***many-sorted partial algebra***:

- a non-empty ***carrier set*** s^M for each sort $s \in S$ (let w^M denote the Cartesian product $s_1^M \times \cdots \times s_n^M$ when $w = s_1 \ldots s_n$),
- a ***partial function*** f^M from w^M to s^M for each function symbol $f \in TF_{w,s}$ or $f \in PF_{w,s}$, the function being required to be total in the former case,

together with:

- a ***predicate*** $p^M \subseteq w^M$ for each predicate symbol $p \in P_w$.

A (weak) ***many-sorted homomorphism*** h from M_1 to M_2, with $M_1, M_2 \in \mathbf{Mod}(S, TF, PF, P)$, consists of a function $h_s : s^{M_1} \to s^{M_2}$ for each $s \in S$ preserving not only the values of functions but also their definedness, and preserving the truth of predicates.

Any signature morphism $\sigma : \Sigma \to \Sigma'$ determines the ***many-sorted reduct*** of each model $M' \in \mathbf{Mod}(\Sigma')$ to a model $M \in \mathbf{Mod}(\Sigma)$, defined by interpreting symbols of Σ in M in the same way that their images under σ are interpreted in M'.

1.3 Sentences

The ***many-sorted terms*** on a signature $\Sigma = (S, TF, PF, P)$ and a set of sorted, non-overloaded variables X are built from:

- variables from X;
- applications of qualified function symbols in $TF \cup PF$ to argument terms of appropriate sorts.

We refer to such terms as ***fully-qualified terms***, to avoid confusion with the terms of the language considered in Chapter 2, which allow the omission of qualifications and explicit sorts when these are unambiguously determined by the context.

For a many-sorted signature $\Sigma = (S, TF, PF, P)$ the ***many-sorted sentences*** in $\mathbf{Sen}(\Sigma)$ are the usual closed many-sorted first-order logic formulae, built from atomic formulae using quantification (over sorted variables) and logical connectives. An inner quantification over a variable makes a hole in the scope of an outer quantification over the same variable, regardless of the sorts of the variables. Implication may be taken as primitive (in the presence of an always-false formula), the other connectives being regarded as derived.

The ***atomic formulae*** are:

- applications of qualified predicate symbols $p \in P$ to argument terms of appropriate sorts;
- assertions about the definedness of fully-qualified terms;
- existential and strong equations between fully-qualified terms of the same sort.

Definedness assertions may be derived from existential equations using conjunction, or regarded as applications of fixed predicates. Strong equations may be derived from existential equations, using implication and conjunction; existential equations may be derived from conjunctions of strong equations and definedness assertions, or regarded as applications of fixed predicates.

The sentences $\mathbf{Sen}(\Sigma)$ also include ***sort-generation constraints***. Let $\Sigma = (S, TF, PF, P)$. A sort-generation constraint consists of (S', F') with $S' \subseteq S$ and $F' \subseteq TF \cup PF$. [3]

[3] The translation of such constraints along signature morphisms adds a further component, for technical reasons.

1.4 Satisfaction

The satisfaction of a sentence in a structure M is determined as usual by the holding of its atomic formulae w.r.t. assignments of (defined) values to all the variables that occur in them, the values assigned to variables of sort s being in s^M. The value of a term w.r.t. a variable assignment may be undefined, due to the application of a partial function during the evaluation of the term. Note, however, that the satisfaction of sentences is 2-valued (as is the holding of open formulae with respect to variable assignments).

The application of a predicate symbol p to a sequence of argument terms holds in M iff the values of all the terms are defined and give a tuple belonging to p^M. A definedness assertion concerning a term holds iff the value of the term is defined (thus it corresponds to the application of a constantly-true unary predicate to the term). An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined.

The value of an occurrence of a variable in a term is that provided by the given variable assignment. The value of the application of a function symbol f to a sequence of argument terms is defined only if the values of all the argument terms are defined and give a tuple in the domain of definedness of f^M, and then it is the associated result value.

A sort-generation constraint (S', F') is satisfied in a Σ-model M if the carriers of the sorts in S' are ***generated*** by the function symbols in F'. I.e., every element of each sort in S' is the value of a term built from just these symbols (possibly using variables of sorts not in S', with appropriate assignments of values to them).

Chapter 2

Basic Constructs

This chapter indicates the abstract and concrete syntax of the constructs of *many-sorted* basic specifications, and describes their intended interpretation.

For an introduction to the form of grammar used here to define the abstract syntax of language constructs, see Appendix A, which also provides the complete grammar defining the abstract syntax of the entire CASL specification language.

```
BASIC-SPEC ::= basic-spec BASIC-ITEMS*
```

A ***well-formed*** many-sorted basic specification `BASIC-SPEC` in the CASL language is written simply as a sequence of `BASIC-ITEMS` constructs:

$BI_1 \ldots BI_n$

The empty basic specification is not usually needed, but can be written '**{ }**'.

This language construct determines a basic specification within the underlying many-sorted institution, consisting of a signature and a set of sentences of the form described in Chapter 1. This signature and the class of models over it that satisfy the set of sentences provide the *semantics* of the basic specification. Thus this chapter explains well-formedness of basic specifications, and the way that they determine the underlying signatures and sentences, rather than directly explaining the intended interpretation of the constructs.

While *well-formedness* of specifications in the language can be checked statically, the question of whether the value of a term that occurs in a well-formed specification is necessarily defined in all models may depend on the specified axioms (and it is not decidable in general).

```
BASIC-ITEMS ::= SIG-ITEMS | FREE-DATATYPE | SORT-GEN
              | VAR-ITEMS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS
```

A `BASIC-ITEMS` construct is always a list, written:

plural-keyword $X_1; \ldots X_n;$

The *plural-keyword* may also be written in the singular (regardless of the number of items), and the final '`;`' may be omitted.

Each `BASIC-ITEMS` construct determines part of a signature and/or some sentences (except for `VAR-ITEMS`, which merely declares some global variables). The order of the basic items is generally significant: there is *linear visibility* of declared symbols and variables in a list of `BASIC-ITEMS` constructs (except within a list of datatype declarations). Verbatim repetition of the declaration of a symbol is allowed, and does not affect the semantics (some tools may however be able to locate and warn about such duplications, in case they were not intentional).

A list of signature declarations and definitions `SIG-ITEMS` determines part of a signature and possibly some sentences. A `FREE-DATATYPE` construct determines part of a signature together with some sentences. A sort-generation construct `SORT-GEN` determines part of a signature, together with some sentences including a corresponding sort generation constraint. A list of variable declaration items `VAR-ITEMS` determines sorted variables that are implicitly universally quantified in the subsequent axioms of the enclosing basic specification; note that variable declarations do not contribute to the signature of the specification in which they occur. A `LOCAL-VAR-AXIOMS` construct restricts the scope of the variable declarations to the indicated list of axioms. (Variables may also be declared locally in individual axioms, by explicit quantification.) An `AXIOM-ITEMS` construct determines a set of sentences.

2.1 Signature Declarations

```
SIG-ITEMS ::= SORT-ITEMS | OP-ITEMS | PRED-ITEMS
            | DATATYPE-ITEMS
```

A list `SORT-ITEMS` of sort declarations determines one or more sorts. A list `OP-ITEMS` of operation declarations and/or definitions determines one or more operation symbols, and possibly some sentences; similarly for a list `PRED-ITEMS` of predicate declarations and/or definitions. Operation and predicate symbols may be overloaded, being declared with several different profiles in the same local environment. A list `DATATYPE-ITEMS` of datatype declarations determines one or more sorts together with some constructor

and (optional) selector operations, and sentences defining the selector operations on the values given by the constructors with which they are associated.

2.1.1 Sorts

```
SORT-ITEMS ::= sort-items SORT-ITEM+
SORT-ITEM  ::= SORT-DECL
```

A list `SORT-ITEMS` of sort declarations is written:

sorts SI_1; ... SI_n;

2.1.1.1 Sort Declarations

```
SORT-DECL ::= sort-decl SORT+
SORT      ::= TOKEN-ID
```

A sort declaration `SORT-DECL` is written:

$s_1, \ldots, s_n$

It declares each of the sorts in the list s_1, ..., s_n.

2.1.2 Operations

```
OP-ITEMS ::= op-items OP-ITEM+
OP-ITEM  ::= OP-DECL | OP-DEFN
```

A list `OP-ITEMS` of operation declarations and definitions is written:

ops OI_1; ... OI_n;

2.1.2.1 Operation Declarations

```
OP-DECL ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
OP-NAME ::= ID
```

An operation declaration `OP-DECL` is written:

$f_1, \ldots, f_n : T, A_1, \ldots, A_m$

When the list A_1, ..., A_m is empty, the declaration is written simply:

$f_1, \ldots, f_n : T$

It declares each operation name $f_1, \ldots, f_n$ as a total or partial operation, with profile as specified by the operation type T, and as having the attributes $A_1, \ldots, A_m$ (if any). If an operation is declared both as total and as partial with the same profile, the resulting signature only contains the total operation.

Operation Types

```
OP-TYPE         ::= TOTAL-OP-TYPE | PARTIAL-OP-TYPE
TOTAL-OP-TYPE   ::= total-op-type SORT-LIST SORT
PARTIAL-OP-TYPE ::= partial-op-type SORT-LIST SORT
SORT-LIST       ::= sort-list SORT*
```

A total operation type `TOTAL-OP-TYPE` with some argument sorts is written:

$s_1 \times \ldots \times s_n \rightarrow s$

When the list of argument sorts is empty, the type is simply written 's'. The sign displayed as '$\times$' may be input as '$\times$' in ISO Latin-1, or as '`*`' in ASCII. The sign displayed as '$\rightarrow$' is input as '`->`'.

A partial operation type `PARTIAL-OP-TYPE` with some argument sorts is written:

$s_1 \times \ldots \times s_n \rightarrow ?\ s$

When the list of argument sorts is empty, the type is simply written '$?\ s$'.

The operation profile determined by the type has argument sorts $s_1, \ldots, s_n$ and result sort s.

Operation Attributes

```
OP-ATTR        ::= BINARY-OP-ATTR | UNIT-OP-ATTR
BINARY-OP-ATTR ::= assoc-op-attr | comm-op-attr | idem-op-attr
UNIT-OP-ATTR   ::= unit-op-attr TERM
```

Operation attributes assert that the operations being declared (which must be binary) have certain common properties, which are characterized by strong equations, universally quantified over variables of the appropriate sort. (This can also be used to add attributes to operations that have previously been declared without them.)

The attribute `assoc-op-attr` is written '*assoc*'. It asserts the ***associativity*** of an operation f:

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

The attribute of associativity moreover implies a parsing annotation that allows an infix operation f of the form '$\_\_t\_\_$' (or '$\_\_\ \_\_$') to be iterated without explicit grouping parentheses.

The attribute `comm-op-attr` is written '*comm*'. It asserts the ***commutativity*** of an operation f:

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

The attribute `idem-op-attr` is written '*idem*'. It asserts the ***idempotency*** of an operation f:

$$f(x, x) = x$$

The attribute `UNIT-OP-ATTR` is written '*unit* T'. It asserts that the value of the term T is the ***unit (left and right)*** of an operation f:

$$f(T, x) = x \wedge f(x, T) = x$$

In practice, the unit T is normally a constant. In any case, T must not contain any variables.

The declaration enclosing an operation attribute is ill-formed unless the operation profile has exactly two argument sorts, both the same as the result sort.

2.1.2.2 Operation Definitions

```
OP-DEFN          ::= op-defn OP-NAME OP-HEAD TERM
OP-HEAD          ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD    ::= total-op-head ARG-DECL* SORT
PARTIAL-OP-HEAD  ::= partial-op-head ARG-DECL* SORT
ARG-DECL         ::= arg-decl VAR+ SORT
VAR              ::= SIMPLE-ID
```

A definition `OP-DEFN` of a total operation with some arguments is written:

$$f(v_{11}, \ldots, v_{1m_1} : s_1;\ \ldots;\ v_{n1}, \ldots, v_{nm_n} : s_n) : s = T$$

When the list of arguments is empty, the definition is simply written:

$$f : s = T$$

A definition `OP-DEFN` of a partial operation with some arguments is written:

$$f(v_{11}, \ldots, v_{1m_1} : s_1;\ \ldots;\ v_{n1}, \ldots, v_{nm_n} : s_n) :? s = T$$

When the list of arguments is empty, the definition is simply written:

$$f :? s = T$$

It declares the operation name f as a total, respectively partial operation, with a profile having argument sorts s_1 (m_1 times), ..., s_n (m_n times) and result sort s. It also asserts the strong equation:

$$f(v_{11}, \ldots, v_{nm_n}) = T$$

universally quantified over the declared argument variables (which must be distinct, and are the only ones allowed in T), or just '$f = T$' when the list of arguments is empty.

In each of the above cases, the operation name f may occur in the term T, and may have *any* interpretation satisfying the equation—not necessarily the least fixed point.

2.1.3 Predicates

```
PRED-ITEMS ::= pred-items PRED-ITEM+
PRED-ITEM  ::= PRED-DECL | PRED-DEFN
PRED-NAME  ::= ID
```

A list `PRED-ITEMS` of predicate declarations and definitions is written:

preds PI_1; ... PI_n;

2.1.3.1 Predicate Declarations

```
PRED-DECL ::= pred-decl PRED-NAME+ PRED-TYPE
```

A predicate declaration `PRED-DECL` is written:

$$p_1, \ldots, p_n : T$$

It declares each predicate name p_1, ..., p_n as a predicate, with profile as specified by the predicate type T.

Predicate Types

```
PRED-TYPE ::= pred-type SORT-LIST
```

A predicate type `PRED-TYPE` with some argument sorts is written:

$$s_1 \times \ldots \times s_n$$

The sign displayed as '$\times$' may be input as '$\times$' in ISO Latin-1, or as '`*`' in ASCII. When the list of argument sorts is empty, the type is written '()'.

The predicate profile determined by the type has argument sorts s_1, ..., s_n.

2.1.3.2 Predicate Definitions

```
PRED-DEFN ::= pred-defn PRED-NAME PRED-HEAD FORMULA
PRED-HEAD ::= pred-head ARG-DECL*
```

A definition `PRED-DEFN` of a predicate with some arguments is written:

$p(v_{11}, \ldots, v_{1m_1} : s_1; \ \ldots; \ v_{n1}, \ldots, v_{nm_n} : s_n) \Leftrightarrow F$

When the list of arguments is empty, the definition is simply written:

$p \Leftrightarrow F$

The sign displayed as '$\Leftrightarrow$' is input as '`<=>`'.

It declares the predicate name p as a predicate, with a profile having argument sorts s_1 (m_1 times), ..., s_n (m_n times). It also asserts the equivalence:

$p(v_{11}, \ldots, v_{nm_n}) \Leftrightarrow F$

universally quantified over the declared argument variables (which must be distinct, and are the only ones allowed in F), or just '$p \Leftrightarrow F$' when the list of arguments is empty. The predicate name p may occur in the formula F, and may have *any* interpretation satisfying the equivalence.

2.1.4 Datatypes

```
DATATYPE-ITEMS ::= datatype-items DATATYPE-DECL+
```

A list `DATATYPE-ITEMS` of datatype declarations is written:

types $DD_1; \ \ldots \ DD_n;$

The order of the datatype declarations is *not* significant: there is ***non-linear visibility*** of the declared sorts in a list (in contrast to the linear visibility between the `BASIC-ITEMS` of a `BASIC-SPEC`, and between the `SIG-ITEMS` of a `SORT-GEN`).

2.1.4.1 Datatype Declarations

```
DATATYPE-DECL ::= datatype-decl SORT ALTERNATIVE+
```

A datatype declaration `DATATYPE-DECL` is written:

$s ::= A_1 \mid \ldots \mid A_n$

It declares the sort s. For each alternative construct $A_1, \ldots, A_n$, it declares the specified constructor and selector operations, and determines sentences asserting the expected relationship between selectors and constructors. All sorts used in an alternative construct must be declared in the local environment (which always includes the sort declared by the datatype declaration itself).

Note that a datatype declaration allows models where the ranges of the constructors are not disjoint, and where not all values are the results of constructors. This looseness can be eliminated in a general way by use of free extensions in structured specifications (as summarized in Part II), or by use of free datatypes within basic specifications (see below). Unreachable values can be eliminated also by the use of sort generation constraints.

Alternatives

```
ALTERNATIVE       ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT
TOTAL-CONSTRUCT   ::= total-construct   OP-NAME COMPONENTS*
PARTIAL-CONSTRUCT ::= partial-construct OP-NAME COMPONENTS+
```

A total constructor `TOTAL-CONSTRUCT` with some components is written:

$f(C_1; \ldots; C_n)$

When the list of components is empty, the constructor is simply written 'f'.

A partial constructor `PARTIAL-CONSTRUCT` with some components is written:

$f(C_1; \ldots; C_n)?$

(Partial constructors without components are not expressible in datatype declarations.)

The alternative declares f as an operation. Each component $C_1, \ldots, C_n$ specifies one or more argument sorts for the profile; the result sort is the sort s declared by the enclosing datatype declaration.

Components

```
COMPONENTS     ::= TOTAL-SELECT | PARTIAL-SELECT | SORT
TOTAL-SELECT   ::= total-select   OP-NAME+ SORT
PARTIAL-SELECT ::= partial-select OP-NAME+ SORT
```

A declaration `TOTAL-SELECT` of total selectors is written:

$f_1, \ldots, f_n : s$

A declaration `PARTIAL-SELECT` of partial selectors is written:

$f_1, \ldots, f_n :? s$

The remaining case is a component sort without any selector, simply written 's'.

In the first two cases, it provides n components: the sort s is taken as an argument sort n times for the constructor operation declared by the enclosing alternative, and it declares $f_1, \ldots, f_n$ as selector operations for the respective components. In the first case, each selector operation is declared as total, and in the second case, as partial. It also determines sentences that define the value of each selector on the values given by the constructor of the enclosing alternative.

In the last case, it provides the sort s only once as an argument sort for the constructor of the enclosing alternative, and it does not declare any selector operation for that component.

Note that when there is more than one alternative construct in a datatype declaration, selectors are usually partial, and must therefore be declared as such; their values on constructs for which they are not declared as selectors are left unspecified. A list of datatype declarations must not declare a function symbol both as a constructor and selector with the same profiles.

2.1.4.2 Free Datatype Declarations

```
FREE-DATATYPE ::= free-datatype DATATYPE-ITEMS
```

A list `FREE-DATATYPE` of free datatype declarations is written:

free types DD_1; ... DD_n;

This construct is only well-formed when all the constructors declared by the datatype declarations are total. Moreover, the constructors and selectors must be distinct (as qualified symbols) from each other and from the operations declared in the local environment.

The free datatype declarations declare the same sorts, constructors, and selectors as ordinary datatype declarations. Apart from the sentences that define the values of selectors, the free datatype declarations determine additional sentences requiring that the constructors are injective, that the ranges of constructors of the same sort are disjoint, that all the declared sorts are generated by the constructors, and that the value of applying a selector to a constructor for which it has not been declared is always undefined. (The sentences ensure that the models, if any, are the same as for a free extension

with the datatype declarations, provided that the sorts and qualified operation symbols declared by the datatype declaration are not already declared in the local environment.)

When the alternatives of a free datatype declaration are all constants, the declared sort corresponds to an (unordered) enumeration type.

2.1.5 Sort Generation

```
SORT-GEN ::= sort-gen SIG-ITEMS+
```

A sort generation `SORT-GEN` is written:

generated { $SI_1 \ldots SI_n$ **}**;

When the list of `SIG-ITEMS` is a single `DATATYPE-ITEMS` construct, writing the grouping signs is optional:

generated types DD_1; ... DD_n;

(The terminating '`;`' is optional in both cases.)

It determines the same elements of signature and sentences as $SI_1, \ldots, SI_n$, together with a corresponding sort generation constraint sentence: all the declared sorts of $SI_1, \ldots, SI_n$ are required to be generated by all the declared operations—but *excluding* operations declared as *selectors* by datatype declarations. A `SORT-GEN` is ill-formed if it does not declare any sorts.

2.2 Variables

Variables for use in terms may be declared globally, locally, or with explicit quantification. Globally or locally declared variables are implicitly universally quantified in subsequent axioms of the enclosing basic specification. Variables are not included in the declared signature.

Note that universal quantification over a variable that does not occur free in an axiom is semantically irrelevant, due to the assumption that all carriers are non-empty.

2.2.1 Global Variable Declarations

```
VAR-ITEMS ::= var-items VAR-DECL+
```

A list `VAR-ITEMS` of variable declarations is written:

vars VD_1; ... VD_n;

Note that local variable declarations are written in a similar way, but followed directly by a bullet ' • ' instead of the optional semicolon.

```
VAR-DECL ::= var-decl VAR+ SORT
VAR      ::= SIMPLE-ID
```

A variable declaration `VAR-DECL` is written:

$v_1, \ldots, v_n : s$

It declares the variables v_1, ..., v_n of sort s for use in subsequent axioms, but it does *not* contribute to the declared signature.

The scope of a global variable declaration is the subsequent axioms of the enclosing basic specification; a later declaration for a variable with the same identifier overrides the earlier declaration (regardless of whether the sorts of the variables are the same). A global declaration of a variable is equivalent to adding a universal quantification on that variable to the subsequent axioms of the enclosing basic specification.

2.2.2 Local Variable Declarations

```
LOCAL-VAR-AXIOMS ::= local-var-axioms VAR-DECL+ AXIOM+
```

A localization `LOCAL-VAR-AXIOMS` of variable declarations to a list of axioms is written:[1]

$\forall VD_1; \ldots; VD_n \bullet F_1 \ldots \bullet F_m;$

The sign displayed as ' • ' may be input as '·' in ISO Latin-1, or as '.' in ASCII.

It declares variables for local use in the axioms F_1, ..., F_m, but it does *not* contribute to the declared signature. A local declaration of a variable is equivalent to adding a universal quantification on that variable to all the indicated axioms.

[1] A `LOCAL-VAR-AXIOMS` may still be written also **vars** $VD_1; \ldots; VD_n \bullet F_1 \ldots \bullet F_m;$, for backwards compatibility with CASL version 1.0.

2.3 Axioms

```
AXIOM-ITEMS ::= axiom-items AXIOM+
AXIOM       ::= FORMULA
```

A list `AXIOM-ITEMS` of axioms is written:[2]

- F_1 ... • F_n

Each well-formed axiom determines a sentence of the underlying basic specification (closed by universal quantification over all declared variables).

```
FORMULA ::= QUANTIFICATION | CONJUNCTION | DISJUNCTION
          | IMPLICATION | EQUIVALENCE | NEGATION | ATOM
```

A formula is constructed from atomic formulae of the form `ATOM` using quantification and the usual logical connectives.

Keywords in formulae and terms are displayed in the same font as identifiers.

2.3.1 Quantifications

```
QUANTIFICATION ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER     ::= universal | existential | unique-existential
```

A quantification with the `universal` quantifier is written:

$\forall VD_1; \ldots; VD_n \bullet F$

The sign displayed as '$\forall$' is input as '`forall`'. The sign displayed as '$\bullet$' may be input as '$\cdot$' in ISO Latin-1, or as '`.`' in ASCII.

A quantification with the `existential` quantifier is written:

$\exists VD_1; \ldots; VD_n \bullet F$

A quantification with the `unique-existential` quantifier is written:

$\exists! VD_1; \ldots; VD_n \bullet F$

The sign displayed as '$\exists$' is input as '`exists`'.

The first case is universal quantification, holding when the body F holds for all values of the quantified variables; the second case is existential quantification, holding when the body F holds for some values of the quantified variables; and the last case is unique existential quantification, abbreviat-

[2] An `AXIOM-ITEMS` may still be written also **axioms** F_1; ... F_n; , for backwards compatibility with CASL version 1.0.

ing a formula that holds when the body F holds for unique values of the quantified variables.

The formula $\forall VD_1; \ldots; VD_n \bullet F$ is equivalent to $\forall VD_1 \bullet \ldots \forall VD_n \bullet F$; and $\forall v_1, \ldots, v_n : s \bullet F$ is equivalent to $\forall v_1 : s \bullet \ldots \forall v_n : s \bullet F$. Similarly for the other quantifiers. The scope of a variable declaration in a quantification is the component formula F, and an inner declaration for a variable with the same identifier as in an outer declaration overrides the outer declaration (regardless of whether the sorts of the variables are the same). Note that the body of a quantification extends as far as possible.

2.3.2 Logical Connectives

These formulae determine the usual logical connectives on the sub-formulae. Conjunction and disjunction apply to lists of two or more formulae; they both have weaker precedence than negation. When mixed, they have to be explicitly grouped, using parentheses '(...)'.

Both implication (which may be written in two different ways) and equivalence have weaker precedence than conjunction and disjunction. When the 'forward' version of implication is iterated, it is implicitly grouped to the right; the 'backward' version is grouped to the left. When these constructs are mixed, they have to be explicitly grouped.

2.3.2.1 Conjunction

```
CONJUNCTION ::= conjunction FORMULA+
```

A conjunction is written:

$F_1 \wedge \ldots \wedge F_n$

The sign displayed as '$\wedge$' is input as '`/\`'.

2.3.2.2 Disjunction

```
DISJUNCTION ::= disjunction FORMULA+
```

A disjunction is written:

$F_1 \vee \ldots \vee F_n$

The sign displayed as '$\vee$' is input as '`\/`'.

2.3.2.3 Implication

```
IMPLICATION ::= implication FORMULA FORMULA
```

An implication is written:

$F_1 \Rightarrow F_2$

The sign displayed as '⇒' is input as '`=>`'. An implication may also be written in reverse order:

F_2 *if* F_1

2.3.2.4 Equivalence

```
EQUIVALENCE ::= equivalence FORMULA FORMULA
```

An equivalence is written:

$F_1 \Leftrightarrow F_2$

The sign displayed as '⇔' is input as '`<=>`'.

2.3.2.5 Negation

```
NEGATION ::= negation FORMULA
```

A negation is written:

$\neg F_1$

The sign displayed as '¬' may be input as '¬' in ISO Latin-1, or as '`not`' in ASCII.

2.3.3 Atomic Formulae

```
ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
       | EXISTL-EQUATION | STRONG-EQUATION
```

An *atomic formula* `ATOM` is well-formed (with respect to the local environment and variable declarations) if it is well-sorted and expands to a unique atomic formula for constructing sentences. The notions of when an atomic formula is ***well-sorted***, of when a term is *well-sorted for a particular sort*, and of the ***expansions*** of atomic formulae and terms, are indicated below for the various constructs.

Due to overloading of predicate and/or operation symbols, a well-sorted atomic formula or term may have several expansions, preventing it from being well-formed. Qualifications on operation and predicate symbols may be used to determine the intended expansion and make it well-formed; explicit sorts on arguments and/or results may also help to avoid unintended expansions.

2.3.3.1 Truth

```
TRUTH ::= true-atom | false-atom
```

The atomic formulae `true-atom` and `false-atom` are written '*true*', '*false*'.

They are always well-sorted, and expand to primitive sentences, such that the sentence for '*true*' always holds, and the sentence for '*false*' never holds.

2.3.3.2 Predicate Application

```
PREDICATION    ::= predication PRED-SYMB TERMS
PRED-SYMB      ::= PRED-NAME | QUAL-PRED-NAME
QUAL-PRED-NAME ::= qual-pred-name PRED-NAME PRED-TYPE
TERMS          ::= terms TERM*
```

An application of a predicate symbol PS to some argument terms is written:

$$PS(T_1, \ldots, T_n)$$

When PS is a mixfix identifier, consisting of a sequence '$t_0 \_\_ \ldots \_\_ t_n$' of tokens or spaces t_i separated by place-holders '__', the application may also be written:

$$t_0\, T_1\, t_1 \ldots T_n\, t_n$$

When the predicate symbol is a constant p with no argument terms, its application is simply written 'p'.

A qualified predicate name `QUAL-PRED-NAME` with type T is written:

$$(pred \;\; p : T)$$

An unqualified predicate name `PRED-NAME` is simply written 'p'.

The application of the predicate symbol is well-sorted when there is a declaration of the predicate name (with the argument sorts indicated by the indicated type in the case of a qualified predicate name) such that all the argument terms are well-sorted for the respective argument sorts. It then expands to an application of the qualified predicate name to the fully-qualified expansions of the argument terms for those sorts.

2.3.3.3 Definedness

```
DEFINEDNESS ::= definedness TERM
```

A definedness formula is written:

def T

It is well-sorted when the term is well-sorted for some sort. It then expands to a definedness assertion on the fully-qualified expansion of the term.

2.3.3.4 Equations

```
EXISTL-EQUATION ::= existl-equation TERM TERM
STRONG-EQUATION ::= strong-equation TERM TERM
```

An existential equation `EXISTL-EQUATION` is written:

$T_1 \stackrel{e}{=} T_2$

The sign displayed as '$\stackrel{e}{=}$' is input as '`=e=`'.

A strong equation is written:

$T_1 = T_2$

An existential equation holds when the values of the terms are both defined and equal; a strong equation holds also when the values of both terms are undefined (thus the two forms of equation are equivalent when the values of both terms are always defined).

An equation is well-sorted if there is a sort such that both terms are well-sorted for that sort. It then expands to the corresponding existential or strong equation on the fully-qualified expansions of the terms for that sort.

2.3.4 Terms

```
TERM ::= SIMPLE-ID | QUAL-VAR | APPLICATION
       | SORTED-TERM | CONDITIONAL
```

A term is constructed from constants and variables by applications of operations. All names used in terms may be qualified by the intended types, and the intended sort of the term may be specified. Note that the condition of a conditional term is a formula, not a term.

2.3.4.1 Identifiers

An unqualified simple identifier in a term may be a variable or a constant, depending on the local environment and the variable declarations. Either is well-sorted for the sort specified in its declaration; a variable expands to the (sorted) variable itself, whereas a constant expands to an application of the qualified symbol to the empty list of arguments. Note that when an identifier is declared both as variable and as a constant of the same sort, unqualified use of the identifier always makes the enclosing atomic formula ill-formed.

2.3.4.2 Qualified Variables

```
QUAL-VAR ::= qual-var VAR SORT
```

A qualified variable `QUAL-VAR` is written:

$(var\ \ v : s)$

It is well-sorted for the sort s.

2.3.4.3 Operation Application

```
APPLICATION  ::= application OP-SYMB TERMS
OP-SYMB      ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME ::= qual-op-name OP-NAME OP-TYPE
TERMS        ::= terms TERM*
```

An application of an operation symbol OS to some argument terms is written:

$OS(T_1, \ldots, T_n)$

When OS is a mixfix identifier, consisting of a sequence '$t_0 \_ \_ \ldots \_ \_ t_n$' of tokens or spaces t_i separated by place-holders '_', the application may also be written:

$t_0\, T_1\, t_1 \ldots T_n\, t_n$

When the operation symbol is a constant c with no argument terms, its application is simply written 'c'.

Declaring different mixfix identifiers that involve some common tokens may lead to ambiguity, with different candidate groupings of the same sequence of tokens and terms. Such ambiguity prevents the enclosing atomic formula from being well-formed, irrespective of the declared profiles of the symbols involved, and generally has to be eliminated by use of explicit grouping parentheses. However, to allow the omission of some parentheses, infix iden-

tifiers are given weaker precedence than prefix identifiers, which in turn are given weaker precedence than postfix identifiers. (The mixfix identifier '__ __' is allowed, and regarded as an infix, although this is unlikely to be the case in higher-order extensions of CASL, since there juxtaposition will be reserved for function application.)

In an application, a qualified operation name `QUAL-OP-NAME` with f qualified by the operation type T is written:

$(op\ \ f : T)$

When the qualified operation name is a constant c, its application (to no arguments) is written $(op\ \ c : T)$.

The application is well-sorted for some particular sort when there is a declaration of the operation name (with the argument and result sorts indicated by the type, if specified) such that all the argument terms are well-sorted for the respective argument sorts, and the result sort is the required sort. It then expands to an application of the qualified operation name to the fully-qualified expansions of the argument terms for those sorts.

2.3.4.4 Sorted Terms

```
SORTED-TERM ::= sorted-term TERM SORT
```

A sorted term is written:

$T : s$

It is well-sorted for some sort if the component term T is well-sorted for the specified sort s. It then expands to those of the fully-qualified expansions of the component term that have the specified sort.

2.3.4.5 Conditional Terms

```
CONDITIONAL ::= conditional TERM FORMULA TERM
```

A conditional term is written:

$T_1\ when\ F\ else\ T_2$

It is well-sorted for some sort when both T_1 and T_2 are well-sorted for that sort and F is a well-formed formula. The enclosing *atomic* formula '$A[T_1\ when\ F\ else\ T_2]$' expands to '$(A[T_1]\ if\ F) \wedge (A[T_2]\ if\ \neg F)$'. When several conditional terms occur in the same atomic formula, the expansions are made in a fixed but arbitrary order (all orders yield equivalent formulae).

2.4 Identifiers

```
SIMPLE-ID ::= WORDS
ID        ::= TOKEN-ID
TOKEN-ID  ::= TOKEN
TOKEN     ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
```

The internal structure of identifiers `ID`, used to identify sorts, operations, and predicates, is insignificant in the abstract syntax of basic many-sorted specifications. (`ID` is extended with compound identifiers, whose structure is significant, in connection with generic specifications in Section 6.5.)

In concrete syntax, an identifier may be written as a single ***token***: either a sequence of letters and/or digits—possibly mixed with single underscores (_) and/or primes (’), and possibly prefixed by a dot (.)—or a sequence of other printable ISO Latin-1 characters (excluding `( ) ; , ‘ " %`). Keywords, and various other sequences that could be confused with separators, are not allowed as tokens in the input syntax (however, ***display annotations*** may be used to produce them when formatting identifiers).

```
ID             ::= ... | MIXFIX-ID
MIXFIX-ID      ::= TOKEN-PLACES
TOKEN-PLACES   ::= token-places TOKEN-OR-PLACE+
TOKEN-OR-PLACE ::= TOKEN | PLACE
```

An identifier may also be a ***mixfix identifier*** ‘$t_0\_\_\ldots\_\_t_n$’, consisting of a list of tokens or spaces t_i interspersed with *place-holders*, each place-holder being written as a *pair* of underscores ‘__’. Mixfix identifiers allow the use of ***mixfix notation***[3] for application of operations and predicates to argument terms in concrete syntax. A mixfix identifier such as `f__` is a different symbol from `f`. An application of the (unqualified) symbol $f\_\_$ to x may be written as `f x`, `f(x)`, `f__(x)`; an application of f to x may only be written as `f(x)`. ‘Invisible’ identifiers, consisting entirely of two or more place-holders (separated by spaces), are allowed.

Braces ‘{’, ‘}’ and square brackets ‘[’, ‘]’ are allowed as complete tokens in identifiers; however, any occurrences of these characters in a declared identifier must be balanced; e.g., ‘{[__}]’ and ‘{__]’ are *not* allowed.

An identifier `ID` may be used simultaneously to identify different kinds of entities (sorts, operations, and predicates) in the same local environment. It would not, however, be appropriate to use it simultaneously for constants and variables of the same sort, since its (unqualified) use would then always be ambiguous, making the enclosing formula ill-formed.

[3] Mixfix notation is so-called because it generalizes infix, prefix, and postfix notation to allow arbitrary mixing of argument positions and identifier tokens.

Chapter 3

Subsorting Concepts

This chapter introduces the signatures, models, and sentences characterizing basic specifications with subsorts, extending Chapter 1. The notion of satisfaction for subsorted specifications is essentially as for many-sorted specifications.

The intuition behind the treatment of subsorts adopted here is to represent subsort inclusion by embedding (which is not required to be identity), commuting, as usual in order-sorted approaches, with overloaded operation symbols. In the language described in Chapter 4, however, no conditions such as 'regularity' are imposed on signatures. Instead, terms and sentences that can be given different parses (up to the commutativity between embedding and overloaded symbols) are simply rejected as ill-formed.

3.1 Signatures

A ***subsorted signature*** $\Sigma = (S, TF, PF, P, \leq)$ consists of a many-sorted signature (S, TF, PF, P) together with a pre-order $\leq$ of *subsort embedding* on the set S of sorts. $\leq$ is extended pointwise to sequences of sorts.

For a subsorted signature, we define ***overloading relations*** for operation and predicate symbols. Let $f \in (TF_{w_1,s_1} \cup PF_{w_1,s_1}) \cap (TF_{w_2,s_2} \cup PF_{w_2,s_2})$ and $p \in P_{w_1} \cap P_{w_2}$. Two qualified operation symbols f_{w_1,s_1} and f_{w_2,s_2} are in the *overloading relation* (written $f_{w_1,s_1} \sim_F f_{w_2,s_2}$) iff there exists a $w \in S^*$ and $s \in S$ such that $w \leq w_1, w_2$ and $s_1, s_2 \leq s$. Similarly, two qualified predicate symbols p_{w_1} and p_{w_2} are in the overloading relation (written $p_{w_1} \sim_P p_{w_2}$) iff there exists a $w \in S^*$ such that $w \leq w_1, w_2$. We say that two profiles of a symbol are in the overloading relation if the corresponding qualified symbols are in overloading relation.

Note that two profiles of an overloaded constant declared with different sorts are in the overloading relation iff the two sorts have a common supersort.

A ***subsorted signature morphism*** $\sigma : \Sigma \to \Sigma'$ is a many-sorted signature morphism that preserves the subsort relation and the overloading relations.

With each subsorted signature $\Sigma = (S, TF, PF, P, \leq)$ a many-sorted signature $\Sigma^{\#}$ is associated, extending (S, TF, PF, P) for each pair of sorts $s \leq s'$ by a total embedding operation (from s into s'), a partial projection operation (from s' onto s), and a membership predicate (testing whether values in s' are embeddings of values in s). The symbols used for embedding, projection, and membership are chosen arbitrarily so as not to be in Σ.

Any subsorted signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ expands to a many-sorted signature morphism $\sigma^{\#} : \Sigma_1^{\#} \to \Sigma_2^{\#}$, preserving the symbols used for embedding, projection, and membership.

3.2 Models

For a subsorted signature Σ the ***subsorted models*** are ordinary many-sorted models for $\Sigma^{\#}$ that satisfy the following properties (which can be formalized as a set of conditional axioms):

- Embedding operations are total and 1-1; projection operations are partial, and 1-1 when defined.
- The embedding of a sort into itself is the identity function.
- All compositions of embedding operations between the same two sorts are equal functions.
- Embedding followed by projection is identity; projection followed by embedding is included in identity.
- Membership in a subsort holds just when the projection to the subsort is defined.
- Embedding is compatible with those operations and predicates that are in the overloading relations.

3.3 Sentences

For a subsorted signature Σ, the ***subsorted sentences*** are the ordinary many-sorted sentences (as defined in Chapter 1) for the associated many-sorted signature $\Sigma^{\#}$.

Chapter 4

Subsorting Constructs

This chapter indicates the abstract and concrete syntax of the constructs of *subsorted* basic specifications, and describes their intended interpretation, extending what was provided for many-sorted specifications in Chapter 2.

A well-formed subsorted basic specification `BASIC-SPEC` of the CASL language determines a basic specification of the underlying subsorted institution, consisting of a subsorted signature and a set of sentences of the form described in Chapter 3. This signature and the class of models over it that satisfy the set of sentences provide the semantics of the basic specification.

4.1 Signature Declarations

4.1.1 Sorts

```
SORT-ITEM ::= ... | SUBSORT-DECL | ISO-DECL | SUBSORT-DEFN
```

4.1.1.1 Subsort Declarations

```
SUBSORT-DECL ::= subsort-decl SORT+ SORT
```

A subsort declaration `SUBSORT-DECL` is written:

$s_1, \ldots, s_n < s$

It declares all the sorts s_1, ..., s_n, and s, as well as the embedding of each s_i as a subsort of s. The s_i must be distinct from s.

When a subsort declaration occurs in a sort generation construct, the embedding and projection operations between the subsort(s) and the supersort are treated as declared operations with regard to generation of sorts.

Introducing an embedding relation between two sorts may cause operation symbols to become related by the overloading relation, so that values of terms become equated when the terms are identical up to embedding.

4.1.1.2 Isomorphism Declarations

```
ISO-DECL ::= iso-decl SORT+
```

An isomorphism declaration `ISO-DECL` is written:

$$s_1 = \ldots = s_n$$

It declares all the sorts $s_1, \ldots, s_n$, as well as their embeddings as subsorts of each other. Thus the carriers for the sorts s_i are required to be isomorphic. The s_i must be distinct.

4.1.1.3 Subsort Definitions

```
SUBSORT-DEFN ::= subsort-defn SORT VAR SORT FORMULA
```

A subsort definition `SUBSORT-DEFN` is written:

$$s = \{v : s' \bullet F\}$$

The sign displayed as ' $\bullet$ ' may be input as '$\cdot$' in ISO Latin-1, or as '.' in ASCII. It provides an explicit specification of the values of the subsort s of s', in contrast to the implicit specification provided by using subsort declarations and overloaded operation symbols.

The subsort definition declares the sort s; it declares the embedding of s as a subsort of s', which must already be declared in the local environment; and it asserts that the values of s are precisely (the projection of) those values of the variable v from s' for which the formula F holds.

The scope of the variable v is restricted to the formula F. Any other variables occurring in F must be explicitly declared by enclosing quantifications.

Note that the terms of sort s' cannot generally be used as terms of sort s. But they can be explicitly projected to s, using a cast.

Defined subsorts may be separately related using subsort (or isomorphism) declarations—implication or equivalence between their defining formulae does *not* give rise to any subsort relationship between them.

4.1.2 Datatypes

Datatype declarations are unchanged, except for a new kind of alternative:

4.1.2.1 Alternatives

```
ALTERNATIVE ::= ... | SUBSORTS
SUBSORTS    ::= subsorts SORT+
```

A subsorts alternative is written:

sorts $s_1, \ldots, s_n$

As with sort declarations, the plural keyword may be written in the singular (regardless of the number of sorts).

The sorts s_i, which must be already declared in the local environment, are declared to be embedded as subsorts of the sort declared by the enclosing datatype declaration. ('*sorts* $s_1, \ldots, s_n$' and '*sort* s_1 | ... | *sort* s_n' are equivalent.)

In a free datatype declaration, all the sorts that are embedded in the declared sort by the alternatives must have no common subsorts. When the alternatives of a free datatype declaration are all subsorts, the declared sort corresponds to the disjoint union of the subsorts. Finally, consider the set of qualified constructor and selector symbols declared by the free datatype: no element of this set may be in the overloading relation with any other element, nor with the qualified operation symbols from the local environment.

4.2 Axioms

4.2.1 Atomic Formulae

```
ATOM ::= ... | MEMBERSHIP
```

As for many-sorted specifications, an atomic formula is well-formed (with respect to the current declarations) if it is well-sorted and expands to a unique atomic formula for constructing sentences of the underlying institution—but now for subsorted specifications, uniqueness is required only up to an ***equivalence*** on atomic formulae and terms. This equivalence is the least one including fully-qualified terms that are the same up to profiles of operation symbols in the overloading relation $\sim_F$ and embedding, and fully-qualified atomic formulae that are the same up to the profiles of predicate symbols in the overloading relation $\sim_P$ and embedding.

The notions of when an atomic formula or term is *well-sorted* and of its *expansion* are indicated below for the various subsorting constructs. Due not only to overloading of predicate and/or operation symbols, but also to implicit embeddings from subsorts into supersorts, a well-sorted atomic formula may have several non-equivalent expansions, preventing it from being well-formed. Qualifications on operation and predicate symbols, or explicit sorts on terms, may be used to determine the intended expansion (up to the equivalence indicated above) and make the enclosing formula well-formed.

4.2.1.1 Membership

```
MEMBERSHIP ::= membership TERM SORT
```

A membership formula is written:

$T \in s$

The sign displayed as '$\in$' is input as '`in`'.

It is well-sorted if the term T is well-sorted for a supersort s' of the specified sort s. It expands to an application of the pre-declared predicate symbol for testing s' values for membership in the embedding of s.

4.2.2 Terms

```
TERM ::= ... | CAST
```

4.2.2.1 Casts

```
CAST ::= cast TERM SORT
```

A cast term is written:

T *as* s

It is well-sorted if the term T is well-sorted for a supersort s' of s. It expands to an application of the pre-declared operation symbol for projecting s' to s.

Term formation is also extended by letting a well-sorted term of a subsort s be regarded as a well-sorted term of a supersort s' as well, which provides implicit embedding. It expands to the explicit application of the pre-declared operation symbol for embedding s into s'. (There are no implicit projections.) Also a sorted-term $T : s'$ expands to an explicit application of an embedding, provided that the apparent sort s of the component term T is a subsort of the specified sort s'.

Part II

Structured Specifications

Chapter 5

Structuring Concepts

A basic specification, as described in Part I, consists essentially of a signature Σ (declaring symbols) and a set of sentences (axioms or constraints) over Σ. The semantics of a well-formed basic specification is the specified signature Σ together with the class of all Σ-models that satisfy the specified sentences.

A ***structured specification*** is formed by combining specifications in various ways, starting from basic specifications. For instance, specifications may be ***united***; a specification may be ***extended*** with further signature items and/or sentences; parts of a signature may be ***hidden***; the signature may be ***translated*** to use different symbols (with corresponding translation of the sentences) by a signature morphism; and models may be restricted to ***initial*** models. The abstract syntax of constructs in the CASL language for presenting such structured specifications is described in Chapter 6.

The structuring concepts and constructs and their semantics do not depend on a specific framework of basic specifications. This means that Part I of the CASL language design is orthogonal to Part II (and also to Parts III and IV). Therefore, CASL basic specifications as given in Part I can be restricted to sublanguages or extended in various ways without the need to reconsider or to change Parts II, III, and IV. [1]

The semantics of a well-formed structured specification is of the same form as that of a basic specification: a signature Σ together with a class of Σ-models. Thus the structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. (Specification of the *architecture* of models in the CoFI framework is addressed in Part III.)

Within a structured specification, the ***current signature*** may vary. For

[1] The occasional reference to the subsort and overloading relations in Part II may simply be ignored (or replaced by the identity relation) when the framework for basic specifications is restricted so as not to include these features.

instance, when two specifications are united, the signature valid in the one is generally different from that valid in the other. The association between symbols and their declarations as given by the valid signature is called the *local environment*.

Parts of structured specifications, in contrast to arbitrary parts of basic specifications, are potentially reusable—either verbatim, or with the adjustment of some ***parameters***. Specifications may be *named*, so that the reuse of a specification may be replaced by a *reference* to it through its name. (Libraries of named specifications are explained in Part IV.) The current association between names and the specifications that they reference is called the *global environment*. Named specifications are implicitly ***closed***, not depending on a local environment of declared symbols. A reference to the name of a specification is equivalent to the referenced specification itself, provided that the closedness is explicitly ensured.

A named specification may declare some ***parameters***, the union of which is extended by a ***body***; it is then called ***generic***. A reference to a generic specification should ***instantiate*** it by providing, for each parameter, an ***argument specification*** together with a ***fitting morphism*** from the parameter to the argument specification. Fitting may also be achieved by (explicit) use of named ***views*** between the parameter and argument specifications. The union of the arguments, together with the translation of the generic specification by an expansion of the fitting morphism, corresponds to a so-called push-out construction—taking into account any explicit ***imports*** of the generic specification, which allow symbols used in the body to be declared also by arguments.

The semantics of structured specifications involve signature morphisms and the corresponding reducts on models. For instance, hiding some symbols in a specification corresponds to a signature morphism that injects the non-hidden symbols into the original signature; the models, after hiding the symbols, are the reducts of the original models along this morphism. Translation goes the other way: the reducts of models over the translated signature back along the morphism give the original models. For the semantics of structured specifications (in particular, those involving hiding) the axiom of choice is assumed.

The semantics of views involves also ***specification morphisms***, which are signature morphisms between particular specifications such that the reduct of each model of the target specification is a model of the source specification.

Given a signature Σ with symbols $|\Sigma|$, ***symbol sets*** and *symbol mappings* determine signature morphisms as follows:

- A set of symbols in $|\Sigma|$ determines the inclusion of the *smallest* subsignature of Σ that contains these symbols. (When an operation or

predicate symbol is included, all the sorts in its profile have to be included too.)

It also determines the inclusion of the *largest* subsignature of Σ that does not contain any of these symbols. (When a sort is not included, no operation or predicate symbol with that sort in its profile can be included either.)

- A mapping of symbols in $|\Sigma|$ determines the morphism from Σ that extends this mapping with identity maps for all the remaining names in $|\Sigma|$. In the case that the signature morphism does not exist, the enclosing construct is ill-formed.
- Given another signature Σ', a mapping of symbols in $|\Sigma|$ to symbols in $|\Sigma'|$ determines the unique signature morphism from Σ to Σ' that extends the given mapping, and then is the identity, as far as possible, on common names of Σ and Σ'. (Mapping an operation or predicate symbol implies mapping the sorts in the profile consistently.) In the case that the signature morphism does not exist or is not unique, the enclosing construct is ill-formed.

Chapter 6

Structuring Constructs

This chapter indicates the abstract and concrete syntax of the constructs of *structured* specifications, and describes their intended interpretation, extending what was provided for basic (many-sorted and subsorted) specifications in Part I.

The summary below indicates when structured specifications are well-formed, and how their signatures and classes of models are determined by those of their component specifications. The interpretation is essentially based on model classes—a "flattening" reduction to sets of sentences is not possible, in general (due to the presence of constructs such as hiding and freeness).

A structured specification can only be well-formed when all its component specifications are well-formed.

6.1 Structured Specifications

```
SPEC ::= BASIC-SPEC | TRANSLATION | REDUCTION
       | UNION | EXTENSION | FREE-SPEC | LOCAL-SPEC
       | CLOSED-SPEC
```

A ***translation*** allows the symbols declared by a specification to be renamed; it may also be used to require that some symbols have been declared, e.g., when referencing a named specification. A ***reduction*** allows symbols to be hidden; for convenience, the remaining 'revealed' symbols may be simultaneously renamed. A ***union*** combines specifications such that when the declaration of a particular symbol is common to some of the combined specifications, its interpretation in a model has to be a common one too. An ***extension*** may ***enrich*** models by declaring new symbols and asserting their properties, and/or ***specialize*** the interpretation of already-declared

symbols. A ***free specification*** `FREE-SPEC` is used to restrict interpretations to ***free extensions***, with initiality as a special case. A ***local specification*** `LOCAL-SPEC` is used to specify ***auxiliary*** symbols for local use, hiding them afterwards. A *closed specification* `CLOSED-SPEC` ensures that the local environment provided to a specification is empty.

When the above constructs are combined in the same specification, the grouping is determined unambiguously by precedence rules: translations and reductions have the highest precedence, then come local specifications, then unions, and finally extensions have the lowest precedence. (Free specifications generally involve explicit grouping, and their relative precedence to the other constructs is irrelevant.) A different grouping may always be obtained by use of grouping braces: '**{** ... **}**'.

A specification `SPEC` may occur in a context (e.g., when it being named) where it is required to be ***self-contained*** or ***closed***, not depending on the local environment at all. In that case, it determines a signature and a class of models straightforwardly.

In structured specifications, however, a specification `SPEC` may also occur in a context where it is to ***extend*** other specifications, providing itself only part of a signature. Then it is interpreted as a (partial) function mapping signatures Σ to the corresponding extended signatures Σ', together with a partial function mapping model classes over Σ to model classes over Σ' (when defined). The signature and model class for the self-contained case above can be obtained by applying these functions to the empty signature and to the model class of the empty specification, respectively.

Translations and reductions in a `SPEC` are not allowed to affect symbols that are already in the local environment that is being extended. The other structuring constructs generalize straightforwardly from self-contained specifications to extensions.

6.1.1 Translations

```
TRANSLATION ::= translation SPEC RENAMING
RENAMING    ::= renaming  SYMB-MAP-ITEMS+
```

A translation is written:

SP **with** *SM*

Symbol mappings *SM* are described in Section 6.4.

The symbols mapped by *SM* must be among those declared by *SP*. The signature Σ given by *SP* and the mapping *SM* then determine a signature morphism to a signature Σ', as explained in Chapter 5. The morphism must

not affect the symbols already declared in the local environment, which is passed unchanged to *SP*.

The class of models of the translation consists exactly of those models over Σ' whose reducts along the morphism are models of *SP*.

If a partial operation symbol is renamed into a total one, this is only well-formed in case that the resulting operation symbol is already total due to another component of the renaming.

When the symbol mapping *SM* is simply a list of identity maps (which may be abbreviated to a simple list of symbols) the only effect of the translation on the semantics of *SP* is to require that the symbols listed are indeed included in the signature given by *SP*, otherwise the translation is not well-formed.

6.1.2 Reductions

```
REDUCTION   ::= reduction SPEC RESTRICTION
RESTRICTION ::= HIDDEN | REVEALED
HIDDEN      ::= hidden SYMB-ITEMS+
REVEALED    ::= revealed SYMB-MAP-ITEMS+
```

A hiding reduction is written:

SP **hide** *SL*

A revealing reduction is written:

SP **reveal** *SM*

Symbol lists *SL* and symbol mappings *SM* are described in Section 6.4.

The symbols listed by *SL*, or mapped by *SM*, must be among those declared by *SP*.

In the case of a hiding reduction, the signature Σ given by *SP* and the set of symbols listed by *SL* determine the inclusion of the largest subsignature Σ' of Σ that does *not* contain any of the listed symbols, as explained in Chapter 5. Note that hiding a sort entails hiding all the operations and predicate symbols whose profiles involve that sort.

In the case of a revealing reduction, the signature Σ given by *SP* and the set of symbols mapped by *SM* determine the inclusion of the smallest subsignature Σ' of Σ that contains all of the listed symbols, as explained in Chapter 5. Note that revealing an operation or predicate symbol entails revealing the sorts involved in its profile.

In both cases, the subsort embedding relation is inherited from that declared by SP, and a model class $\mathcal{M}$ is given by the reducts of the models of SP along the inclusion of Σ' in Σ.

In the case of a hiding reduction, its model class is simply $\mathcal{M}$. In the case of a revealing reduction, however, the signature Σ' and the mapping SM of (all) the symbols in it determine a signature morphism to a signature Σ'', as explained in Chapter 5. The class of models of the reduction then consists exactly of those models over Σ'' whose reducts along this morphism are in $\mathcal{M}$.

A reduction must not affect the symbols already declared in the local environment, which is passed unchanged to SP.

6.1.3 Unions

```
UNION ::= union SPEC+
```

A union is written:

SP_1 **and** ... **and** SP_n

When the current local environment is empty, each SP_i must determine a complete signature Σ_i. The signature of the union is obtained by the ordinary union of the Σ_i (not their disjoint union). Thus all (non-localized) occurrences of a symbol in the SP_i are interpreted uniformly (rather than being regarded as homonyms for potentially different entities). If the same name is declared both as a total and as a partial operation with the same profile (in different signatures), the operation becomes total in the union.

The models are those models of the union signature for which the reduct along the signature inclusion morphism from SP_i is a model of SP_i, for each $i = 1, \ldots, n$.

When the current local environment is non-empty, each SP_i must determine an extension from it to a complete signature Σ_i; then the resulting signature is determined as above. Similarly, models of the local environment are extended to models of the SP_i; then the resulting models are determined as above. This provides the required partial functions from signatures to signatures, and from model classes to model classes.

6.1.4 Extensions

```
EXTENSION ::= extension SPEC+
```

An extension is written:

SP_1 **then** ... **then** SP_n

When the current local environment is empty, SP_1 must determine a complete signature Σ_1; otherwise, it must determine an extension from the local environment to a complete signature Σ_1. For $i = 2, \ldots, n$ each SP_i must determine an extension from Σ_{i-1} to a complete signature Σ_i. The signature determined by the entire extension is then Σ_n.

Similarly, SP_1 determines a class of models M_1 over Σ_1. For $i = 2, \ldots, n$ each SP_i determines the class M_i of those models over Σ_i whose reducts to Σ_{i-1} are in M_{i-1}. The class of models determined by the entire extension is then M_n.

An annotation is provided (see Section C.5) for indicating that a series of extensions is conservative, i.e., every model in M_{i-1} is the reduct of some model in M_i, for $i = 2, \ldots, n$.

6.1.5 Free Specifications

```
FREE-SPEC ::= free-spec SPEC
```

A free specification `FREE-SPEC` is written:

free { *SP* }

Note that the specification written:

free types DD_1; ... DD_n;

is parsed as a free datatype of a basic specification, but it usually has the same interpretation as the free structured specification written:

free { types DD_1; ... DD_n; **}**

This equivalence holds at least in the framework for basic specifications given in Part I, under some minor restrictions: in a datatype declaration with more than one alternative, any selector that is declared as total for some alternative must be declared as a total selector with the same result sort for every other alternative; and the sorts and qualified operation symbols declared by the datatype declaration must not be already declared in the local environment.

When the current local environment is empty, *SP* must determine a complete signature Σ; otherwise, it must determine an extension from the local environment to a complete signature Σ. In both cases, Σ is the signature determined by the free specification.

When the current local environment is empty, the free specification determines the class of initial models of *SP*; otherwise, it determines the class of models that are free extensions for *SP* of their own reducts to models of the current local environment.

6.1.6 Local Specifications

```
LOCAL-SPEC ::= local-spec SPEC SPEC
```

A local specification `LOCAL-SPEC` is written:

local SP_1 **within** SP_2

It is equivalent to writing:

{ SP_1 **then** SP_2 **} hide** $SY_1, \ldots, SY_n$

where $SY_1, \ldots, SY_n$ are all the symbols declared by SP_1 that are not already in the current local environment. Thus the symbols $SY_1, \ldots, SY_n$ are only for local use in (SP_1 and) SP_2. The hiding must not affect symbols that are declared only in SP_2 (thus operation or predicate symbols declared in SP_2 should not have sorts declared by SP_1 in their profiles).

6.1.7 Closed Specifications

```
CLOSED-SPEC ::= closed-spec SPEC
```

A closed specification `CLOSED-SPEC` is written:

closed { SP **}**

It determines the same signature and class of models as SP determines in the empty local environment, thus ensuring the closedness of SP.

6.2 Named and Parametrized Specifications

Specifications are named by specification definitions, and referenced by use of the name. A named specification may also have some parameters, which have to be instantiated when referencing the specification.

6.2.1 Specification Definitions

```
SPEC-DEFN  ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY ::= genericity PARAMS IMPORTED
PARAMS     ::= params SPEC*
IMPORTED   ::= imported SPEC*
```

A generic specification definition `SPEC-DEFN` with some parameters and some imports is written:

spec *SN* [*SP*$_1$] ... [*SP*$_n$] **given** *SP*$''_1$, ..., *SP*$''_m$ =
SP
end

When the list of imports *SP*$''_1$, ..., *SP*$''_m$ is empty, the definition is written:

spec *SN* [*SP*$_1$] ... [*SP*$_n$] =
SP
end

When the list of parameters *SP*$_1$, ..., *SP*$_n$ is empty, the definition merely names a specification and is simply written:

spec *SN* =
SP
end

The terminating '**end**' keyword is optional.

It defines the name *SN* to refer to the specification that has parameter specifications *SP*$_1$, ..., *SP*$_n$ (if any), import specifications *SP*$''_1$, ..., *SP*$''_m$ (if any), and body specification *SP*. This extends the global environment (which must not already include a definition for *SN*).

The well-formedness and semantics of a generic specification are essentially as for the imports, extended by the union of the parameter specifications, extended by the body:

{ *SP*$''_1$ **and** ... **and** *SP*$''_m$ **} then {** *SP*$_1$ **and** ... **and** *SP*$_n$ **} then** *SP*

The local environment given to the defined specification is empty, i.e., the above specification is implicitly closed. The difference between declaring parameters and leaving them implicit in an extension is that each parameter has to be provided with a fitting argument specification in all references to the specification name *SN*. The declared parameters show just which parts of the generic specification are *intended* to vary between different references to it. The imports, in contrast, are fixed, and common to the parameters, body, and arguments.

N.B. When a declared parameter happens to be merely a specification name, it always must refer to an *existing* specification definition in the global environment—it does *not* declare a local name for an argument specification.

```
SPEC-NAME ::= SIMPLE-ID
```

A specification name `SPEC-NAME` is normally displayed in a Small-Caps font, and input in mixed upper and lower case.

6.2.2 Specification Instantiation

```
SPEC      ::= ... | SPEC-INST
SPEC-INST ::= spec-inst SPEC-NAME FIT-ARG*
```

An instantiation `SPEC-INST` of a generic specification with some fitting argument specifications is written

$SN[FA_1]\ldots[FA_n]$

When the list of fitting arguments $FA_1, \ldots, FA_n$ is empty, the instantiation is merely a reference to the name of a specification that has no declared parameters at all, and it is simply written 'SN'. Note that the grouping braces '**{ }**', normally required when writing free (or closed) specifications, may always be omitted around instantiations.

The instantiation refers to the specification named SN in the global environment, providing a fitting argument FA_i for each declared parameter (in the same order).

```
FIT-ARG  ::= FIT-SPEC
FIT-SPEC ::= fit-spec SPEC SYMB-MAP-ITEMS*
```

A fitting argument specification `FIT-SPEC` is written:

SP'_i **fit** SM_i

When SM_i is empty, the fitting argument specification is simply written SP'_i. Symbol mappings SM are described in Sections 6.4 and 6.5.

The signature Σ_i given by the parameter specification SP_i, the signature Σ'_i given by the corresponding argument specification, and the symbol mapping SM determine a signature morphism from Σ_i to Σ'_i, as explained in Chapter 5. The fitting argument is well-formed only when the signature morphism is defined, i.e., the fitting argument morphism is well-defined. Note that mapping an operation or predicate symbol generally implies non-identity mapping of the sorts in the profile.

When there is more than one parameter, the separate fitting argument morphisms have to be ***compatible***, and their union has to yield a single morphism from the union of the parameters to the union of the arguments. Thus any common parts of declared parameters have to be instantiated in the same way, and it is pointless to declare the same parameter twice in a generic specification. (Generic specifications that require two similar but independent parameters can be expressed by using a translation to distinguish between the symbols in the signatures of the two parameters.)

Each fitting argument FA_i is regarded as an extension of the union of the imports (the current local environment is ignored). The fitting argument morphism has to be identity on all symbols declared by the imports SP''_1, ..., SP''_m of the generic specification, if there are any.

Let SP' be the extension of the imports by the generic parameters and then by the body of the specification named SN:

{ SP''_1 **and** ... **and** SP''_m **} then {** SP_1 **and** ... **and** SP_n **} then** SP

Let FM be the morphism yielded by the fitting arguments FA_1, ..., FA_n, extended to a morphism applicable to the signature of SP' as explained in Sections 6.4 and 6.5 (and written as a list of symbol maps). Then the semantics of the well-formed instantiation $SN[FA_1]...[FA_n]$ is the same as that of the specification:

{ SP' **with** FM **} and** SP'_1 **and** ... **and** SP'_n

where each SP'_i is the specification of the corresponding fitting argument FA_i. Each model of an argument FA_i (these are models of SP'_i reduced by the signature morphism determined by SM_i) is required to be a model of the corresponding parameter SP_i, otherwise the instantiation is undefined. The instantiation is not well-formed if the result signature is not a push-out of the body and argument signatures: if the translated body

{ SP' **with** FM **}**

and the union of the argument specifications

SP'_1 **and** ... **and** SP'_n

share any symbols, these symbols have to be translations (along FM) of symbols that share in the extension of the imports by the generic parameters

{ SP''_1 **and** ... **and** SP''_m **} then {** SP_1 **and** ... **and** SP_n **}**

Here, two sorts share if they are identical, and two function or predicate symbols share if they are in the overloading relation.

6.3 Views

Views between specifications are named by view definitions, and referenced by use of the name. A named view may also have some parameters, which have to be instantiated when referencing the view.

6.3.1 View Definitions

```
VIEW-DEFN ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE SYMB-MAP-ITEMS*
VIEW-TYPE ::= view-type SPEC SPEC
```

A view definition `VIEW-DEFN` with some generic parameters and some imports is written:

view VN $[SP_1]$ ... $[SP_n]$ **given** $SP''_1, \ldots, SP''_m$: SP **to** SP' =
SM
end

A view definition `VIEW-DEFN` with some generic parameters is written:

view VN $[SP_1]$ ... $[SP_n]$: SP **to** SP' =
SM
end

When the list of generic parameters is empty, the view definition is simply written:

view VN : SP **to** SP' =
SM
end

The terminating '**end**' keyword is optional.

It declares the view name VN to have the type of specification morphisms from SP to SP', parameter specifications $SP_1, \ldots, SP_n$ (if any), import specifications $SP''_1, \ldots, SP''_m$ (if any), and defines it by the symbol mapping SM. Symbol mappings SM are described in Sections 6.4 and 6.5.

SP gets the empty local environment. The well-formedness conditions for SP' are as if SP' were the body of a parameterized specification with formal parameters $SP_1, \ldots, SP_n$ and import specifications $SP''_1, \ldots, SP''_m$. The view definition is well-formed only if the signature morphism determined by the symbol mapping SM, as explained in Chapter 5, is defined. The view definition extends the global environment (which must not already include a definition for VN).

Generic parameters in a view definition allow the same view to be instantiated with different fitting arguments, giving compositions of the morphism defined by the view with other fitting morphisms. The source SP of the view is *not* in the scope of the view parameters $SP_1, \ldots, SP_n$, and view instantiation affects only the target of the generic view.

It is required that the reduct by the specification morphism of each model of the target

{ SP''_1 **and** ... **and** SP''_m **} then {** SP_1 **and** ... **and** SP_n **} then** SP'

is a model of the source SP; otherwise the semantics is undefined.

```
VIEW-NAME ::= SIMPLE-ID
```

A view name `VIEW-NAME` is normally displayed in a Small-Caps font, and input in mixed upper and lower case.

6.3.2 Fitting Views

```
FIT-ARG  ::= ... | FIT-VIEW
FIT-VIEW ::= fit-view VIEW-NAME
```

A reference to a non-generic fitting argument view `FIT-VIEW` is simply written:

view VN

It refers to the current global environment, and is well-formed as an argument for a parameter SP_i only when the global environment includes a view definition for VN of type from SP to SP', such that the signatures of SP and of SP_i are the same. The view definition then provides the fitting morphism from the parameter SP_i to the argument specification given by the target SP' of the view.

If the generic specification being instantiated has imports, the fitting morphism is then the union of the specification morphism given by the view and the identity morphism on the imports. The argument specification is the union of the target of the view and the imports.

Each model of SP is required to be a model of SP_i, otherwise the instantiation is undefined.

```
FIT-VIEW ::= ... | fit-view VIEW-NAME FIT-ARG+
```

A fitting argument view `FIT-VIEW` involving the instantiation of a generic view to fitting arguments is written:

view VN $[FA_1] \ldots [FA_n]$

It refers to the current global environment, and is well-formed only when the global environment includes a generic view definition for VN with parameters that can be instantiated by the indicated fitting arguments FA_1, ..., FA_n to give a view of type from SP to SP', such that the signatures of SP and of SP_i are the same. As with non-generic views, each model of

SP is required to be a model of SP_i, otherwise the instantiation is undefined. The instantiation of a generic view with some fitting arguments is not well-formed if the instantiation of the target SP' of the view with the same fitting arguments is not well-formed.

6.4 Symbol Lists and Mappings

6.4.1 Symbol Lists

Symbol lists are used in hiding reductions.

```
SYMB-ITEMS ::= symb-items SYMB-KIND SYMB+
SYMB-KIND  ::= implicit | sorts-kind | ops-kind | preds-kind
SYMB       ::= ID | QUAL-ID
QUAL-ID    ::= qual-id ID TYPE
TYPE       ::= OP-TYPE | PRED-TYPE
```

A list of symbols `SYMB-ITEMS` with implicit kinds `SYMB-KIND` is written simply:

$SY_1, \ldots, SY_n$

Overloaded operation symbols and predicate symbols may be disambiguated by explicit qualification; when SY_i is not qualified, the effect is as if all (sort, operation, or predicate) symbols declared with the same name in the current local environment are listed.

Optionally, the list may be sectioned into sub-lists by inserting the keywords '**sorts**', '**ops**', '**preds**' (or their singular forms), which explicitly indicate that the subsequent symbols are of the corresponding kind:

sorts $s_1, \ldots,$ **ops** $f_1, \ldots,$ **preds** $p_1, \ldots$

As with signature declarations in basic specifications, there is no restriction on the order of the various sections of the list.

A single sort occurring as a type in a qualified identifier `QUAL-ID` is interpreted as a constant operation type or unary predicate type, as determined by the latest keyword, or, when there is none, unambiguously by the local environment.

The list determines a set of qualified symbols, obtained from the listed symbols with reference to a given signature; the order in which symbols are listed is not significant (except regarding their position in relation to any specified kinds).

6.4.2 Symbol Mappings

Symbol mappings are used in translations, revealing reductions, fitting arguments, and views.

```
SYMB-MAP-ITEMS ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-OR-MAP    ::= SYMB | SYMB-MAP
SYMB-MAP       ::= symb-map SYMB SYMB
```

A list of symbol maps `SYMB-MAP-ITEMS` with implicit kinds `SYMB-KIND` is written simply:

$SY_1 \mapsto SY_1', \ldots, SY_n \mapsto SY_n'$

The sign displayed as '$\mapsto$' is input as '`|->`'.

$SY_i \mapsto SY_i'$ denotes the map that takes the symbol SY_i to the symbol SY_i'. The mapped symbols in the list must be distinct. Overloaded operation symbols and predicate symbols may be disambiguated by explicit qualification; when SY_i is not qualified, the effect is as if all (sort, operation, or predicate) symbols declared with the same name in the current environment are mapped uniformly to SY_i'.

Optionally, the list may be sectioned into sub-lists by inserting the keywords '**sorts**', '**ops**', '**preds**' (or their singular forms), which explicitly indicate that the subsequent symbols are of the corresponding kind:

sorts $s_1 \mapsto s_1', \ldots$, **ops** $f_1 \mapsto f_1', \ldots$, **preds** $p_1 \mapsto p_1', \ldots$

As with signature declarations in basic specifications, there is no restriction on the order of the various sections of the list.

An identity map '$SY_i \mapsto SY_i$' may be simply written 'SY_i'. Thus a symbol list may be regarded as a special case of a symbol mapping.

The list determines a set of qualified symbols, obtained from the first components of the listed symbol maps with reference to a given signature, together with a mapping of these symbols to qualified symbols obtained from the second components of the listed symbol maps. The order in which symbol maps are listed is not significant (except regarding their position in relation to any specified kinds).

6.5 Compound Identifiers

```
TOKEN-ID       ::= ... | COMP-TOKEN-ID
MIXFIX-ID      ::= ... | COMP-MIXFIX-ID
COMP-TOKEN-ID  ::= comp-token-id TOKEN ID+
COMP-MIXFIX-ID ::= comp-mixfix-id TOKEN-PLACES ID+
```

This extension of the syntax of identifiers for sorts, operations, and predicates is of relevance to generic specifications. An ordinary ***compound identifier*** `COMP-TOKEN-ID` is written '$I_[I_1, \ldots, I_n]$'; a mixfix compound identifier `COMP-MIXFIX-ID` is written by inserting '$[I_1, \ldots, I_n]$' directly after the last token of the identifier. (Compound 'invisible' identifiers without any tokens are not allowed.) Note that declaration of both compound identifiers and mixfix identifiers as operation symbols in the same local environment may give rise to ambiguity, when they involve overlapping sets of tokens.

The components I_i may (but need not) themselves identify sorts, operations, or predicates that are specified in the declared parameters of a generic specification.

When such a compound identifier is used to name, e.g., a sort in the body of a generic specification, the translation determined by fitting arguments to parameters applies to the components I_1,...,I_n as well. Thus instantiations with different arguments generally give rise to different compound identifiers for what would otherwise be the same sort, which avoids unintended identifications when the instantiations are united.

E.g., a generic specification of sequences of arbitrary elements might use the simple identifier $Elem$ for a sort in the parameter, and a compound identifier $Seq[Elem]$ for the sort of sequences in the body. Fitting various argument sorts to $Elem$ in different instantiations then results in distinct sorts of sequences.

Subsort embeddings between component sorts do *not* induce subsort embeddings between the compound sorts: when desired, these have to be declared explicitly. For example, when Nat is declared as a subsort of Int, we do *not* automatically get $Seq[Nat]$ embedded as a subsort of $Seq[Int]$ in signatures containing all these sorts.

Instantiation, however, does preserve subsorts: if in a generic specification we have $Elem$ declared as a subsort of $Seq[Elem]$, where $Elem$ is a parameter sort, then in the result of instantiation of $Elem$ by Nat, one does get Nat automatically declared as a subsort of $Seq[Nat]$. Compound identifiers must not be identified through the identification of components by the fitting morphism. E.g., if the body of a generic specification contains both $List[Elem1]$ and $List[Elem2]$, the fitting morphism must not map both $Elem1$ and $Elem2$ to Nat.

Higher-order extensions of CASL are expected to provide a more semantic treatment of parametrized sorts, etc.

Part III

Architectural Specifications

Chapter 7

Architectural Concepts

The intention with architectural specifications is primarily to impose structure on models, expressing their ***composition*** from component units—and thereby also a ***decomposition*** of the task of developing such models from requirements specifications. This is in contrast to the structured specifications summarized in Part II, where the specified models have no more structure than do those of the basic specifications summarized in Part I.

The component units may all be regarded as ***unit functions***: functions without arguments give self-contained units; functions with arguments use such units in constructing further units. Note that a resulting unit may be needed for use as an argument in more than one application.

The specification of a unit function indicates the properties to be assumed of the arguments, and the properties to be guaranteed of the result. Such a specification provides the appropriate interfaces for the development of the function. In CASL, self-contained units are simply models as defined in Part I, and their properties are expressed by ordinary (perhaps structured) specifications.

Thus a unit function maps models of argument specifications to models of a result specification. A specification of such functions can be simply a list of the argument specifications together with the result specification. Thinking of argument and result specifications as *types* of models, a specification of a unit function may be regarded as a *function type*.

An entire ***architectural specification*** is a collection of unit function specifications, together with a description of how the functions are to be composed to give a resulting unit. A model of an architectural specification is a collection of unit functions with the specified types or definitions, together with the result of composing them as described.

The intention is that a unit function should actually make use of its arguments. In particular, it should not re-implement the argument specifications. This is ensured by requiring the unit function to be *persistent*: the reduct of the result to each argument signature yields exactly the given arguments.

As a consequence, the result *signature* has to include each argument *signature*—any desired hiding has to be left to when functions are composed. Moreover, since each *symbol* in the union of the argument signatures has to be implemented the same way in the result as in each argument where it occurs, the arguments must already have the same implementation of all common symbols. In the absence of subsorts, this is sufficient to allow one to unambiguously *amalgamate* arguments into a single model over the union of argument signatures. When subsorts are present, extra conditions to ensure that implicit subsort embeddings can be defined unambiguously in such an amalgamated model may be necessary. Let us call arguments satisfying such a requirement ***compatible***.

Hence the interpretation of the specification of a unit function is as all *persistent* functions from *compatible* tuples of models of the argument specifications to models of the result specification. When composing such functions, care must be taken to ensure that arguments are indeed compatible. Notice that if two arguments have the same signature, the arguments must be identical. It is not possible to specify a function that should take two arguments that implement the same signature independently—although one can get the same effect, by renaming one or both of the argument signatures.

Chapter 8

Architectural Constructs

This chapter indicates the abstract and concrete syntax of the constructs of *architectural* specifications, and describes their intended interpretation, extending what was provided for basic and structured specifications in Parts I and II.

```
ARCH-SPEC-DEFN ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC      ::= BASIC-ARCH-SPEC | ARCH-SPEC-NAME
```

An architectural specification definition `ARCH-SPEC-DEFN` is written:

arch spec *ASN* =
    ASP
end

where the terminating '**end**' keyword is optional.

It defines the name *ASN* to refer to the architectural specification *ASP*, extending the global environment (which must not already include a definition for *ASN*). The local environment given to *ASP* is empty.

```
ARCH-SPEC-NAME ::= SIMPLE-ID
```

An architectural specification name `ARCH-SPEC-NAME` is normally displayed in a Small-Caps font, and input in mixed upper and lower case.

A reference in an architectural specification `ARCH-SPEC` to an architectural specification named *ASN* is simply written as the name itself '*ASN*'. It refers to the the current global environment, and is well-formed only when the global environment includes an architectural specification definition for *ASN*. The enclosing definition then merely introduces a synonym for a previously-defined architectural specification.

```
BASIC-ARCH-SPEC ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT
UNIT-DECL-DEFN  ::= UNIT-DECL | UNIT-DEFN
RESULT-UNIT     ::= result-unit UNIT-EXPRESSION
```

A basic architectural specification `BASIC-ARCH-SPEC` is written:

units UD_1; ... UD_n; **result** UE;

where both the last two semicolons are optional.

It consists of a list of unit declarations and definitions UD_1, ..., UD_n, together with a unit expression UE describing how such units are to be composed. A model of such an architectural specification consists of a unit for each UD_i, and the composition of these units as described by UE.

8.1 Unit Declarations and Definitions

The visibility of unit names in architectural specifications is linear: each name has to be declared or defined before it is used in a unit expression; and no unit name may be introduced more than once in a particular architectural specification. Note that declarations and definitions of units do not affect the global environment: a unit may be referenced only within the architectural specification in which it occurs.

8.1.1 Unit Declarations

```
UNIT-DECL     ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
UNIT-IMPORTED ::= unit-imported UNIT-TERM*
UNIT-NAME     ::= SIMPLE-ID
```

A unit declaration `UNIT-DECL` is written:

UN : USP **given** UT_1,...,UT_n

When the list `UNIT-TERM*` of unit terms is empty, it is simply written:

UN : USP

It provides not only a unit specification USP but also a unit name UN, which is used for referring to the unit in subsequent unit expressions, so that the same unit may be used more than once in a composition.

In addition, the `UNIT-IMPORTED` lists any units UT_1, ...,UT_n that are imported for the implementation of the declared unit (corresponding to implementing a generic unit function and applying it only once, to the imported units, the argument type of the generic function being merely the list of the

signatures of the UT_i). The unit specification *USP* is treated as an extension of the signatures of the imported units, thus being given a non-empty local environment, in general.

8.1.2 Unit Definitions

```
UNIT-DEFN ::= unit-defn UNIT-NAME UNIT-EXPRESSION
```

A unit definition `UNIT-DEFN` is written:

UN = *UE*

It defines the name *UN* to refer to the unit resulting from the composition described by the unit expression *UE*.

8.2 Unit Specifications

```
UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | ARCH-UNIT-SPEC
                   | CLOSED-UNIT-SPEC
```

A unit specification definition `UNIT-SPEC-DEFN` is written:

unit spec *SN* =
    USP
end

where the terminating '**end**' keyword is optional.

It provides a name *SN* for a unit specification *USP*. The unit specification may be a unit type. It may also be the name of another unit specification (in the context-free concrete syntax, this is indistinguishable from a reference to a named structured specification in a constant unit type, but the global environment determines how the name should be interpreted). It may be an architectural specification (either a reference to the defined name of an architectural specification, or an anonymous architectural specification). Finally, it may be an explicitly-closed unit specification.

It defines the name *SN* to refer to the unit specification *USP*, extending the global environment (which must not already include a definition for *SN*). The local environment given to *USP* is empty, i.e., the unit specification is implicitly closed.

8.2.1 Unit Types

```
UNIT-TYPE ::= unit-type SPEC* SPEC
```

A unit type is written:

$$SP_1 \times \ldots \times SP_n \rightarrow SP$$

When the list `SPEC*` of argument specifications is empty, the unit type is simply written '*SP*'.

A unit satisfies a unit type when it is a persistent function that maps compatible tuples of models of the argument specifications SP_1, ..., SP_n to models of their extension by the result specification *SP*.

8.2.2 Architectural Unit Specifications

```
ARCH-UNIT-SPEC   ::= arch-unit-spec ARCH-SPEC
```

An architectural unit specification `ARCH-UNIT-SPEC` is written:

arch spec *ASP*

A unit satisfies '**arch spec** *ASP*' when it is the result unit of some model of *ASP*.

8.2.3 Closed Unit Specifications

```
CLOSED-UNIT-SPEC ::= closed-unit-spec UNIT-SPEC
```

A closed unit specification `CLOSED-UNIT-SPEC` is written:

closed *USP*

It determines the same type as *USP* determines in the empty local environment, thus ensuring the closedness of *USP*.

8.3 Unit Expressions

```
UNIT-EXPRESSION ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING    ::= unit-binding UNIT-NAME UNIT-SPEC
```

A unit expression with some unit bindings is written:

$$\lambda UN_1 : USP_1;\ \ldots;\ UN_n : USP_n \bullet UT$$

The sign displayed as 'λ' is input as '`lambda`'. The sign displayed as ' • ' may be input as '·' in ISO Latin-1, or as '`.`' in ASCII. When the list of unit bindings is empty, just the unit term 'UT' is written.

It describes a composition of units declared (or defined) in the enclosing architectural specification. The result unit is a function, mapping the arguments specified by the unit bindings (if any) to the unit described by the unit term UT. The unit names UN_1, ..., UN_n for the arguments must be distinct, and not include the names of units previously declared in the enclosing architectural specification.

The unit bindings for the arguments (which are like unit declarations but with no possibility of importing other units) in a unit expression are for (non-parameterized) units that are required to build the result, but are not directly provided yet. This allows for compositions which express partial architectural specifications that depend on additional units, and might be used to instantiate the same composition for various realizations of the required units.

8.3.1 Unit Terms

```
UNIT-TERM ::= UNIT-REDUCTION | UNIT-TRANSLATION | AMALGAMATION
            | LOCAL-UNIT | UNIT-APPL
```

Unit terms provide counterparts to most of the constructs of structured specifications: translations, reductions, amalgamations (corresponding to unions), local unit definitions, and applications (corresponding to instantiations).

Unit terms use the same notation as structured specifications—but with a crucially different semantics, however. This is easiest to notice when considering the difference between union and amalgamation as well as between translation and unit translation. For units, enough sharing is required so that the constructs as applied to the given units will always make sense and produce results. This is in contrast with the constructs for structured specifications, where well-formed unions or (non-injective) translations of consistent specifications might result in inconsistencies.

The sharing between symbols is understood here semantically: two symbols share if they coincide semantically. However, there is also a static semantics (with the corresponding static analysis supported by CASL tools) that exploits situations where symbols required to share in fact originate from the same symbol in some unit declaration or definition. Such direct information should be sufficient to discharge the verification conditions implicit in the above semantic requirement in most typical cases. This is simplest when

no subsorting constructs are involved. The presence of subsorts, and the properties that subsort embeddings and overloaded operations and predicates must satisfy, make the static analysis more complex [SMT+01] (but still tractable in practical examples).

Taking the unit type of each unit name from its declaration, the unit term must be well-typed. All the constructs involved must get argument units over the appropriate signatures.

8.3.1.1 Unit Translations

```
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
```

A unit translation is written:

$UT\ R$

where the renaming R is written '**with** SM', and determines a mapping of symbols, cf. Section 6.1.1.

It allows some of the unit symbols to be renamed. Any symbols that happen to be glued together by the renaming must share.

8.3.1.2 Unit Reductions

```
UNIT-REDUCTION ::= unit-reduction UNIT-TERM RESTRICTION
```

A unit-reduction is written:

$UT\ R$

where the restriction R is written '**hide** SL' or '**reveal** SM', and determines a set of symbols, and in the latter case also a mapping of them, cf. Section 6.1.2.

It allows parts of the unit to be hidden and other parts to be simultaneously renamed.

8.3.1.3 Amalgamations

```
AMALGAMATION ::= amalgamation UNIT-TERM+
```

An amalgamation is written:

UT_1 **and** ... **and** UT_n

It produces a unit that consists of the components of all the amalgamated units put together. Compatibility of the unit terms must be ensured.

8.3.1.4 Local Units

```
LOCAL-UNIT ::= local-unit UNIT-DEFN+ UNIT-TERM
```

A local unit is written:

local UD_1; ...; UD_n; **within** UT

where the final '**;**' may be omitted.

This allows for naming units that are locally defined for use in a unit term, these units being intermediate results that are not to be visible in the models of the enclosing architectural specification.

8.3.1.5 Unit Applications

```
UNIT-APPL ::= unit-appl UNIT-NAME FIT-ARG-UNIT*
```

A unit application `UNIT-APPL` is written:

$UN[FAU_1]...[FAU_n]$

It refers to a generic unit named UN that has already been declared or defined in the enclosing architectural specification, providing a fitting argument FAU_i for each declared parameter (in the same order).

```
FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*
```

A fitting argument FAU_i is written:

UT'_i **fit** SM_i

When the symbol mapping SM_i is empty, just the unit term UT'_i is written.

The fitting argument fits the argument unit given by the unit term UT'_i to the corresponding formal argument for the generic unit via a signature morphism determined by the symbol mapping SM_i. The signature morphism is obtained in the same way as for generic specifications. Unmapped symbols are included unchanged. Of course, the signature of the actual argument might coincide with the corresponding signature in the generic unit type, in which case no extra fitting is needed, and the argument unit is passed to the generic unit directly. The compatibility of the arguments must be ensured.

Each fitting argument unit FAU_i is required to be a model of the corresponding argument specification, otherwise the unit application is undefined.

Part IV

Specification Libraries

Chapter 9

Library Concepts

Specifications may be ***named*** by ***definitions*** and collected in ***libraries***. In the context of a library, the (re)use of a specification may be replaced by a ***reference*** to it through its name. The current association between names and the specifications that they reference is called the ***global environment***; it may vary throughout a library: with ***linear visibility***, as in CASL, the global environment for a named specification is determined exclusively by the definitions that precede it. When overriding is forbidden, as in CASL, each valid reference to a particular name refers to the same defined entity.

The local environment given to each named specification in a library should be independent of the other specifications in the library (in CASL, it is empty). Thus any dependence between the specifications is always apparent from the explicit references to the names of specifications.

A library may be located at a particular ***site*** on the Internet. The library is referenced from other sites by a name which determines the location and perhaps identifies a particular version of the library. To allow libraries to be relocated without this invalidating existing references to them, library names may be interpreted relative to a ***global directory*** that maps names to URLs. Libraries may also be referenced directly by their (relative or absolute) URLs, independently of their registration in the global directory.

A library may incorporate the ***downloading*** of copies of named specifications from (perhaps particular versions of) other libraries, whenever the library is used. To ensure continuous access to specifications despite temporary failures at a particular library site, registered libraries may be mirrored at archive sites.

The semantics of a specification library is the name of the library together with a map taking each specification name defined in it to the semantics of that specification. The initial global environment for the library is empty.

Chapter 10

Library Constructs

This chapter indicates the abstract and concrete syntax of the constructs of *specification libraries*, and describes their intended interpretation, extending what was provided for basic, structured, and architectural specifications in Parts I–III.

First, the constructs of *local* libraries are presented. Such libraries are not dependent on other libraries. Then constructs for referencing *distributed* libraries are added. Finally, the form and intended interpretation of library names are explained.

10.1 Local Libraries

```
LIB-DEFN ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM ::= SPEC-DEFN | VIEW-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
```

A library definition `LIB-DEFN` is written:

library LN $LI_1 \ldots LI_n$

Each library item LI_i starts with a distinctive keyword, and may be terminated by an optional '**end**'.

The library definition provides a collection of specification (and perhaps also view) definitions. It is well-formed only when the defined names are distinct, and not referenced until (strictly) after their definitions. The global environment for each definition is that determined by the preceding definitions. Thus a library in CASL provides linear visibility, and mutual or cyclic chains of references are not allowed.

The local environment for each definition is empty: the symbols declared by the preceding specifications in the library are only made available by explicit reference to the name of the specification concerned.

Each specification definition in a library must be self-contained (after resolving references to names defined in the current global environment), determining a complete signature—fragments of specifications cannot be named.

A local library definition determines a library name, together with a map from names to the semantics of the named specifications.

10.2 Distributed Libraries

```
LIB-ITEM          ::= ... | DOWNLOAD-ITEMS
DOWNLOAD-ITEMS    ::= download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP ::= ITEM-NAME | ITEM-NAME-MAP
ITEM-NAME-MAP     ::= item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME         ::= SIMPLE-ID
```

The syntax of local libraries is here extended with a further sort of library item, for use with distributed libraries. The `DOWNLOAD-ITEMS` construct is written:

from LN **get** $IN_1 \mapsto IN'_1, \ldots, IN_n \mapsto IN'_n$ **end**

where the terminating '**end**' keyword is optional. An identity map '$IN_n \mapsto IN_n$' may be simply written 'IN_n'.

It specifies which definitions to copy from the remote library named LN, changing the remote names IN_i to the local names IN'_i.

The semantics corresponds to having already replaced all references in the downloaded definitions by the corresponding (closed) specifications; cyclic chains of references via remote libraries are not allowed. The download items show exactly which specification names are added to the current global environment of the library in which they occur, allowing references to named specifications to be checked locally (although not whether the kind of specification to be downloaded from the remote library is consistent with its local use).

10.3 Library Names

```
LIB-NAME       ::= LIB-ID | LIB-VERSION
LIB-VERSION    ::= lib-version LIB-ID VERSION-NUMBER
VERSION-NUMBER ::= version-number NUMBER+
```

A library name `LIB-NAME` without a `VERSION-NUMBER` is written simply as a library identifier *LI*. A library name `LIB-NAME` with version numbers N_1, ..., N_n is written:

LI **version** $N_1. \ldots .N_n$

The lists of version numbers are ordered lexicographically on the basis of the usual ordering between natural numbers.

The library name of a library definition determines how the library is to be referenced from other libraries; its interpretation as a URL determines the primary location of the library (any copies of a library are to retain the original name).

When the name of a defined library is simply a library identifier `LIB-ID`, it must be changed to an explicit library version `LIB-VERSION` before defining further versions of that library. A library identifier without an explicit version in a downloading construct always refers to the current version of the identified library: the one with the largest list of version numbers (which is not necessarily the last-created version, due to the lexicographic ordering on such lists).

```
LIB-ID        ::= DIRECT-LINK | INDIRECT-LINK
DIRECT-LINK   ::= direct-link URL
INDIRECT-LINK ::= indirect-link PATH
```

A direct link to a library is simply written as the URL of the library. The location of a library is always a directory, giving access not only to the individual specifications defined by the current version of the library but also to previously-defined versions, various indexes, and perhaps other documentation.

An indirect link is written:

$FI_1/\ldots/FI_n$

where each file identifier FI_i is a valid file name, as for use in a path in a URL. An indirect link is interpreted as a URL by the current global library directory.

Bibliography

[BST98] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *AMAST '98, Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology, Manaus*, volume 1548 of *LNCS*, pages 341–357. Springer-Verlag, 1998.

[BST00] Michel Bidoit, Don Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. Note M-4 (submitted for publication), in [CoF], November 2000.

[CoF] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI/`.

[CoF96] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language (Tentative Design, version 0.95) – Language Summary. Documents/Tentative/LanguageSummary, in [CoF], December 1996.

[CoF97a] CoFI Language Design Task Group. Response to the Referee Report on CASL. Documents/CASL/RefereeResponse (initial response, referring to CASL version 0.97), in [CoF], August 1997.

[CoF97b] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary, version 0.97. Documents/CASL/Summary-v0.97, in [CoF], May 1997.

[CoF97c] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language (Tentative Design, version 0.95) – Language Summary, with annotations concerning questions and doubts. Note S-1 (revised), in [CoF], April 1997.

[CoF97d] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language (Tentative Design, version 0.95) – Language Summary, with annotations concerning the semantics of constructs. Note S-4, in [CoF], April 1997.

[CoF97e] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language (version 0.97) – Semantics. Note S-6, in [CoF], July 1997.

[CoF99] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Documents/CASL/Semantics (version 0.96), in [CoF], July 1999.

[CoF00] CoFI Language Design Task Group. Response to the Referee Report on CASL. Documents/CASL/RefereeResponse (final response), in [CoF], March 2000.

[GB92] Joseph A. Goguen and Rodney M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[IFI97] IFIP WG 1.3. Referee Report on CASL. Documents/CASL/RefereeReport (referring to CASL version 0.97), in [CoF], June 1997.

[Mos97] Peter D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97, Proc. Intl. Symp. on Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 115–137. Springer-Verlag, 1997.

[Mos98] Peter D. Mosses. Formatting CASL specifications using LaTeX. Note C-2, in [CoF], June 1998.

[RM00] Markus Roggenbach and Till Mossakowski. Basic datatypes in CASL. Note L-12, version 0.4.1, in [CoF], May 2000.

[SMT+01] Lutz Schröder, Till Mossakowski, Andrzej Tarlecki, Piotr Hoffman, and Bartosz Klin. Semantics of architectural specifications in CASL. In *Fundamental Approaches to Software Engineering (FASE2001)*, LNCS. Springer, 2001. To appear.

Index

Appendices

Appendix A

Abstract Syntax

The *abstract syntax* is central to the definition of a formal language. It stands between the concrete representations of documents, such as marks on paper or images on screens, and the abstract entities, semantic relations, and semantic functions used for defining their meaning.

The abstract syntax has the following objectives:

- to identify and separately name the abstract syntactic entities;
- to simplify and unify underlying concepts, putting like things with like, and reducing unnecessary duplication.

There are many possible ways of constructing an abstract syntax, and the choice of form is a matter of judgement, taking into account the somewhat conflicting aims of simplicity and economy of semantic definition.

The abstract syntax is presented as a set of production rules in which each sort of entity is defined in terms of its subsorts:

```
SOME-SORT       ::=  SUBSORT-1  |  ...  |  SUBSORT-n
```

or in terms of its constructor and components:

```
SOME-CONSTRUCT  ::=  some-construct COMPONENT-1 ... COMPONENT-n
```

The productions form a context-free grammar; algebraically, the nonterminal symbols of the grammar correspond to sorts (of trees), and the terminal symbols correspond to constructor operations. The notation `COMPONENT*` indicates repetition of `COMPONENT` any number of times; `COMPONENT+` indicates repetition at least once. (These repetitions could be replaced by auxiliary sorts and constructs, after which it would be straightforward to transform the grammar into a CASL `FREE-DATATYPE` specification.)

The context conditions for well-formedness of specifications are not determined by the grammar (these are considered as part of semantics).

The grammar here has the property that there is a sort for each construct (although an exception is made for constant constructs with no components). Appendix B provides an abbreviated grammar defining the same abstract syntax. It was obtained by eliminating each sort that corresponds to a single construct, when this sort occurs only once as a subsort of another sort.

The following nonterminal symbols correspond to lexical syntax, and are left unspecified in the abstract syntax: `WORDS`, `DOT-WORDS`, `SIGNS`, `DIGIT`, `DIGITS`, `NUMBER`, `QUOTED-CHAR`, `PLACE`, `URL`, and `PATH`.

A.1 Basic Specifications

```
BASIC-SPEC       ::= basic-spec BASIC-ITEMS*

BASIC-ITEMS      ::= SIG-ITEMS | FREE-DATATYPE | SORT-GEN
                   | VAR-ITEMS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS

SIG-ITEMS        ::= SORT-ITEMS | OP-ITEMS | PRED-ITEMS
                   | DATATYPE-ITEMS

SORT-ITEMS       ::= sort-items SORT-ITEM+
SORT-ITEM        ::= SORT-DECL

SORT-DECL        ::= sort-decl SORT+

OP-ITEMS         ::= op-items OP-ITEM+
OP-ITEM          ::= OP-DECL | OP-DEFN

OP-DECL          ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
OP-TYPE          ::= TOTAL-OP-TYPE | PARTIAL-OP-TYPE
TOTAL-OP-TYPE    ::= total-op-type SORT-LIST SORT
PARTIAL-OP-TYPE  ::= partial-op-type SORT-LIST SORT
SORT-LIST        ::= sort-list SORT*
OP-ATTR          ::= BINARY-OP-ATTR | UNIT-OP-ATTR
BINARY-OP-ATTR   ::= assoc-op-attr | comm-op-attr | idem-op-attr
UNIT-OP-ATTR     ::= unit-op-attr TERM

OP-DEFN          ::= op-defn OP-NAME OP-HEAD TERM
OP-HEAD          ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD    ::= total-op-head ARG-DECL* SORT
PARTIAL-OP-HEAD  ::= partial-op-head ARG-DECL* SORT
ARG-DECL         ::= arg-decl VAR+ SORT

PRED-ITEMS       ::= pred-items PRED-ITEM+
PRED-ITEM        ::= PRED-DECL | PRED-DEFN

PRED-DECL        ::= pred-decl PRED-NAME+ PRED-TYPE
PRED-TYPE        ::= pred-type SORT-LIST
```

```
PRED-DEFN         ::= pred-defn PRED-NAME PRED-HEAD FORMULA
PRED-HEAD         ::= pred-head ARG-DECL*

DATATYPE-ITEMS    ::= datatype-items DATATYPE-DECL+
DATATYPE-DECL     ::= datatype-decl SORT ALTERNATIVE+
ALTERNATIVE       ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT
TOTAL-CONSTRUCT   ::= total-construct   OP-NAME COMPONENTS*
PARTIAL-CONSTRUCT::= partial-construct OP-NAME COMPONENTS+
COMPONENTS        ::= TOTAL-SELECT | PARTIAL-SELECT | SORT
TOTAL-SELECT      ::= total-select   OP-NAME+ SORT
PARTIAL-SELECT    ::= partial-select OP-NAME+ SORT

FREE-DATATYPE     ::= free-datatype DATATYPE-ITEMS

SORT-GEN          ::= sort-gen SIG-ITEMS+

VAR-ITEMS         ::= var-items VAR-DECL+
VAR-DECL          ::= var-decl VAR+ SORT

LOCAL-VAR-AXIOMS ::= local-var-axioms VAR-DECL+ AXIOM+

AXIOM-ITEMS       ::= axiom-items AXIOM+

AXIOM             ::= FORMULA
FORMULA           ::= QUANTIFICATION | CONJUNCTION | DISJUNCTION
                    | IMPLICATION | EQUIVALENCE | NEGATION | ATOM
QUANTIFICATION    ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER        ::= universal | existential | unique-existential
CONJUNCTION       ::= conjunction FORMULA+
DISJUNCTION       ::= disjunction FORMULA+
IMPLICATION       ::= implication FORMULA FORMULA
EQUIVALENCE       ::= equivalence FORMULA FORMULA
NEGATION          ::= negation FORMULA

ATOM              ::= TRUTH | PREDICATION | DEFINEDNESS
                    | EXISTL-EQUATION | STRONG-EQUATION
TRUTH             ::= true-atom | false-atom
PREDICATION       ::= predication PRED-SYMB TERMS
PRED-SYMB         ::= PRED-NAME | QUAL-PRED-NAME
QUAL-PRED-NAME    ::= qual-pred-name PRED-NAME PRED-TYPE
DEFINEDNESS       ::= definedness TERM
EXISTL-EQUATION   ::= existl-equation TERM TERM
STRONG-EQUATION   ::= strong-equation TERM TERM

TERMS             ::= terms TERM*
TERM              ::= SIMPLE-ID | QUAL-VAR | APPLICATION
                    | SORTED-TERM | CONDITIONAL
QUAL-VAR          ::= qual-var VAR SORT
APPLICATION       ::= application OP-SYMB TERMS
OP-SYMB           ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME      ::= qual-op-name OP-NAME OP-TYPE
SORTED-TERM       ::= sorted-term TERM SORT
CONDITIONAL       ::= conditional TERM FORMULA TERM
```

```
SORT              ::= TOKEN-ID
OP-NAME           ::= ID
PRED-NAME         ::= ID
VAR               ::= SIMPLE-ID

SIMPLE-ID         ::= WORDS
ID                ::= TOKEN-ID | MIXFIX-ID
TOKEN-ID          ::= TOKEN
TOKEN             ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
MIXFIX-ID         ::= TOKEN-PLACES
TOKEN-PLACES      ::= token-places TOKEN-OR-PLACE+
TOKEN-OR-PLACE    ::= TOKEN | PLACE
```

A.2 Basic Specifications with Subsorts

```
SORT-ITEM         ::= ... | SUBSORT-DECL | ISO-DECL | SUBSORT-DEFN

SUBSORT-DECL      ::= subsort-decl SORT+ SORT
ISO-DECL          ::= iso-decl SORT+
SUBSORT-DEFN      ::= subsort-defn SORT VAR SORT FORMULA

ALTERNATIVE       ::= ... | SUBSORTS
SUBSORTS          ::= subsorts SORT+

ATOM              ::= ... | MEMBERSHIP
MEMBERSHIP        ::= membership TERM SORT

TERM              ::= ... | CAST
CAST              ::= cast TERM SORT
```

A.3 Structured Specifications

```
SPEC              ::= BASIC-SPEC | TRANSLATION | REDUCTION
                    | UNION | EXTENSION | FREE-SPEC | LOCAL-SPEC
                    | CLOSED-SPEC | SPEC-INST

TRANSLATION       ::= translation SPEC RENAMING
RENAMING          ::= renaming  SYMB-MAP-ITEMS+

REDUCTION         ::= reduction SPEC RESTRICTION
RESTRICTION       ::= HIDDEN | REVEALED
HIDDEN            ::= hidden   SYMB-ITEMS+
REVEALED          ::= revealed SYMB-MAP-ITEMS+

UNION             ::= union SPEC+
EXTENSION         ::= extension SPEC+
FREE-SPEC         ::= free-spec SPEC
LOCAL-SPEC        ::= local-spec SPEC SPEC
CLOSED-SPEC       ::= closed-spec SPEC

SPEC-DEFN         ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY        ::= genericity PARAMS IMPORTED
PARAMS            ::= params SPEC*
IMPORTED          ::= imported SPEC*

SPEC-INST         ::= spec-inst SPEC-NAME FIT-ARG*

FIT-ARG           ::= FIT-SPEC | FIT-VIEW
FIT-SPEC          ::= fit-spec SPEC SYMB-MAP-ITEMS*
FIT-VIEW          ::= fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN         ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE
                                SYMB-MAP-ITEMS*
VIEW-TYPE         ::= view-type SPEC SPEC

SYMB-ITEMS        ::= symb-items     SYMB-KIND SYMB+
SYMB-MAP-ITEMS    ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-KIND         ::= implicit | sorts-kind | ops-kind | preds-kind

SYMB              ::= ID | QUAL-ID
QUAL-ID           ::= qual-id ID TYPE
TYPE              ::= OP-TYPE | PRED-TYPE
SYMB-MAP          ::= symb-map SYMB SYMB
SYMB-OR-MAP       ::= SYMB | SYMB-MAP

SPEC-NAME         ::= SIMPLE-ID
VIEW-NAME         ::= SIMPLE-ID

TOKEN-ID          ::= ... | COMP-TOKEN-ID
MIXFIX-ID         ::= ... | COMP-MIXFIX-ID
COMP-TOKEN-ID     ::= comp-token-id TOKEN ID+
COMP-MIXFIX-ID    ::= comp-mixfix-id TOKEN-PLACES ID+
```

A.4 Architectural Specifications

```
ARCH-SPEC-DEFN   ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC        ::= BASIC-ARCH-SPEC | ARCH-SPEC-NAME
BASIC-ARCH-SPEC  ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT

UNIT-DECL-DEFN   ::= UNIT-DECL | UNIT-DEFN
UNIT-DECL        ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
UNIT-IMPORTED    ::= unit-imported UNIT-TERM*
UNIT-DEFN        ::= unit-defn UNIT-NAME UNIT-EXPRESSION

UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | ARCH-UNIT-SPEC
                   | CLOSED-UNIT-SPEC
ARCH-UNIT-SPEC   ::= arch-unit-spec ARCH-SPEC
CLOSED-UNIT-SPEC ::= closed-unit-spec UNIT-SPEC
UNIT-TYPE        ::= unit-type SPEC* SPEC

RESULT-UNIT      ::= result-unit UNIT-EXPRESSION
UNIT-EXPRESSION  ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING     ::= unit-binding UNIT-NAME UNIT-SPEC
UNIT-TERM        ::= UNIT-REDUCTION | UNIT-TRANSLATION | AMALGAMATION
                   | LOCAL-UNIT | UNIT-APPL
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
UNIT-REDUCTION   ::= unit-reduction UNIT-TERM RESTRICTION
AMALGAMATION     ::= amalgamation UNIT-TERM+
LOCAL-UNIT       ::= local-unit UNIT-DEFN+ UNIT-TERM
UNIT-APPL        ::= unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT     ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*

ARCH-SPEC-NAME   ::= SIMPLE-ID
UNIT-NAME        ::= SIMPLE-ID
```

A.5 Specification Libraries

```
LIB-DEFN         ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM         ::= SPEC-DEFN | VIEW-DEFN
                   | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
                   | DOWNLOAD-ITEMS

DOWNLOAD-ITEMS   ::= download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP ::= ITEM-NAME | ITEM-NAME-MAP
ITEM-NAME-MAP    ::= item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME        ::= SIMPLE-ID

LIB-NAME         ::= LIB-ID | LIB-VERSION
LIB-VERSION      ::= lib-version LIB-ID VERSION-NUMBER
VERSION-NUMBER   ::= version-number NUMBER+
LIB-ID           ::= DIRECT-LINK | INDIRECT-LINK
DIRECT-LINK      ::= direct-link URL
INDIRECT-LINK    ::= indirect-link PATH
```

Appendix B

Abbreviated Abstract Syntax

The full grammar, defining the same (tree) language but using more nonterminal symbols, is given in Appendix A.

The following nonterminal symbols correspond to lexical syntax, and are left unspecified in the abstract syntax: `WORDS`, `DOT-WORDS`, `SIGNS`, `DIGIT`, `DIGITS`, `NUMBER`, `QUOTED-CHAR`, `PLACE`, `URL`, and `PATH`.

B.1 Basic and Subsorted Specifications

```
BASIC-SPEC       ::= basic-spec BASIC-ITEMS*

BASIC-ITEMS      ::= SIG-ITEMS
                   | free-datatype DATATYPE-DECL+
                   | sort-gen SIG-ITEMS+
                   | var-items VAR-DECL+
                   | local-var-axioms VAR-DECL+ FORMULA+
                   | axiom-items FORMULA+

SIG-ITEMS        ::= sort-items SORT-ITEM+
                   | op-items OP-ITEM+
                   | pred-items PRED-ITEM+
                   | datatype-items DATATYPE-DECL+

SORT-ITEM        ::= sort-decl SORT+
                   | subsort-decl SORT+ SORT
                   | subsort-defn SORT VAR SORT FORMULA
                   | iso-decl SORT+

OP-ITEM          ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
                   | op-defn OP-NAME  OP-HEAD TERM
OP-TYPE          ::= total-op-type   SORT-LIST SORT
                   | partial-op-type SORT-LIST SORT
SORT-LIST        ::= sort-list SORT*
```

```
OP-ATTR          ::= assoc-op-attr | comm-op-attr | idem-op-attr
                   | unit-op-attr TERM
OP-HEAD          ::= total-op-head   ARG-DECL* SORT
                   | partial-op-head ARG-DECL* SORT
ARG-DECL         ::= arg-decl VAR+ SORT

PRED-ITEM        ::= pred-decl PRED-NAME+ PRED-TYPE
                   | pred-defn PRED-NAME  PRED-HEAD FORMULA
PRED-TYPE        ::= pred-type SORT-LIST
PRED-HEAD        ::= pred-head ARG-DECL*

DATATYPE-DECL    ::= datatype-decl SORT ALTERNATIVE+
ALTERNATIVE      ::= total-construct   OP-NAME COMPONENTS*
                   | partial-construct OP-NAME COMPONENTS+
                   | subsorts SORT+
COMPONENTS       ::= total-select   OP-NAME+ SORT
                   | partial-select OP-NAME+ SORT
                   | SORT

VAR-DECL         ::= var-decl VAR+ SORT

FORMULA          ::= quantification QUANTIFIER VAR-DECL+ FORMULA
                   | conjunction FORMULA+
                   | disjunction FORMULA+
                   | implication FORMULA FORMULA
                   | equivalence FORMULA FORMULA
                   | negation FORMULA
                   | true-atom | false-atom
                   | predication PRED-SYMB TERMS
                   | definedness TERM
                   | existl-equation TERM TERM
                   | strong-equation TERM TERM
                   | membership TERM SORT
QUANTIFIER       ::= universal | existential | unique-existential
PRED-SYMB        ::= PRED-NAME | qual-pred-name PRED-NAME PRED-TYPE

TERMS            ::= terms TERM*
TERM             ::= SIMPLE-ID
                   | qual-var VAR SORT
                   | application OP-SYMB TERMS
                   | sorted-term TERM SORT
                   | cast TERM SORT
                   | conditional TERM FORMULA TERM
OP-SYMB          ::= OP-NAME | qual-op-name OP-NAME OP-TYPE

SORT             ::= TOKEN-ID
OP-NAME          ::= ID
PRED-NAME        ::= ID
VAR              ::= SIMPLE-ID

SIMPLE-ID        ::= WORDS
```

```
ID               ::= TOKEN-ID | MIXFIX-ID
TOKEN-ID         ::= TOKEN
MIXFIX-ID        ::= TOKEN-PLACES
TOKEN-PLACES     ::= token-places TOKEN-OR-PLACE+
TOKEN-OR-PLACE   ::= TOKEN | PLACE
TOKEN            ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
```

B.2 Structured Specifications

```
SPEC             ::= BASIC-SPEC
                   | translation SPEC RENAMING
                   | reduction SPEC RESTRICTION
                   | union SPEC+
                   | extension SPEC+
                   | free-spec SPEC
                   | local-spec SPEC SPEC
                   | closed-spec SPEC
                   | spec-inst SPEC-NAME FIT-ARG*

RENAMING         ::= renaming SYMB-MAP-ITEMS+

RESTRICTION      ::= hide   SYMB-ITEMS+
                   | reveal SYMB-MAP-ITEMS+

SPEC-DEFN        ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY       ::= genericity PARAMS IMPORTED
PARAMS           ::= params SPEC*
IMPORTED         ::= imported SPEC*

FIT-ARG          ::= fit-spec SPEC SYMB-MAP-ITEMS*
                   | fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN        ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE
                               SYMB-MAP-ITEMS*
VIEW-TYPE        ::= view-type SPEC SPEC

SYMB-ITEMS       ::= symb-items     SYMB-KIND SYMB+
SYMB-MAP-ITEMS   ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-KIND        ::= implicit | sorts-kind | ops-kind | preds-kind

SYMB             ::= ID | qual-id ID TYPE
TYPE             ::= OP-TYPE | PRED-TYPE
SYMB-MAP         ::= symb-map SYMB SYMB
SYMB-OR-MAP      ::= SYMB | SYMB-MAP

SPEC-NAME        ::= SIMPLE-ID
VIEW-NAME        ::= SIMPLE-ID

TOKEN-ID         ::= ... | COMP-TOKEN-ID
MIXFIX-ID        ::= ... | COMP-MIXFIX-ID
COMP-TOKEN-ID    ::= comp-token-id TOKEN ID+
COMP-MIXFIX-ID   ::= comp-mixfix-id TOKEN-PLACES ID+
```

B.3 Architectural Specifications

```
ARCH-SPEC-DEFN   ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC        ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT
                   | ARCH-SPEC-NAME
UNIT-DECL-DEFN   ::= UNIT-DECL | UNIT-DEFN

UNIT-DECL        ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
UNIT-IMPORTED    ::= unit-imported UNIT-TERM*
UNIT-DEFN        ::= unit-defn UNIT-NAME UNIT-EXPRESSION

UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | arch-unit-spec ARCH-SPEC
                   | closed-unit-spec UNIT-SPEC
UNIT-TYPE        ::= unit-type SPEC* SPEC

RESULT-UNIT      ::= result-unit UNIT-EXPRESSION
UNIT-EXPRESSION  ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING     ::= unit-binding UNIT-NAME UNIT-SPEC
UNIT-TERM        ::= unit-translation UNIT-TERM RENAMING
                   | unit-reduction UNIT-TERM RESTRICTION
                   | amalgamation UNIT-TERM+
                   | local-unit UNIT-DEFN+ UNIT-TERM
                   | unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT     ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*

ARCH-SPEC-NAME   ::= SIMPLE-ID
UNIT-NAME        ::= SIMPLE-ID
```

B.4 Specification Libraries

```
LIB-DEFN         ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM         ::= SPEC-DEFN | VIEW-DEFN
                   | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
                   | download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP ::= ITEM-NAME | item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME        ::= SIMPLE-ID

LIB-NAME         ::= LIB-ID | LIB-VERSION
LIB-VERSION      ::= lib-version LIB-ID VERSION-NUMBER
VERSION-NUMBER   ::= version-number NUMBER+
LIB-ID           ::= direct-link URL | indirect-link PATH
```

Appendix C

Concrete Syntax

The relationship between the concrete syntax and the corresponding abstract syntax is rather straightforward—except that mapping the use of mixfix notation in a concrete `ATOM` to an abstract `ATOM` depends on the declared operation and predicate symbols (although not on their profiles). Here, the relationship is merely suggested by the use of the same nonterminal symbols in the concrete and abstract grammars.

Examples of specifications illustrating the concrete syntax are given in Appendix E. Parsers for CASL are available via the CoFI Tools task group web page.

C.1 Introduction

The concrete syntax of CASL involves both input syntax (for writing and editing, and subsequent parsing) and display format (for browsing on the screen, and publication on paper). The input syntax is easy to relate to the display format, and also sufficiently readable for use in (plain-text) e-mail messages.

Section C.2 below provides a context-free grammar for the CASL input syntax. It has been derived systematically from the 'abbreviated' abstract syntax grammar in Appendix B, except for the productions for mixfix formulae and terms. The context-free grammar is ambiguous; Section C.3 explains various precedence rules for disambiguation, and the intended grouping of mixfix formulae and terms. Section C.4 specifies the lexical symbols of the concrete syntax. Section C.5 shows how comments and various kinds of annotations may be written. Finally, Section C.6 introduces several annotations used to provide literal syntax for numbers, strings, and lists. The CASL display format is described in Appendix D.

C.2 Context-Free Syntax

The grammar in this section uses uppercase words for nonterminal symbols, allowing also hyphens. All other characters stand for themselves, with the following exceptions:

- '`::=`' and '`|`' are generally used as meta-notation, as in BNF;
- A string of characters enclosed in double quotation marks '`"..."`' always stands for the enclosed characters themselves;
- '$N\ t \ldots t\ N$' indicates one or more repetitions of the nonterminal symbol N separated by the terminal symbol t (which is usually a comma or semicolon);
- '$N \ldots N$' is simply one or more repetitions of N (occasionally, N here is a sequence of terminal and nonterminal symbols, such as '`[ SPEC ]`');
- '`var/vars`' indicates that the singular and plural forms may be used interchangeably, and similarly for other keywords; '`end/`' indicates that the use of '`end`' is optional, and similarly for semicolons: '`;/`'.

The following nonterminal symbols are for lexical syntax, and defined in Section C.4: `WORDS`, `DOT-WORDS`, `NO-BRACKET-SIGNS`, `DIGIT`, `DIGITS`, `QUOTED-CHAR`, `FRACTION`, `FLOATING`, `STRING`; the lexical syntax of `URL` and `PATH` is left open. Lexical analysis for CASL is generally independent of the context-free parsing (apart from the recognition of `URL` and `PATH`, which may appear in libraries but not within individual specifications).

Context-free parsing of CASL specifications according to the grammar in this section yields a parse tree where terms and formulae occurring in axioms and definitions have been grouped with respect to explicit parentheses and brackets, but where the intended applicative structure has not yet been recognized. A further phase of *mixfix grouping analysis* is needed, dependent on the identifiers declared in the specification and on parsing annotations, before the parse tree can be mapped to a complete abstract syntax tree.

C.2.1 Basic Specifications with Subsorts

```
BASIC-SPEC      ::= BASIC-ITEMS...BASIC-ITEMS  |  { }

BASIC-ITEMS     ::= SIG-ITEMS
                  | free      type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | generated type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | generated { SIG-ITEMS...SIG-ITEMS } ;/
                  | var/vars VAR-DECL ;...; VAR-DECL ;/
                  | forall VAR-DECL ;...; VAR-DECL
                            "." FORMULA "."..."." FORMULA ;/
                  | "." FORMULA "."..."." FORMULA ;/
```

The following alternative concrete syntax productions:

```
BASIC-ITEMS     ::=  var/vars VAR-DECL ;...; VAR-DECL
                            "." FORMULA "."..."." FORMULA ;/
                  |  axiom/axioms FORMULA ;...; FORMULA ;/
```

are included in CASL v1.0.1 for backwards compatibility with v1.0, but may be removed in some future version.

```
SIG-ITEMS       ::= sort/sorts SORT-ITEM ;...; SORT-ITEM ;/
                  | op/ops OP-ITEM ;...; OP-ITEM ;/
                  | pred/preds PRED-ITEM ;...; PRED-ITEM  ;/
                  | type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/

SORT-ITEM       ::= SORT ,..., SORT
                  | SORT ,..., SORT < SORT
                  | SORT = { VAR : SORT "." FORMULA }
                  | SORT =...= SORT

OP-ITEM         ::= OP-NAME ,..., OP-NAME : OP-TYPE
                  | OP-NAME ,..., OP-NAME : OP-TYPE , OP-ATTR ,..., OP-ATTR
                  | OP-NAME OP-HEAD = TERM

OP-TYPE         ::= SOME-SORTS  -> SORT    |  SORT
                  | SOME-SORTS  -> ? SORT  |  ? SORT

SOME-SORTS      ::= SORT *...* SORT

OP-ATTR         ::= assoc  |  comm  |  idem  |  unit TERM

OP-HEAD         ::= ( ARG-DECL ;...; ARG-DECL ) : SORT    |  : SORT
                  | ( ARG-DECL ;...; ARG-DECL ) : ? SORT  |  : ? SORT
ARG-DECL        ::= VAR ,..., VAR : SORT

PRED-ITEM       ::= PRED-NAME ,..., PRED-NAME : PRED-TYPE
                  | PRED-NAME PRED-HEAD <=> FORMULA
                  | PRED-NAME <=> FORMULA

PRED-TYPE       ::= SOME-SORTS  |  ( )

PRED-HEAD       ::= ( ARG-DECL ;...; ARG-DECL )
```

```
DATATYPE-DECL   ::= SORT "::=" ALTERNATIVE "|"..."|" ALTERNATIVE
ALTERNATIVE     ::= OP-NAME  ( COMPONENT ;...; COMPONENT )
                  | OP-NAME  ( COMPONENT ;...; COMPONENT ) ?
                  | OP-NAME
                  | sort/sorts SORT ,..., SORT
COMPONENT       ::= OP-NAME ,..., OP-NAME : SORT
                  | OP-NAME ,..., OP-NAME : ? SORT
                  | SORT

VAR-DECL        ::= VAR ,..., VAR : SORT

FORMULA         ::= QUANTIFIER  VAR-DECL ;...; VAR-DECL "." FORMULA
                  | FORMULA /\ FORMULA /\.../\ FORMULA
                  | FORMULA \/ FORMULA \/...\/ FORMULA
                  | FORMULA => FORMULA
                  | FORMULA if FORMULA
                  | FORMULA <=> FORMULA
                  | not FORMULA
                  | true | false
                  | def TERM
                  | TERM =e= TERM
                  | TERM = TERM
                  | TERM in SORT
                  | ( FORMULA )
                  | MIXFIX...MIXFIX

QUANTIFIER      ::= forall  |  exists  |  exists!

TERMS           ::= TERM ,..., TERM

TERM            ::= MIXFIX...MIXFIX

MIXFIX          ::= NO-BRACKET-TOKEN  |  LITERAL  |  PLACE
                  | QUAL-PRED-NAME  |  QUAL-VAR-NAME  |  QUAL-OP-NAME
                  | TERM : SORT
                  | TERM as SORT
                  | TERM when FORMULA else TERM
                  | ( TERMS )
                  | [ TERMS ]  |  [ ]
                  | { TERMS }  |  { }

QUAL-VAR-NAME   ::= ( var VAR : SORT )

QUAL-PRED-NAME  ::= ( pred PRED-NAME : PRED-TYPE )

QUAL-OP-NAME    ::= ( op OP-NAME : OP-TYPE )

SORT            ::= TOKEN-ID
OP-NAME         ::= ID
PRED-NAME       ::= ID
VAR             ::= SIMPLE-ID
```

```
SIMPLE-ID       ::= WORDS
ID              ::= TOKEN-ID  |  MIXFIX-ID
TOKEN-ID        ::= TOKEN
MIXFIX-ID       ::= TOKEN-ID PLACE-TOKEN-ID ... PLACE-TOKEN-ID
                  |          PLACE-TOKEN-ID ... PLACE-TOKEN-ID
PLACE-TOKEN-ID  ::= PLACE TOKEN-ID
                  | PLACE
PLACE           ::= __

TOKEN           ::= WORDS  |  DOT-WORDS  |  DIGIT  |  QUOTED-CHAR
                  | SIGNS
NO-BRACKET-TOKEN::= WORDS  |  DOT-WORDS  |  DIGIT  |  QUOTED-CHAR
                  | NO-BRACKET-SIGNS

SIGNS           ::= NO-BRACKET-SIGNS  |  BRACKET-SIGNS
                  | NO-BRACKET-SIGNS BRACKET-SIGNS
BRACKET-SIGNS   ::= BRACKET SIGNS
                  | BRACKET
BRACKET         ::= [  |  ]  |  {  |  }

LITERAL         ::= DIGITS  |  FRACTION  |  FLOATING  |  STRING
```

C.2.2 Structured Specifications

```
SPEC            ::= BASIC-SPEC
                  | SPEC RENAMING
                  | SPEC RESTRICTION
                  | SPEC and SPEC and...and SPEC
                  | SPEC then SPEC then...then SPEC
                  | free GROUP-SPEC
                  | local SPEC within SPEC
                  | closed GROUP-SPEC
                  | GROUP-SPEC
GROUP-SPEC      ::= { SPEC }
                  | SPEC-NAME
                  | SPEC-NAME [ FIT-ARG ]...[ FIT-ARG ]

RENAMING        ::= with SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

RESTRICTION     ::= hide SYMB-ITEMS ,..., SYMB-ITEMS
                  | reveal SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

SPEC-DEFN       ::= spec SPEC-NAME = SPEC end/
                  | spec SPEC-NAME SOME-GENERICS = SPEC end/
SOME-GENERICS   ::= SOME-PARAMS  |  SOME-PARAMS SOME-IMPORTED
SOME-PARAMS     ::= [ SPEC ]...[ SPEC ]
SOME-IMPORTED   ::= given GROUP-SPEC ,..., GROUP-SPEC

FIT-ARG         ::= SPEC fit SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS
                  | SPEC
                  | view VIEW-NAME
                  | view VIEW-NAME [ FIT-ARG ]...[ FIT-ARG ]

VIEW-DEFN       ::= view VIEW-NAME : VIEW-TYPE end/
                  | view VIEW-NAME : VIEW-TYPE =
                        SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS end/
                  | view VIEW-NAME SOME-GENERICS : VIEW-TYPE end/
                  | view VIEW-NAME SOME-GENERICS : VIEW-TYPE =
                        SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS end/
VIEW-TYPE       ::= GROUP-SPEC to GROUP-SPEC

SYMB-ITEMS      ::= SYMB
                  | SOME-SYMB-KIND SYMB ,..., SYMB
SYMB-MAP-ITEMS  ::= SYMB-OR-MAP
                  | SOME-SYMB-KIND SYMB-OR-MAP ,..., SYMB-OR-MAP
SOME-SYMB-KIND  ::= sort/sorts  |  op/ops  |  pred/preds

SYMB            ::= ID  |  ID : TYPE
TYPE            ::= OP-TYPE  |  PRED-TYPE
SYMB-MAP        ::= SYMB "|->" SYMB
SYMB-OR-MAP     ::= SYMB  |  SYMB-MAP

SPEC-NAME       ::= SIMPLE-ID
VIEW-NAME       ::= SIMPLE-ID
TOKEN-ID        ::= ...  |  TOKEN [ ID ,..., ID ]
```

C.2.3 Architectural Specifications

```
ARCH-SPEC-DEFN  ::= arch spec ARCH-SPEC-NAME = ARCH-SPEC end/
ARCH-SPEC       ::= BASIC-ARCH-SPEC  |  GROUP-ARCH-SPEC
GROUP-ARCH-SPEC ::= { ARCH-SPEC }  |  ARCH-SPEC-NAME
BASIC-ARCH-SPEC ::= unit/units UNIT-DECL-DEFN ;...; UNIT-DECL-DEFN ;/
                               result UNIT-EXPRESSION ;/

UNIT-DECL-DEFN  ::= UNIT-DECL  |  UNIT-DEFN
UNIT-DECL       ::= UNIT-NAME : UNIT-SPEC
                          given GROUP-UNIT-TERM ,..., GROUP-UNIT-TERM
                  | UNIT-NAME : UNIT-SPEC
UNIT-DEFN       ::= UNIT-NAME = UNIT-EXPRESSION

UNIT-SPEC-DEFN  ::= unit spec SPEC-NAME = UNIT-SPEC end/
UNIT-SPEC       ::= GROUP-SPEC
                  | GROUP-SPEC *...* GROUP-SPEC -> GROUP-SPEC
                  | arch spec GROUP-ARCH-SPEC
                  | closed UNIT-SPEC

UNIT-EXPRESSION ::= lambda UNIT-BINDING ;...; UNIT-BINDING "." UNIT-TERM
                  | UNIT-TERM
UNIT-BINDING    ::= UNIT-NAME : UNIT-SPEC

UNIT-TERM       ::= UNIT-TERM RENAMING
                  | UNIT-TERM RESTRICTION
                  | UNIT-TERM and...and UNIT-TERM
                  | local UNIT-DEFN ;...; UNIT-DEFN ;/ within UNIT-TERM
                  | GROUP-UNIT-TERM
GROUP-UNIT-TERM ::= { UNIT-TERM }
                  | UNIT-NAME
                  | UNIT-NAME [ FIT-ARG-UNIT ]...[ FIT-ARG-UNIT ]

FIT-ARG-UNIT    ::= UNIT-TERM
                  | UNIT-TERM fit SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

ARCH-SPEC-NAME  ::= SIMPLE-ID
UNIT-NAME       ::= SIMPLE-ID
```

C.2.4 Specification Libraries

```
LIB-DEFN        ::= library LIB-NAME LIB-ITEM...LIB-ITEM
LIB-ITEM        ::= SPEC-DEFN  |  VIEW-DEFN
                  | ARCH-SPEC-DEFN  |  UNIT-SPEC-DEFN
                  | from LIB-NAME
                       get ITEM-NAME-OR-MAP ,..., ITEM-NAME-OR-MAP end/

ITEM-NAME-OR-MAP::= ITEM-NAME  |  ITEM-NAME "|->" ITEM-NAME
ITEM-NAME       ::= SIMPLE-ID
LIB-NAME        ::= LIB-ID  |  LIB-ID VERSION-NUMBER
LIB-ID          ::= URL  |  PATH
VERSION-NUMBER  ::= version NUMBER "."..."." NUMBER
NUMBER          ::= DIGIT  |  DIGITS
```

C.3 Disambiguation

The context-free grammar given in Section C.2 for input syntax is quite ambiguous. This section explains various precedence rules for disambiguation, and the intended grouping of mixfix formulae and terms (which is to be recognized in a separate phrase, dependent on the declared symbols and parsing annotations).

C.3.1 Precedence

At the level of structured specifications, ambiguities of grouping are resolved as follows:

- '`free`' and '`closed`' have the highest precedence;
- '`with`', '`reveal`', and '`hide`' have lower precedence;
- '`within`' has still lower precedence;
- '`and`' has lower precedence than all the above; and
- '`then`' has the lowest precedence of all.

At the level of architectural specifications, ambiguities of grouping in unit terms are resolved in the same way as for structured specifications. Moreover, a `SPEC-NAME` occurring as a `UNIT-SPEC` gives rise to just the `SPEC-NAME` itself in the abstract syntax tree, rather than a `UNIT-TYPE` with an empty list `SPEC*` of argument specifications.

In `BASIC-ITEMS`, a list of '`. FORMULA ... . FORMULA`' extends as far to the right as possible. Within a `FORMULA`, the use of prefix and infix notation for the logical connectives gives rise to some potential ambiguities. These are resolved as follows:

- '`not FORMULA`' has the highest precedence;
- '`FORMULA /\.../\ FORMULA`' and '`FORMULA \/...\/ FORMULA`' both have lower precedence, but may not be combined without explicit grouping;
- The connectives '`FORMULA => FORMULA`', '`FORMULA if FORMULA`', '`FORMULA <=> FORMULA`' all have even lower precedence. When repeated, '`=>`' groups to the right, whereas '`if`' groups to the left; '`<=>`' may not be repeated without explicit grouping. These constructs may not be combined without explicit grouping.
- '`QUANTIFIER VAR-DECL;... . FORMULA`' has the lowest precedence of the logical constructs, with the last `FORMULA` extending as far to the

right as possible, e.g., '`forall x:S . F => G`' is disambiguated as '`forall x:S . (F => G)`', not as '`(forall x:S . F) => G`'.

Moreover, a quantification may be used on the right of a logical connective without grouping parentheses. For instance,

'`F <=> exists x:s . G <=> H`' is parsed as
'`F <=> (exists x:s . G <=> H)`'.

The declaration[1] of infix, prefix, postfix, and general mixfix operation symbols may introduce further potential ambiguities, which are partially resolved as follows (remaining ambiguities have to be eliminated by explicit use of grouping parentheses in terms, or by use of parsing annotations):

- Ordinary function application '`OP-SYMB(TERMS)`' has the highest precedence.
- Applications of all postfix symbols have the next-highest precedence within terms after ordinary function application. This extends to all mixfix operation symbols of the form '`__ ... __ TOKEN`', and to sorted terms and casts.
- Applications of all prefix symbols have the next-highest precedence within terms after postfixes. This extends to all mixfix operation symbols of the form '`TOKEN __ ... __`'.
- Applications of infix symbols have the next-highest precedence within terms after prefixes. This extends to all mixfix symbols of the form '`__ ... __ ... __`'. Mixtures of different infix symbols and iterations of the same infix symbol have to be explicitly grouped—although the attribute of associativity implies a parsing annotation that allows iterated applications of that symbol to be written without grouping.
- The conditional '`TERM when FORMULA else TERM`' has the weakest precedence within terms, and iterations such as:

  T_1 `when` F_1 `else` T_2 `when` F_2 `else` T_3

  are implicitly grouped to the right:

  T_1 `when` F_1 `else (`T_2 `when` F_2 `else` T_3`)`

Various other techniques for allowing the omission of grouping parentheses and/or list-separators in input (and display) are familiar from previous specification and programming languages, e.g., user-specified precedence (relative or absolute), and the "offside" rule. Moreover, not all parsers are expected to implement full mixfix notation. CASL therefore allows *parsing annotations*

[1] Declarations occurring anywhere in the enclosing list of basic items are taken into account when disambiguating the grouping of symbols in a term.

on (libraries of) specifications, to indicate the possible omission of grouping parentheses, and the degree of use of mixfix notation. (Such annotations are expected to apply uniformly to CASL sublanguages, and to most extensions.) Parsing annotations may even override the rules given above for the relative precedence of postfix, prefix, and infix symbols. See Section C.5.2.3 for details of the available parsing annotations.

C.3.2 Mixfix Grouping Analysis

Note that `ID` is *not* an alternative of `MIXFIX`, since the notation for compound identifiers could be confused with mixfix notation involving square brackets.

Mixfix grouping analysis of a specification should be *equivalent* to context-free parsing according to a derived grammar—obtained from the grammar in Section C.2 by replacing the phrases involving `MIXFIXES` with phrases determined (partly) by the declared symbols, as follows:

```
FORMULA ::= ... | QUAL-PRED-NAME
          | QUAL-PRED-NAME ( TERMS )

TERMS   ::= TERM ,..., TERM

TERM    ::= LITERAL | QUAL-VAR-NAME | QUAL-OP-NAME
          | QUAL-OP-NAME ( TERMS )
          | TERM : SORT
          | TERM as SORT
          | TERM when FORMULA else TERM
          | ( TERM )
```

plus

`TERM ::= ...` | *id*

for each declared variable or constant name *id*, plus

`TERM ::=` *id* `( TERMS )`

for each declared operation symbol *id* of positive arity, plus

`TERM ::=` t_1 `TERM` t_2 `... TERM` t_n

for each declared mixfix operation symbol $t_1 \_\_ t_2 \ldots \_\_ t_n$ (with t_1 and t_n possibly empty), plus

`FORMULA ::= ...` | *id*

for each declared predicate constant name *id*, plus

`FORMULA ::=` *id* `( TERMS )`

for each declared predicate symbol *id* of positive arity, plus

`FORMULA ::=` t_1 `TERM` t_2 `...` `TERM` t_n

for each declared mixfix predicate symbol $t_1 \_\_t_2 \ldots \_\_t_n$ (with t_1 and t_n possibly empty).

It would be possible to obtain a fixed grammar for a sub-language of CASL lacking mixfix notation in a similar way, using the appropriate kinds of `ID` in place of the declared *id*s above. (It may be convenient to obtain all these various grammars as extensions of a root grammar that is completely uncommitted about the notation used for applications, etc.)

The context-free parsing during mixfix grouping analysis involves disambiguation as determined by the general precedence rules for applications (see Section C.3.1) and by any parsing annotations (see Section C.5.2.3).

C.4 Lexical Syntax

This section defines the lexical syntax of WORDS, DOT-WORDS, NO-BRACKET-SIGNS, DIGIT, DIGITS, and QUOTED-CHAR, which are used in Section C.2, together with that of FRACTION, FLOATING, and STRING, which are used also in Section C.6. The lexical syntax of URL and PATH is left open; they are to be recognized only directly following the keywords '`library`' and '`from`'.

Spaces and other layout characters terminate lexical symbols (except for QUOTED-CHAR and STRING) and are otherwise ignored. The next lexical symbol recognized is as long as possible.

```
WORDS         ::= WORD _ ... _ WORD

DOT-WORDS     ::= . WORDS

WORD          ::= LETTER-P-D ... LETTER-P-D
LETTER-P-D    ::= LETTER | "'" | DIGIT

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
                | À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì
                | Í | Î | Ï | Ñ | Ò | Ó | Ô | Õ | Ö | Ø | Ù | Ú | Û
                | Ü | Ý | ß | à | á | â | ã | ä | å | æ | ç | è | é
                | ê | ë | ì | í | î | ï | ñ | ò | ó | ô | õ | ö | ø
                | ù | ú | û | ü | ý | ÿ

DIGIT         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

DIGITS        ::= DIGIT DIGIT ... DIGIT
```

A WORDS must start with a LETTER, and must not be one of the *reserved keywords* used in the context-free syntax in Section C.2. The (51) keywords are:

```
and arch as assoc axiom axioms closed comm def else end
exists false fit forall free from generated get given
hide idem if in lambda library local not op ops pred preds
result reveal sort sorts spec then to true type types
unit units var vars version view when with within .
```

LETTER includes all the ISO Latin-1 national and accented letters except for the Icelandic 'eth' and 'thorn'.

```
NO-BRACKET-SIGNS ::= NO-BRACKET-SIGN ... NO-BRACKET-SIGN
```

A NO-BRACKET-SIGNS must not be one of the following *reserved symbols*:

```
:  :?  ::=  =  =>  <=>  .  ·  |  |->  \/  /\ ¬
```

These sequences of characters may however be used together with other characters in a `NO-BRACKET-SIGNS`. For example, '`==`', '`:=`', and '`||`' is each recognized as a complete `NO-BRACKET-SIGNS`. Note that identifiers that start or finish with a `NO-BRACKET-SIGNS` need to be separated by (e.g.) a space from adjacent reserved symbols: a sequence of characters such as ' `#:` ' is always recognized as a single symbol, whereas ' `# :` ' is recognized as two symbols.

Despite its use in the context-free syntax as a terminal symbol, a single character '`<`', '`*`', '`?`', '`!`', or '`/`' is also recognized as a complete `NO-BRACKET-SIGNS`. The ISO Latin-1 characters for product '×', negation '¬', and raised dot '·' are recognized as alternatives for the terminal symbols '`*`', '`not`', and '`.`', respectively.

```
NO-BRACKET-SIGN  ::= + | - | * | / | \ | & | = | < | >
                   | ! | ? | : | . | $ | @ | # | ^ | ~
                   | ¡ | ¿ | × | ÷ | £ | © | ± | ¶ | §
                   | ¹ | ² | ³ | · | ¢ | ° | ¬ | µ | "|"
```

Note that `NO-BRACKET-SIGN` does *not* include the following ASCII signs:

```
( ) [ ] { } ; , ‘ " %
```

nor the ISO Latin-1 signs for general currency, yen, broken vertical bar, registered trade mark, masculine and feminine ordinals, left and right angle quotes, fractions, soft hyphen, acute accent, cedilla, macron, and umlaut.

```
QUOTED-CHAR ::= "'" CHAR "'"

CHAR        ::= " " | ! | '\"' | # | $ | ...
                | "\'" | ... | "\\" | ...
                | \n | \t | \r | \v | \b | \f | \a | \?
                | \000 | ... | \255
                | \x00 | ... | \xFF
                | \o000 | ... | \o377

FRACTION    ::= NUMBER . NUMBER

FLOATING    ::= NUMBER "E" OPT-SIGN NUMBER
                | FRACTION "E" OPT-SIGN NUMBER
OPT-SIGN    ::= + | - |

STRING      ::= '"' '"'
                | '"' CHAR ... CHAR '"'
```

Pairs of single quotes '`'...'`' (with no spaces between them) are used in the above productions to indicate symbols containing the double quote character '`"`', and vice versa.

C.5 Comments and Annotations

Both comments and annotations can be used to provide auxiliary information that gets attached to the abstract syntax trees of CASL specifications during parsing. Such auxiliary information does not affect the semantics of the specifications. Comments may also be used to ignore parts of specifications (so-called "commenting-out").

The general form of comments and annotations is similar: they start with a percent character '%', and extend either to the end of the line, or to the end of the (shortest) following bracketed group. Groups with different bracket signs may be nested within each other, but nesting of groups of the same kind is not possible.

```
COMMENT       ::= COMMENT-LINE | COMMENT-GROUP | COMMENT-OUT
COMMENT-LINE  ::= %% TEXT-LINE
COMMENT-GROUP ::= %{ TEXT-LINES }%
COMMENT-OUT   ::= %[ TEXT-LINES ]%

ANNOTE        ::= ANNOTE-LINE | ANNOTE-GROUP | LABEL
ANNOTE-LINE   ::= %WORDS TEXT-LINE
ANNOTE-GROUP  ::= %WORDS( TEXT-LINES )%
LABEL         ::= %( TEXT-LINES )%

TEXT          ::= NOT-NEWLINE ... NOT-NEWLINE | EMPTY
TEXT-LINE     ::= TEXT NEWLINE | NEWLINE
TEXT-LINES    ::= TEXT NEWLINE ... NEWLINE TEXT
EMPTY         ::=
```

NEWLINE denotes the character that indicates the start of a new line; NOT-NEWLINE denotes all the other printable ISO Latin-1 characters, together with the space and tab characters (which thus may appear in TEXT but not in WORDS). In ANNOTE-LINE and ANNOTE-GROUP, spaces are not allowed before the WORDS, and a space directly following the WORDS distinguishes an ANNOTE-LINE from an ANNOTE-GROUP.

A single-line comment of the form '%%*text newline*' is equivalent to '%{*text*}%'; the latter form also allows multi-line comments. Similarly, a single-line annotation of the form '%*words text newline*' is equivalent to '%*words*(*text*)%'.

An ordinary comment or an annotation may be inserted only at restricted positions in specifications:

1. following a SORT-ITEM, OP-ITEM, PRED-ITEM, ALTERNATIVE, or AXIOM (and its terminating ';', if any), where it applies to the preceding construct;
2. preceding a list of the above constructs, or of LIB-ITEMs, where it applies to the construct containing the list;

3. preceding the body of the definition of a sort, operation, predicate, datatype, or view, where it applies to the enclosing definition; or.

4. preceding a `SPEC` or `ARCH-SPEC`, where it applies to that construct.

(The positions of particular kinds of annotations are further restricted in Section C.5.2.2.) At each position, only one comment is allowed, but there may be more than one annotation (even of the same kind). A `COMMENT-OUT` may occur anywhere (between lexical symbols) and gets discarded.

Annotations on a `LIBRARY` (i.e., preceding its first `LIB-ITEM`) are global to all its `LIB-ITEM`s, and to all libraries that download any of those items. *Conflicting* annotations that arise due to downloading from different remote libraries are simply ignored, whereas local annotations override conflicting annotations from remote libraries. Conflicting annotations within the same library are ignored as well.

C.5.1 Comments

```
COMMENT       ::= COMMENT-LINE | COMMENT-GROUP | COMMENT-OUT

COMMENT-LINE  ::= %% TEXT-LINE
COMMENT-GROUP ::= %{ TEXT-LINES }%
```

An ordinary comment `COMMENT-LINE` at the end of a line is written '`%%`*text*', whereas a comment within a line, or a multi-line comment, is always written '`%{`*text-lines*`}%`'.

CASL specification text within comments should be delimited by a bracketed group of the form '`%CASL(...)%`', to allow its appropriate display. The kind of CASL construct may be indicated by using a non-terminal symbol from the CASL abstract syntax (such as '`ID`' or '`TERM`') instead of '`CASL`'.

The preferred formatting of a part of a comment by different formatters may be indicated using the following syntax (which is similar to that of display annotations, see Section C.5.2.2):

`%display` *text* `%HTML ... %LATEX ... %RTF ...`

at the end of a line, or, possibly over several lines:

`%display(` *text* `%HTML ... %LATEX ... %RTF ... )%`

Both the above indicate that the *text* is to be displayed according to the formatting instructions given for HTML, LaTeX, and RTF (which may be listed in any order, or omitted). Formatters for which there are no instructions should display the *text* as input.

```
COMMENT-OUT   ::= %[ TEXT-LINES ]%
```

Arbitrary text is commented-out by using '`%[`*text-lines*`]%`'. Note that the *text-lines* may contain `COMMENT-LINE`s and `COMMENT-GROUP`s, but `COMMENT-OUT`s cannot themselves be nested.

C.5.2 Annotations

```
ANNOTE        ::= ANNOTE-LINE | ANNOTE-GROUP | LABEL
ANNOTE-LINE   ::= %WORDS TEXT-LINE
ANNOTE-GROUP  ::= %WORDS( TEXT-LINES )%
```

The kind of `ANNOTE-LINE` or `ANNOTE-GROUP` is indicated by the `WORDS` occurring after the initial '`%`':

- '`display`' indicates a *display* annotation;
- '`prec`', '`left_assoc`', and '`right_assoc`' indicate *parsing* annotations;
- '`cons`', '`def`', and '`implies`' indicate *semantic* annotations; and
- '`number`', '`floating`', '`string`', and '`list`' indicate annotations concerned with *literal syntax* (see Section C.6).

Each kind of annotation imposes restrictions on the syntax of its text, and on the positions where it may occur. (It is envisaged that further kinds of annotations will be added later, but only with the same general form as indicated above.)

Display, parsing, and literal syntax annotations may occur only at the beginning of libraries (following the `LIB-NAME` of the library), and apply globally. Label and semantic annotations apply only to the construct to which they are attached.

C.5.2.1 Label Annotations

```
LABEL         ::= %( TEXT-LINES )%
```

A label annotation is written '`%(`*text-lines*`)%`', where *text-lines* is the label itself. For instance, in '`%(reverse-NeList)%`' the label is '`reverse-NeList`'.

A label annotation is normally attached to an axiom, although other constructs within a specification `SPEC` may be labelled as well.

Different occurrences of the same label in the same `LIB-ITEM` are regarded as conflicting.

C.5.2.2 Display Annotations

```
ANNOTE-LINE   ::= %display  TEXT-LINE
ANNOTE-GROUP  ::= %display( TEXT-LINES )%
```

A single-line display annotation `ANNOTE-LINE` is written:

`%display` *id* `%HTML ... %LATEX ... %RTF ...`

A multi-line display annotation `ANNOTE-GROUP` is written:

`%display(` *id* `%HTML ... %LATEX ... %RTF ... )%`

Both indicate that the identifier with input syntax *id* is to be displayed according to the formatting instructions given for HTML, LaTeX, and RTF (which may be listed in any order, or omitted). When there are no instructions given for the language of the formatter being used, the identifier is displayed as its input syntax.[2]

Display annotations for the same identifier are regarded as conflicting unless their formatting instructions are identical, up to reordering.

The following example indicates that the identifier input as '`div`' should be displayed as '÷' by formatters that understand HTML or LaTeX commands:

```
%display div %HTML &divide; %LATEX \div
```

Display annotations generalize to formatting mixfix notation by interpreting the place-holder '`__`' as such in the formatting instructions, e.g.:

```
%display( sum__to__
%HTML  SUM<sub>__<sup>__
%LATEX \sum_{__}^{__}
)%
```

The HTML level is assumed to be 4.0; the version of LaTeXis assumed to be LaTeX2e, using the CASL package [Mos98], in math mode.

C.5.2.3 Parsing Annotations

These annotations are to allow users to specify the precedence and associativity of operation symbols. Their primary purpose is to allow the omission of grouping parentheses in the input; but formatters may also exploit them to avoid superfluous parentheses in the display.

[2] `%`*words* delimiters identifying further formatters may be introduced in future versions of CASL.

Parsing annotations include *precedence* and *associativity* annotations, as well as annotations providing literal syntax for numbers, strings, and lists (see Section C.6).

Precedence

```
ANNOTE-LINE   ::= %prec  TEXT-LINE
ANNOTE-GROUP  ::= %prec( TEXT-LINES )%
```

A single-line precedence annotation `ANNOTE-LINE` is written:

`%prec` $\{id_1$ `,` ... `,` $id_n\}$ `<` $\{id_{n+1}$ `,` ... `,` $id_{n+k}\}$

A multi-line precedence annotation `ANNOTE-GROUP` is written:

`%prec(` $\{id_1$ `,` ... `,` $id_n\}$ `<` $\{id_{n+1}$ `,` ... `,` $id_{n+k}\}$ `)%`

Each id_i is a mixfix identifier of the form '__...__...__'. The relation specifies that for $1 \leq i \leq n$ the symbol id_i has lower priority (i.e., binds weaker) than the symbol id_j, where $n + 1 \leq j \leq n + k$.

It is also possible to specify that mixfix identifiers (which need not to be of form '__...__...__') are *not* allowed to be combined without explicit grouping parentheses. This is done using '`<>`' instead of '`<`' between the groups of identifiers.

In both cases, a precedence annotation involving groups of identifiers abbreviates the collection of corresponding precedence annotations between each pair of identifiers from the two groups.

Two different precedence annotations for the same pair of identifiers are regarded as conflicting.

The precedence annotations determine a pre-order, which is obtained in the following way:

1. Expand all precedence relations into binary relations:
   - from annotations of the form '`%prec` $\{id_1\}$ `<` $\{id_2\}$' we get $\{(id_1, id_2)\}$, and
   - from annotations of the form '`%prec` $\{id_1\}$ `<>` $\{id_2\}$' we get $\{(id_1, id_2), (id_2, id_1)\}$.
2. Take the union of all the expanded precedence relations thus obtained with the predefined precedences listed in Section C.3.1.
3. Take the reflexive transitive closure of this union.

If two symbols occurring in a term or atomic formula are equivalent (i.e. related in both directions) or incomparable (i.e. related in no direction) in

the precedence relation, their grouping has to be explicitly specified by using parentheses.

Associativity

```
ANNOTE-LINE   ::= %left_assoc  TEXT-LINE
                | %right_assoc  TEXT-LINE
ANNOTE-GROUP  ::= %left_assoc( TEXT-LINES )%
                | %right_assoc( TEXT-LINES )%
```

A single-line left-associativity annotation `ANNOTE-LINE` is written:

`%left_assoc` id_1 `, ...,` id_n

A multi-line left-associativity annotation `ANNOTE-GROUP` is written:

`%left_assoc(` id_1 `, ... ,` id_n `)%`

The id_i must be infix operation symbols. Similarly for right-associativity annotations.

In both cases, an associativity annotation involving a group of identifiers abbreviates the collection of corresponding associativity annotations for each identifier in the group. Left and right associativity annotations for the same identifier are regarded as conflicting.

For example, declaring $\_\_+\_\_$ to be left associative means that $t_1+t_2+t_3$ is parsed as $(t_1+t_2)+t_3$, while declaring it to be right associative leads to $t_1+(t_2+t_3)$. If there is no associativity annotation for an infix symbol, it is not allowed to repeat that symbol without explicit grouping using parentheses.

C.5.2.4 Semantic Annotations

These annotations are used to express known (or presumed) features of the semantics of the specification, e.g., that an extension is 'conservative', or that certain formulae are consequences of the specification. Theorem-proving tools may interpret these annotations as *proof obligations*. Note, however, that the annotations do *not* affect the semantics of a specification, regardless of whether the specification has the indicated features or not.

These annotations may immediately follow either:

- a '`then`' keyword within an `EXTENSION`, in which case, let SP be the part of the `EXTENSION` just up to, but excluding the annotated '`then`', and let SP' be the specification immediately following the '`then`'; or
- the equals sign within a `SPEC-DEFN`, in which case, let SP be the union of the imports, extended by the union of the parameters, and let SP' be the body of the specification definition.

Different semantic annotations at the same position are not regarded as conflicting.

Conservative Extension

```
ANNOTE-LINE   ::= %cons NEWLINE
```

The annotation expresses that SP' is a conservative extension of SP, i.e. each SP-model can be expanded to an (SP **then** SP')-model.

Definitional Extension

```
ANNOTE-LINE   ::= %def NEWLINE
```

The annotation expresses that SP' is a definitional extension of SP, i.e. each model of SP can be uniquely extended to a model of (SP **then** SP') (this implies a bijective correspondence between the two model classes).

Note that '`%def`' is strictly stronger than the '`%cons`' annotation.

Implied Extension

```
ANNOTE-LINE   ::= %implies NEWLINE
```

The annotation '`%implies`' is well-formed iff:

1. the signature of (SP **then** SP') is the signature of SP and
2. SP' is a `BASIC-SPEC`.

A well-formed '`%implies`' annotation holds iff the model class of (SP **then** SP') is the model class of SP.

C.6 Syntax for Literals

In this section, several annotations for operations are introduced that can be used to interpret the literal syntax for numbers and strings, and provide a literal syntax for lists.

C.6.1 Literal syntax for numbers

```
LITERAL ::= DIGITS
```

The annotation for declaring an operation to be used for concatenation of digits within a number is written '`%number` f'.

The annotation has the effect that an `DIGITS` of the form $d_1 \ldots d_n$ (where $n > 1$ and each d_i is a `DIGIT`) is translated to the (abstract syntax of) the term $f(f(\ldots f(t_1, t_2) \ldots, t_{n-1}), t_n)$, where t_i is the abstract syntax tree for d_i.

Vice versa, an abstract syntax tree corresponding to a term of the above form which is maximal (i.e., it is not a subterm of a larger term of the same form) is expected to be printed as $d_1 \ldots d_n$.

Different '`%number`' annotations are regarded as conflicting. If there is no '`%number`' annotation, then an `DIGITS` is not recognized as a well-formed `LITERAL`.

```
LITERAL ::= ... | FRACTION | FLOATING
```

The annotation for declaring the operations used for evaluating the decimal point and the exponentiation '`E`' within `FRACTION` or a `FLOATING` is written '`%floating` f`,` g'.

The annotation has the effect that a `FRACTION` of the form $n_1.n_2$ (where each n_i is a `NUMBER`) is translated to the (abstract syntax of) the term $f(t_1, t_2)$, where t_i is the abstract syntax tree for n_i, $i = 1, 2$.

Similarly, a `FLOATING` of the form 'n_1`E`n_2' (where n_1 is a `NUMBER` or a `FRACTION` and n_2 is of form `OPT-SIGN NUMBER`) is translated to the (abstract syntax of) the term $g(t_1, t_2)$, where t_i is the abstract syntax tree for n_i, $i = 1, 2$.

Vice versa, an abstract syntax tree corresponding to a term of one the above forms which is maximal (i.e., it is not a subterm of a larger term of the same form) is expected to be printed as $n_1.n_2$ or $n_1 E n_2$, respectively.

Different '`%floating`' annotations are regarded as conflicting. If there is no '`%floating`' annotation, then neither a `FRACTION` nor a `FLOATING` is recognized as a well-formed `LITERAL`.

C.6.2 Literal syntax for strings

```
LITERAL ::= ... | STRING
```

The annotation for declaring operations for the empty string and for concatenation of a character with a string is written '`%string` c`,` f'.

The annotation has the effect that an `STRING` of the form '`"`c_1 $\ldots c_n$`"`' (where $n \geq 0$ and each c_i is a `CHAR`) is translated to the (abstract syntax of) the term $f(t_1, f(t_2, \ldots f(t_n, c)\ldots))$, where t_i is the abstract syntax tree for the `QUOTED-CHAR` '`'`c_i`'`', or simply to c when $n = 0$.

Vice versa, an abstract syntax tree corresponding to a term of the above form which is maximal (i.e., it is not a subterm of a larger term of the same form) is expected to be printed as '`"`c_1 $\ldots c_n$`"`'.

Different '`%string`' annotations are regarded as conflicting. If there is no '`%string`' annotation, then a `STRING` is not recognized as a well-formed `LITERAL`.

C.6.3 Literal syntax for lists

The annotation for declaring a macro for applying a binary function on a list of arguments is written '`%list` b_1`__`b_2`,` c`,` f'. b_1 and b_2 are `SIGNS`. This annotation can in particular be used to introduce a syntax for lists, e.g., '`%list [__], nil, cons`' allows the use of the notation '`[`x_1`,...,`x_n`]`' for lists constructed using `cons`, starting from the empty list `nil`.

The attribute leads to an extension of the syntax for `LITERAL`s:

```
LITERAL ::= ... | b1 b2
                | b1 TERM , ... , TERM b2
```

A list of the form 'b_1 t_1`,...,`t_n b_2' (where $n \geq 0$ and each t_i is a `TERM`) is translated to the (abstract syntax of) the term $f(u_1, f(u_2, \ldots f(u_n, c)\ldots))$, where u_i is the abstract syntax tree for t_i, or simply to c when $n = 0$.

Vice versa, an abstract syntax tree corresponding to a term of the above form which is maximal (i.e., it is not a subterm of a larger term of the same form) is expected to be printed as 'b_1 t_1`,...,`t_n b_2'.

Different '`%list`' annotations are regarded as conflicting when their pairs of `SIGNS` 'b_1`__`b_2' are identical.

Appendix D

Display Format

This appendix indicates how each input symbol is to be displayed when formatted for printing using LaTeX, as well as for web browsing using HTML. A LaTeXpackage implementing this display format is available [Mos98].

No restrictions are imposed concerning which font families are to be used for displaying CASL specifications.

D.1 Mathematical Symbols

The input symbols in the following table are to be displayed as the mathematical symbols shown below them.

`*`	`->`	`forall`	`exists`	`/\`	`\/`	`=>`	`<=>`	`not`	`in`	`.`	`|->`	`lambda`
$\times$	$\rightarrow$	$\forall$	$\exists$	$\wedge$	$\vee$	$\Rightarrow$	$\Leftrightarrow$	$\neg$	$\in$	$\bullet$	$\mapsto$	λ

When a mathematical symbol is not available (e.g., when browsing HTML on WWW) the input syntax for it may be displayed instead. Moreover, characters whose display format is in ISO Latin-1 may be used for input. This allows the direct input of the symbols displayed as '¬' and '' (also ' • ' may be input as a raised dot), and ensures that the text of a specification as shown by a WWW browser is valid input syntax (at least in the absence of display annotations).

D.2 Keywords

Only keywords that indicate *specification structure* are displayed boldface; all keywords occurring in a `FORMULA`, an `ATTRIBUTE`, or an `ALTERNATIVE` are displayed in the same italic font as identifiers.

D.3 Identifiers

Identifiers for sorts, operations, predicates, and variables are generally displayed with letters in italic: f, x, *Very_Long_Identifier*. Non-letter characters in identifiers are displayed as faithfully as practically possible.

Names for specifications, views, and libraries are displayed with the letters in the SMALL-CAPS font when available, and otherwise in ordinary upper and lower case. Names for units are displayed in the same way as variables.

D.4 Comments and Annotations

If available, a smaller font than normal may be used when displaying comments and annotations.

The delimiters of comments and annotations are always to be displayed in boldface.

Comments '`%%`*text*' and '`%{`*text-lines*`}%`' are to be displayed with the body in the same font as ordinary informal text that might appear before and after a CASL specification (but note that this may be overruled by explicit formatting instructions in the text of the comment). The line breaks of a multi-line comment are to be preserved in the display.

Display annotations '`%display ...`' affect the formatting of identifiers throughout the enclosing library. For the annotation itself, only the input syntax and the display relevant to the formatter being used are to be shown.

Label annotations '`%(`*words*`)%`' are to be displayed flush with the right margin, with the *words* in the same font as used for text in comments.

Other annotations '`%`*words*`...`' are to be displayed with the *words* in boldface, and with any CASL symbols in the body displayed as usual.

Appendix E

Examples

This appendix illustrates the concrete syntax of basic, structured, and architectural specifications in CASL libraries (although not all features of CASL, nor all styles of specifications supported by CASL, are covered). The illustrative specifications are *not* intended for general use in other CASL specifications: comprehensive CASL libraries of specifications of basic datatypes are available separately [RM00], and may be reused in other libraries simply by inserting the appropriate downloading items.

The examples below are shown only in the display format, but the intended input syntax should in general be easy to deduce—in fact, it should be the same text as displayed when browsing the HTML-formatted version of this document (modulo display annotations, which are needed for displaying mathematical symbols such as $\cup$). The input syntax[1] of the examples is available.

[1] `http://www.brics.dk/Projects/CoFI/Documents/CASL/v1.0.1/Sample/Sample/`

E.1 Simple Structured Specifications

library SAMPLE/SIMPLE

%display __=<__ %LATEX __ ≤ __
%display __>=__ %LATEX __ ≥ __

spec TOTAL_ORDER =
  sort *Elem*
  pred $\_\_ \leq \_\_ : Elem \times Elem$
  $\forall x, y, z : Elem$
  - $x \leq x$ %(reflexive)%
  - $x = y \ if \ x \leq y \wedge y \leq x$ %(anti_symmetric)%
  - $x \leq z \ if \ x \leq y \wedge y \leq z$ %(transitive)%
  - $x \leq y \vee y \leq x$ %(total)%

spec MONOID =
  sort *Elem*
  ops $n : Elem$;
    $\_\_ * \_\_ : Elem \times Elem \to Elem, assoc, unit\ n$
  %% Alternatively, just specify the corresponding axioms:
  $\forall x, y, z : Elem$
  - $n * x = x$ %(1)%
  - $x * n = x$ %(2)%
  - $(x * y) * z = x * (y * z)$ %(3)%

spec NAT =
free
{ **sorts** *Nat*;
    Zero, *Pos* < *Nat*
  ops *zero* : *Zero*;
    $succ\_\_ : Nat \to Pos$
}
then **op** $pre\_\_ : Pos \to Nat$
  $\forall x : Nat$
  - $pre(succ\ x) = x$
then
local **pred** $odd\_\_ : Nat$
  $\forall x : Nat$
  - $\neg\ odd\ zero$
  - $odd(succ\ x) \Leftrightarrow \neg\ odd\ x$
within
  sort $Odd = \{n : Nat \bullet odd\ n\}$

E.2 Generic Structured Specifications

library SAMPLE/GENERIC

from SAMPLE/SIMPLE **get** TOTAL_ORDER, NAT

%display __cup__ %LATEX __ $\cup$ __
%display __inset__ %LATEX __ $\subseteq$ __
%display __dot__ %LATEX __ $\cdot$ __

spec ELEM =
sort *Elem*

spec SET1 [ELEM] =
free
{ **type** $Set[Elem] ::= \{\} \mid \{\_\_\}(Elem) \mid \_\_ \cup \_\_(Set[Elem];\ Set[Elem])$
op $\_\_ \cup \_\_ : Set[Elem] \times Set[Elem] \to Set[Elem]$,
$assoc, comm, idem, unit\{\}$
}

spec SET2 [ELEM] =
SET1 [ELEM]
then
pred $\_\_ \subseteq \_\_ : Elem \times Set[Elem]$
$\forall a, b : Elem;\ s, t : Set[Elem]$
- $\neg\ a \subseteq \{\}$
- $a \subseteq \{b\} \Leftrightarrow a = b$
- $a \subseteq (s \cup t) \Leftrightarrow (a \subseteq s) \vee (a \subseteq t)$

spec LIST [ELEM] =
free **type** $List[Elem] ::= nil \mid cons(first :? Elem;\ rest :? List[Elem])$
op $\_\_ ++ \_\_ : List[Elem] \times List[Elem] \to List[Elem], assoc, unit\ nil$
vars $e : Elem;\ l, l' : List[Elem]$
- $cons(e, l) ++ l' = cons(e, l ++ l')$

op $reverse : List[Elem] \to List[Elem]$
- $reverse(nil) = nil$
- $reverse(cons(e, l)) = reverse(l) ++ cons(e, nil)$

end

spec LIST_WITH_ORDER [TOTAL_ORDER] =
LIST [**sort** *Elem*]
then **ops** $insert : Elem \times List[Elem] \to List[Elem]$;
$order[\_\_ \leq \_\_] : List[Elem] \to List[Elem]$
$\forall x, y : Elem;\ l : List[Elem]$
- $order[\_\_ \leq \_\_](nil) = nil$
- $order[\_\_ \leq \_\_](cons(x, l)) = insert(x, order[\_\_ \leq \_\_](l))$
- $insert(x, nil) = cons(x, nil)$
- $x \leq y \Rightarrow insert(x, cons(y, l)) = cons(x, insert(y, l))$
- $\neg(x \leq y) \Rightarrow insert(x, cons(y, l)) = cons(y, insert(x, l))$

hide *insert*
end

spec ORDERED_NAT =
NAT
then **preds** $\_\_ \leq \_\_ ,\ \_\_ \geq \_\_ : Nat \times Nat$
$\forall m, n : Nat$
- $zero \leq n$
- $\neg(succ\ m \leq zero)$
- $succ\ m \leq succ\ n \Leftrightarrow m \leq n$
- $m \geq n \Leftrightarrow n \leq m$

end

spec NAT_LIST_WITH_REVERSE_ORDERS =
LIST_WITH_ORDER [ORDERED_NAT **fit** $Elem \mapsto Nat, \_\_ \leq \_\_ \mapsto \_\_ \leq \_\_$]
and
LIST_WITH_ORDER [ORDERED_NAT **fit** $Elem \mapsto Nat, \_\_ \leq \_\_ \mapsto \_\_ \geq \_\_$]
then
$\forall l : List[Nat]$ • $order[\_\_ \geq \_\_](l) = reverse(order[\_\_ \leq \_\_](l))$
end

spec NON_EMPTY_LIST [ELEM] =
free type $NeList[Elem] ::= sort\ Elem \mid \_\_\ \_\_(Elem;\ NeList[Elem])$
ops $first : NeList[Elem] \to Elem$;
$rest : NeList[Elem] \to ? NeList[Elem]$
$\forall e : Elem;\ l : NeList[Elem]$
- $first(e) = e$
- $first(e\ l) = e$
- $\neg\, def\ rest(e)$
- $rest(e\ l) = l$

end

spec PATH =
NON_EMPTY_LIST [**sort** *Name*]
with $NeList[Name] \mapsto Path$,
$\_\_\ \_\_ \mapsto \_\_/\_\_$,
$first \mapsto the\_first\_name\_of\ \_\_$,
$rest \mapsto the\_last\_part\_of\ \_\_$
then ops $the\_first\_part\_of\ \_\_ : Path \to? Path$;
$the\_last\_name\_of\ \_\_ : Path \to Name$
$\forall n : Name;\ p : Path$
- $def\,(the\_first\_part\_of\ p) \Leftrightarrow \neg(p \in Name)$
- $\neg(p \in Name) \Rightarrow the\_first\_part\_of\ (n/p) \stackrel{e}{=} n/the\_first\_part\_of\ p$
- $p \in Name \Rightarrow the\_first\_part\_of\ (n/p) \stackrel{e}{=} n$
- $the\_last\_name\_of\ n = n$
- $the\_last\_name\_of\ (n/p) = the\_last\_name\_of\ p$

end

spec NAME = **sort** *Name*

%% ...

spec CONTENT = **sort** *Content*

%% ...

spec FILE =
NAME **and** CONTENT
then
generated type $File ::= < \_\_ \ . \ \_\_ > (the\_name\_of\ \_\_ : Name;\ the\_content\_of\ \_\_ : Content)$
end

view NAT_AS_ELEM : ELEM **to** NAT =
$Elem \mapsto Nat$

view LIST_AS_ELEM [ELEM] : ELEM **to** LIST [ELEM] =
$Elem \mapsto List[Elem]$

spec LIST_OF_LIST_OF_LIST_OF_NAT =
LIST [**view** LIST_AS_ELEM [**view** LIST_AS_ELEM [**view** NAT_AS_ELEM]]]

view ORDERED_NAT_AS_TOTAL_ORDER :
TOTAL_ORDER **to** ORDERED_NAT =

%{ ORDERED_NAT_AS_TOTAL_ORDER can be seen as the requirement that ORDERED_NAT indeed specifies a partial order. Thus defining the view would be significant even if the following instantiation were to be omitted. }%
$Elem \mapsto Nat$

spec NAT_LIST_WITH_ORDER =
LIST_WITH_ORDER [**view** ORDERED_NAT_AS_TOTAL_ORDER]

spec BOUNDED_LIST [ELEM] [**op** *bound* : *Nat*] **given** NAT =
LIST [ELEM] **and** ORDERED_NAT
then **op** $length : List[Elem] \to Nat$
$\forall e : Elem;\ l : List[Elem]$
- $length(nil) = zero$
- $length(cons(e, l)) = succ\ length(l)$

sort $Bounded\_List[Elem] = \{l : List[Elem] \bullet length(l) \leq bound\}$
type $Bounded\_List[Elem] ::= nil \mid cons(first :? Elem;\ rest :? Bounded\_List[Elem])?$

%{ The properties of the operations on $Bounded\_List[Elem]$ are determined by their overloadings on $List[Elem]$. }%

end

spec BOUNDED_NAT_LIST [**op** *bound* : *Nat*] **given** NAT =
BOUNDED_LIST [**view** NAT_AS_ELEM] [**op** *bound* : *Nat*]

E.3 Architectural Specifications

library SAMPLE/ARCHITECTURAL

%{ The example at the end of this section illustrates the difference between the structure of specifications and the architectural specification of structure.

It is more efficient to implement successor in terms of (binary) addition, while it is easier to specify addition in terms of successor than in terms of binary addition. Thus, the structure of the implementation differs from the structure of the specification:

We have that EFFICIENT_ADD_NUM is a refinement of ADD_NUM. }%

from SAMPLE/SIMPLE **get** MONOID

spec NUM =
  sort *Num*
  ops $0 : Num$;
    $succ : Num \to Num$
end

spec NUM_MONOID =
  MONOID **with** $Elem \mapsto Num, n \mapsto 0, \_\_ * \_\_ \mapsto \_\_ + \_\_$

spec ADD_NUM =
  NUM **and** NUM_MONOID
then
  $\forall x, y : Num \bullet x + succ(y) = succ(x + y)$
end

spec ADD_NUM_EFFICIENTLY =
  generated type $Num ::= 0 \mid 1 \mid \_\_0(Num) \mid \_\_1(Num)$
  ops $\_\_ + \_\_ , \_\_ +\!+ \_\_ : Num \times Num \to Num$
    %{ $\_\_ + \_\_$ is binary addition; $\_\_ +\!+ \_\_$ is binary addition with carry. }%
  $\forall x, y : Num$
  - $0\ 0 = 0$
  - $x\ 0 + y\ 0 = (x + y)\ 0$
  - $x\ 0 + y\ 1 = (x + y)\ 1$
  - $x\ 1 + y\ 0 = (x + y)\ 1$
  - $x\ 1 + y\ 1 = (x +\!+ y)\ 0$
  - $0\ 1 = 1$
  - $x\ 0 +\!+ y\ 0 = (x + y)\ 1$
  - $x\ 0 +\!+ y\ 1 = (x +\!+ y)\ 0$
  - $x\ 1 +\!+ y\ 0 = (x +\!+ y)\ 0$
  - $x\ 1 +\!+ y\ 1 = (x +\!+ y)\ 1$
end

arch spec EFFICIENT_ADD_NUM =
units N : ADD_NUM_EFFICIENTLY;
  M : { **op** $succ(n : Num) : Num = n + 1$ } **given** N
result
  M **hide** $1, \_\_0, \_\_1, \_\_ +\!+ \_\_$
end