CASL

The Common Algebraic Specification Language

Semantics

CoFI Document: CASL/Semantics
Version: 1.1 October 16, 2002

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

This document is available in various formats from the CoFI *archives.*[1]

Abstract

This is the formal semantics of version 1.0.2 of CASL. Although it is self-contained, it is intended for readers who are already familiar with the main concepts of algebraic specification and with the CASL Language Summary. The structure of this document is deliberately identical to that of the CASL Language Summary to aid cross-reference.

[1] `ftp://ftp.brics.dk/Projects/CoFI/Documents/CASL/Semantics/`

Contents

Chapter 0

Introduction

This document is the final outcome of the CoFI Semantics Task Group's work on the formal semantics of CASL, as informally presented in version 1.0.2 of the CASL Language Summary [CoF01]. This is the second "free-standing" version of this document. Previous versions were in the form of annotations on the Language Summary, to ease checking that the formal semantics accurately reflects what is expressed less formally there.

0.1 Structure of this document

With the exception of this Introduction, the structure of this document is deliberately identical to the structure of the CASL Language Summary [CoF01] to aid cross-reference. As in the Language Summary, Part I (Chapters 1–4) 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 *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*. For ease of reference, Appendix A provides a complete grammar for the *abstract syntax* of the language, collecting together the fragments that are scattered throughout the rest of the document. This is a repetition of the bulk of Appendix A of the CASL Language Summary [CoF01].

In each part, a chapter defining the *semantic concepts* underlying the kind of specification concerned is followed by a chapter presenting the abstract syntax of the associated CASL *language constructs* and defining their semantics. The concrete syntax is fully defined in the CASL Language Summary and this is not repeated here.

Brief informal summaries of the main concepts and constructs precede each block of formal definitions. This material, which is in boxes (like this paragraph) is provided as a supplement to the formal material; since it deliberately glosses over the details, it should *not* be regarded as definitive. There is other informal explanatory text in between the definitions, but nothing that is likely to be mistaken for a definition.

0.2 Notation

This section summarizes some of the basic notation used in the definitions below.

Sets. $\mathit{Set}(A)$ is the set of all subsets of A, and $\mathit{FinSet}(A)$ is the set of finite subsets of A. If S is a set then $|S|$ is the cardinality of S. *unit* denotes the singleton set $\{*\}$.

Tuples. $A_1 \times \cdots \times A_n$ is the set of n-tuples with jth component from A_j. Tuples are written like this: $(a_1, \ldots, a_n)$. Sometimes the parentheses are omitted, especially when tuples are used as subscripts or superscripts.

Sequences. $\mathit{FinSeq}(A)$ is the set of finite sequences of elements from A. Sequences are written like this: $\langle a_1, \ldots, a_n \rangle$. (This notation is different from that used in the Language Summary, where $\mathit{FinSeq}(A)$ is written A^* and $\langle a_1, \ldots, a_n \rangle$ is written $a_1 \ldots a_n$.) If $w = \langle a_1, \ldots, a_n \rangle$ then $|w| = n$.

Functions. $A \rightharpoonup B$ is the set of partial functions from A to B. $\mathit{Dom}(f) \subseteq A$ is the domain of $f : A \rightharpoonup B$. $A \to B$ is the set of total functions from A to B. Any total function $f : A \to B$ can also be regarded as a partial function $f : A' \rightharpoonup B$ for any $A' \supseteq A$, and any partial function $f : A \rightharpoonup B$ is a total function $f : \mathit{Dom}(f) \to B$. Functions are written like this: $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ or $\{x \mapsto x + 3 \mid x \in \mathit{Nat}\}$. We use the notation $f(x)$ for application of a function f to an argument x. Sometimes the parentheses are omitted, for instance when x is a tuple or a sequence. When f is an indexed family (a function from an index set to a domain of elements) we write f_x instead of $f(x)$. $A \xrightarrow{\text{fin}} B$ is the set of finite maps (i.e. partial functions with finite domain) from A to B.

Union and $\emptyset$. We use union ($\cup$) to combine semantic objects of various kinds, with the evident interpretation (e.g. component-wise union for tuples

and point-wise union for functions, that is $(f \cup g)(x) = f(x) \cup g(x)$ if f and g are set-valued functions such that $Dom(f) = Dom(g)$). More generally, for any set-valued functions f and g we take

$$(f \cup g)(x) = \begin{cases} f(x) \cup g(x) & \text{if } x \in Dom(f) \cap Dom(g) \\ f(x) & \text{if } x \in Dom(f) \setminus Dom(g) \\ g(x) & \text{if } x \in Dom(g) \setminus Dom(f) \\ \text{undefined} & \text{otherwise} \end{cases}$$

which gives $Dom(f \cup g) = Dom(f) \cup Dom(g)$. Similarly, $\emptyset$ is used for the empty object of various kinds (e.g. empty signature, empty function).

Disjoint union. $A \uplus B$ is the disjoint union of A and B. Injection from A and B to $A \uplus B$ is implicit, but sometimes we distinguish between $a \in A$ and $a \in A \uplus B$ (similarly for $b \in B$) by writing the latter as "a qua $A \uplus B$". We also use the "qua" notation for syntactic categories such as

```
OP-TYPE          ::=  TOTAL-OP-TYPE | PARTIAL-OP-TYPE
```

Function completion. We sometimes need to "complete" a function f with $Dom(f) \subseteq S$ to give a function with domain S by mapping values in $S \setminus Dom(f)$ to an appropriate neutral value. In particular, if f is a set-valued function, we define $complete(f, S) = f \cup \{x \mapsto \emptyset \mid x \in S \setminus Dom(f)\}$.

Categories. Some elementary category theory is used in places. A suitable introduction is [Pie91]. The category of sets is denoted **Set** and the (quasi)category of categories is denoted **Cat**.[1] We use the notation $f \circ g$ for (applicative order) composition of morphisms in a category. In **Set** this gives $(f \circ g)(x) = f(g(x))$.

Semantic domains. We define various semantic domains below. By convention, semantic domains containing "syntactic" objects (e.g. *Signature*) are in italics and semantic domains containing "semantic" objects (e.g. **Model**) are in boldface. Here is an example of a domain of "syntactic" objects:

$$(w, s) \text{ or } ws \quad \in \quad FunProfile = FinSeq(Sort) \times Sort$$

[1] There are foundational problems connected with the use of **Cat**—see [HS73] for how to solve them.

This defines the set *FunProfile* as the set of pairs having finite sequences of elements from *Sort* as first component and elements of *Sort* as second component. The metavariable *ws* ranges over elements of *FunProfile*. When we need to refer to the components of the pair we use the notation (w, s) instead, so w ranges over elements of $FinSeq(Sort)$ and s ranges over elements of *Sort*.

Validity. Typically, semantic domains are constructed from "more basic" domains together with some well-formedness requirements. Then a *valid* object is a value in the given set that satisfies the given requirements. Here is an example:

$$X \quad \in \quad Variables = Sort \xrightarrow{\text{fin}} FinSet(Var)$$

Requirements on an S-sorted set of variables X:

- $Dom(X) = S$
- for all $s, s' \in S$ such that $s \neq s'$, $X_s \cap X_{s'} = \emptyset$.

This says that a "set of variables" is a finite map taking elements of *Sort* to finite subsets of *Var*, while a "valid S-sorted set of variables" is a finite map of this kind that satisfies the two requirements given. Often, as in this case, validity of an object is relative to some other (valid) object, here a set S of sorts.

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

0.3 Static semantics and model semantics

The semantics of language constructs is given in two parts. The *static semantics* checks well-formedness of phrases of the abstract syntax and produces a "syntactic" object as result, failing to produce any result for ill-formed phrases. For example, for a many-sorted basic specification (see Chapter 2) the static semantics yields an enrichment containing the sorts, function symbols, predicate symbols and axioms that belong to the specification. A judgement of the static semantics has the following form: $context \vdash phrase \rhd result$. The *model semantics* provides the corresponding model-theoretic part of the semantics, and is intended to be applied only to phrases that are well-formed according to the static semantics. For a basic

specification, the model semantics yields a class of models. A judgement of the model semantics has the following form: *context* $\vdash$ *phrase* $\Rightarrow$ *result*.

A statically well-formed phrase may still be ill-formed according to the model semantics, and then no result is produced. This can never happen in the semantics of basic constructs but it can happen in the semantics of structured specifications and architectural specifications.

0.4 Semantic rules

The judgements of the static semantics and model semantics are defined inductively by means of rules in the style of Natural Semantics [Kah88]. For each phrase class we give a group of rules defining the semantics of the constructs in that class. The group is preceded by a specification of the "type" of the judgement(s) being defined. This is followed by pre-conditions on the "inputs" to the judgement(s) which, if satisfied, guarantee that the "outputs" satisfy the given post-conditions. Each of the rules should ensure that this is the case. For example, here is the section of the semantics for the phrase class `AXIOM-ITEMS` from Section 2.3 below, for which there is just one rule.

$$\boxed{\Sigma, X \vdash \texttt{AXIOM-ITEMS} \rhd \Psi}$$

X is required to be a valid set of variables over the sorts of Σ.
Ψ is a set of Σ-sentences.

$$\frac{\Sigma, X \vdash \texttt{AXIOM}_1 \rhd \psi_1 \qquad \cdots \qquad \Sigma, X \vdash \texttt{AXIOM}_n \rhd \psi_n}{\Sigma, X \vdash \texttt{axiom-items AXIOM}_1 \ \ldots \ \texttt{AXIOM}_n \rhd \{\psi_1, \ldots, \psi_n\}}$$

The "type" of the judgement is $\Sigma, X \vdash$ `AXIOM-ITEMS` $\rhd \Psi$. Intuitively, this says that in the local environment Σ with declared variables X, a phrase `AXIOM-ITEMS` yields a set Ψ of sentences. The pre-condition on the "inputs" is the requirement that X be a valid set of variables over the sorts of Σ. (The requirement that Σ itself be valid is implicit—use of a metavariable always refers to a valid object of the relevant kind.) The post-condition on the "output" is the assertion that Ψ will then be a set of Σ-sentences. It is easy to see that the given rule satisfies the pre/post-condition: if Σ, X satisfy the pre-condition then the post-condition associated with `AXIOM` guarantees that all of $\psi_1, \ldots, \psi_n$ will be Σ-sentences, and Ψ here is just $\{\psi_1, \ldots, \psi_n\}$.

Rules in the static semantics and model semantics have the form

$$\frac{\alpha_1 \qquad \cdots \qquad \alpha_n}{\beta}$$

where the conclusion β is a judgement and each premise α_j is either a judgement or a side-condition. When all the judgements occurring in all rules are positive (i.e. not negated) then the rules unambiguously define a family of relations via the usual notion of derivation tree, or equivalently as the smallest family of relations that is closed under the rules. Conclusions are always positive but there are situations in which negative premises are convenient. These are potentially problematic for at least two reasons: first, there may be *no* family of relations that is closed under the rules; second, there may be no *smallest* family of relations that is closed under the rules. It follows that care is required in situations where the natural choice of rules would involve negative premises. One way out is to simultaneously define a relation and its negation using rules with positive premises only, as in Section 1.4 below. Another is via the use of stratification to ensure the absence of dangerous circularities, cf. "negation by failure" in logic programming [Prz88], as in Section 2.3.3. See [vG96] for further discussion.

When a syntactic category C is defined as the disjoint union of other syntactic categories $C_1, \ldots, C_n$, rules that merely translate a judgement for C_1 etc. to a judgement for C are elided. Here is a schematic example of the kind of rules that are elided, for the static semantics:

$$\frac{\mathit{context} \vdash \mathit{phrase} \rhd \mathit{result}}{\mathit{context} \vdash \mathit{phrase}\ \mathrm{qua}\ C \rhd \mathit{result}}$$

Whenever such a rule is elided there will be a statement to this effect in the rule's place.

0.5 Institution independence

CASL is the heart of a *family* of languages. *Sub-languages* of CASL are obtained by imposing syntactic or semantic restrictions, while *extensions* of CASL will be defined to support various paradigms and applications.

The features of CASL for defining structured specifications, architectural specifications and specification libraries do not depend on the details of the features for basic specifications, so this part of the design is orthogonal to the rest. As a consequence, sub-languages and extensions of CASL can be defined by restricting or extending the language of basic specifications without the need to reconsider or change the rest of the language. On a semantic level, this is reflected by giving the semantics in an "institution independent" style. The semantics of basic specifications defines an *institution* [GB92] for CASL—actually, a variant of the notion of institution called an *institution with symbols* [Mos00]— and the rest of the semantics is based on an arbitrary institution (with symbols).

0.6 Contributors

The formal semantics of each part of the Language Summary was written by one or more authors under the watchful gaze of a kibitzer.[2] The authors were responsible for actually doing the work, while the kibitzer was to serve as first reader, act as devil's advocate, push the authors to do the work, and perhaps jump in and help if needed. Authors and kibitzers were as follows:

Basic specifications: Don Sannella (kibitzer Hubert Baumeister)

Subsorting: Maura Cerioli and Anne Haxthausen (kibitzer Till Mossakowski)

Structured specifications: Hubert Baumeister and Till Mossakowski (kibitzer Andrzej Tarlecki)

Architectural specifications: Andrzej Tarlecki (kibitzer Don Sannella)

Libraries: Peter Mosses (kibitzer Till Mossakowski)

This document was assembled by Don Sannella. The CoFI Semantics Task Group is coordinated by Andrzej Tarlecki.

Alexandre Zamulin read drafts of all parts of this document and sent many useful comments and suggestions, for which the authors are extremely grateful. Special thanks to Piotr Hoffman for pointing out inadequacies in an earlier version of the semantics of architectural specifications.

[2] kibitzer, n. Meddlesome person, one who gives advice gratuitously; one who watches a game of cards from behind the players.

Part I

Basic Specifications

Chapter 1

Basic Concepts

The concepts underlying basic specifications in CASL are those involved in defining an *institution* [GB92] for CASL. The following elements are required:

- a category **Sig** of *signatures* Σ, with *signature morphisms* $\sigma : \Sigma \to \Sigma'$;
- a (contravariant) functor $\mathbf{Mod} : \mathbf{Sig}^{op} \to \mathbf{Cat}$ giving for each signature Σ a category $\mathbf{Mod}(\Sigma)$ of *models* over Σ, with *homomorphisms* between them, and for each signature morphism $\sigma : \Sigma \to \Sigma'$ a *reduct* functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ (usually written $.|_\sigma$) translating models and homomorphisms over Σ' to models and homomorphisms over Σ;
- a functor $\mathbf{Sen} : \mathbf{Sig} \to \mathbf{Set}$ giving for each signature Σ a set $\mathbf{Sen}(\Sigma)$ of *sentences* (or *axioms*) over Σ, and for each signature morphism $\sigma : \Sigma \to \Sigma'$ a *translation* function $\mathbf{Sen}(\sigma)$, usually written $\sigma(.)$, taking Σ-sentences to Σ'-sentences;
- a relation $\models$ of *satisfaction* between models and sentences over the same signature.

Satisfaction is required to be compatible with reducts of models and translation of sentences: for all $\psi \in \mathbf{Sen}(\Sigma)$ and $M' \in \mathbf{Mod}(\Sigma')$,

$$M'|_\sigma \models \psi \iff M' \models \sigma(\psi).$$

(Additional structure is required for Parts II and III, including a functor $|.| : \mathbf{Sig} \to \mathbf{Set}$ with certain properties which determines the set of *symbols* of any signature.)

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.

The rest of this chapter considers many-sorted basic specifications of the CASL specification framework, and indicates the underlying signatures, models, sentences, and satisfaction relation. 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* Σ consists of: a set of *sorts*; separate families of sets of *total* and *partial function symbols*, indexed by *function profile* (a sequence of *argument sorts* and a *result sort*—*constants* are treated as functions with no arguments); and a family of sets of *predicate* symbols, indexed by *predicate profile* (a sequence of argument sorts). Constants and functions are also referred to as *operations*.

The internal structure of identifiers used to identify sorts, functions and predicates is insignificant for the semantics of basic specifications, see Section 2.4. Following the Language Summary, we therefore leave this unspecified for now, promising that there will be no circularity when the definitions of the sets *Sort*, *FunName* and *PredName* are eventually provided:

$$\begin{array}{lcl} s & \in & \mathit{Sort} \\ f & \in & \mathit{FunName} \\ p & \in & \mathit{PredName} \end{array}$$

(In Section 2.1 the internal structure of sorts will be defined as `TOKEN-ID` and the internal structure of function and predicate symbols will be defined as `ID`.)

$$\begin{array}{rcl} S & \in & \mathit{SortSet} = \mathit{FinSet}(\mathit{Sort}) \\ (w, s) \text{ or } ws & \in & \mathit{FunProfile} = \mathit{FinSeq}(\mathit{Sort}) \times \mathit{Sort} \\ TF, PF & \in & \mathit{FunSet} = \mathit{FunProfile} \rightharpoonup \mathit{FinSet}(\mathit{FunName}) \\ w & \in & \mathit{PredProfile} = \mathit{FinSeq}(\mathit{Sort}) \\ P & \in & \mathit{PredSet} = \mathit{PredProfile} \rightharpoonup \mathit{FinSet}(\mathit{PredName}) \end{array}$$

For a set of total function symbols TF over S it is required that $\mathit{Dom}(TF) = \mathit{FinSeq}(S) \times S$ and that $TF_{ws} \neq \emptyset$ for only finitely many function profiles $ws \in \mathit{FinSeq}(S) \times S$, and similarly for a set of partial function symbols PF. For a set of predicate symbols P over S it is required that $\mathit{Dom}(P) = \mathit{FinSeq}(S)$ and that $P_w \neq \emptyset$ for only finitely many predicate profiles $w \in \mathit{FinSeq}(S)$.

$$\begin{array}{rcl} (S, TF, PF, P) & & \\ \text{or } \Sigma & \in & \mathit{Signature} = \\ & & \quad \mathit{SortSet} \times \mathit{FunSet} \times \mathit{FunSet} \times \mathit{PredSet} \end{array}$$

Requirements on a signature (S, TF, PF, P):

- TF and PF are sets of total resp. partial function symbols over S
- P is a set of predicate symbols over S
- for all $ws \in \mathit{FinSeq}(S) \times S$, $TF_{ws} \cap PF_{ws} = \emptyset$

(An alternative to the use of the separate signature components TF and PF would be a single component F with a totality marker, so e.g. $F : \mathit{FunProfile} \times \{\mathsf{total}, \mathsf{partial}\} \rightharpoonup \mathit{FinSet}(\mathit{FunName})$ cf. [Wag99].)

Later we will need signature *extensions* as well. These are signature fragments that are interpreted relative to some other signature. First we define signature fragments.

$$\begin{array}{rcl} (S, TF, PF, P) & \in & \mathit{SigFragment} = \\ & & \quad \mathit{SortSet} \times \mathit{FunSet} \times \mathit{FunSet} \times \mathit{PredSet} \end{array}$$

These are simply signatures minus the validity requirements.

Union of signature fragments is defined as follows:

$$\begin{array}{l} (S, TF, PF, P) \cup (S', TF', PF', P') = \\ \qquad \mathit{reconcile}(S \cup S', \ \mathit{complete}(TF \cup TF', \mathit{FinSeq}(S'') \times S''), \\ \qquad\qquad\qquad\qquad \mathit{complete}(PF \cup PF', \mathit{FinSeq}(S'') \times S''), \\ \qquad\qquad\qquad\qquad \mathit{complete}(P \cup P', \mathit{FinSeq}(S''))) \\ \text{where } S'' = S \cup S' \ \cup \mathit{sorts}(\mathit{Dom}(TF)) \cup \mathit{sorts}(\mathit{Dom}(PF)) \cup \mathit{sorts}(\mathit{Dom}(P)) \\ \qquad\qquad\qquad\qquad \cup \mathit{sorts}(\mathit{Dom}(TF')) \cup \mathit{sorts}(\mathit{Dom}(PF')) \cup \mathit{sorts}(\mathit{Dom}(P')) \\ \text{and } \mathit{reconcile}(S, TF, PF, P) = (S, TF, \{ws \mapsto PF_{ws} \setminus TF_{ws} \mid ws \in \mathit{Dom}(PF)\}, P) \end{array}$$

Here, $sorts(T)$ is the set of sorts appearing in function/predicate profiles in T. The idea of this definition is to give the same result as if signature fragments were defined as sets of individual sort/function/predicate declarations. Note that any signature is also a signature fragment so this definition also defines union of two signatures as well as union of a signature and a signature fragment. According to this definition, the union of two signatures will always be a signature with $S'' = S \cup S'$. When a function name is declared as both partial and total, the *reconcile* function causes it to be regarded as total in the union, as required in Sects. 2.1.2.1 and 6.1.3.

$$(S, TF, PF, P) \text{ or } \Delta \quad \in \quad Extension = SigFragment$$

A signature extension relative to a signature Σ is a signature fragment Δ such that $\Sigma \cup \Delta$ (the *target* of the signature extension) is a signature. This guarantees that all the sorts used for function and predicate profiles in Δ are declared in either Δ or Σ.

Proposition 1 *If $\Delta = (S, TF, PF, P)$ and $\Delta' = (S', TF', PF', P')$ are signature extensions relative to Σ then $\Delta \cup \Delta'$ is a signature extension relative to Σ.*

PROOF: *Straightforward.* □

A signature Σ is a *subsignature* of a signature Σ' if there is some extension Δ relative to Σ such that $\Sigma' = \Sigma \cup \Delta$. Note that this allows a function name to be a partial function symbol in Σ but a total function symbol in Σ'.

Symbols used to identify sorts, operations, and predicates may be *overloaded*. For example, it is possible that $f \in TF_{ws}$ and $f \in TF_{ws'}$ for $ws \neq ws'$, as well as $f \in S$. To ensure that there is no ambiguity in sentences at this level, 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 they are unambiguously determined by the context.)

$$\begin{aligned} f_{ws} &\in QualFunName = FunName \times FunProfile \\ p_w &\in QualPredName = PredName \times PredProfile \end{aligned}$$

Requirements on a qualified function name f_{ws} over $\Sigma = (S, TF, PF, P)$:

- $ws \in FinSeq(S) \times S$
- $f \in TF_{ws} \cup PF_{ws}$

Requirements on a qualified predicate name p_w over $\Sigma = (S, TF, PF, P)$:

- $w \in FinSeq(S)$

- $p \in P_w$

Following [Mos00], Parts II and III below require that we define a set *Sym* of *symbols* and a function $|.|$ taking any signature to the set of symbols it contains (in fact we need a functor $|.| : \mathbf{Sig} \to \mathbf{Set}$ having certain properties, see Prop. 4 below). Symbols are essentially just qualified function/predicate names together with sort names.

$$\begin{array}{rcl} & & Sym = \\ s & \in & Sort \uplus \\ f_{ws} & \in & QualFunName \uplus \\ p_w & \in & QualPredName \end{array}$$

If $\Sigma = (S, TF, PF, P)$, we define $|\Sigma| \subseteq Sym$ as follows:

$$\begin{aligned} |\Sigma| = S \cup \{f_{ws} \mid\ & ws \in FinSeq(S) \times S, f \in TF_{ws} \cup PF_{ws}\} \\ \cup \{p_w \mid\ & w \in FinSeq(S), p \in P_w\} \end{aligned}$$

A *many-sorted signature morphism* $\sigma : \Sigma \to \Sigma'$ maps symbols in Σ to symbols in Σ'. A partial function symbol may be mapped to a total function symbol, but not vice versa.

$$\begin{array}{rcl} \sigma^{\mathsf{S}} & \in & SMap = Sort \stackrel{\text{fin}}{\to} Sort \\ \sigma^{\mathsf{TF}} & \in & TFMap = FunProfile \rightharpoonup (FunName \stackrel{\text{fin}}{\to} FunName) \\ \sigma^{\mathsf{PF}} & \in & PFMap = FunProfile \rightharpoonup (FunName \stackrel{\text{fin}}{\to} FunName) \\ \sigma^{\mathsf{P}} & \in & PMap = PredProfile \rightharpoonup (PredName \stackrel{\text{fin}}{\to} PredName) \\ (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}}) : \Sigma \to \Sigma' & & \\ \text{or } \sigma : \Sigma \to \Sigma' & \in & SignatureMorphism = \\ & & \quad Signature \\ & & \quad \times SMap \times TFMap \times PFMap \times PMap \\ & & \quad \times Signature \end{array}$$

Requirements on a signature morphism $(\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}}) : (S, TF, PF, P) \to (S', TF', PF', P')$:

- $\sigma^{\mathsf{S}} : S \to S'$
- $Dom(\sigma^{\mathsf{TF}}) = Dom(\sigma^{\mathsf{PF}}) = FinSeq(S) \times S$
- for all $ws \in FinSeq(S) \times S$:
  - $\sigma^{\mathsf{TF}}_{ws} : TF_{ws} \to TF'_{\sigma^{\mathsf{S}}(ws)}$
  - $\sigma^{\mathsf{PF}}_{ws} : PF_{ws} \to TF'_{\sigma^{\mathsf{S}}(ws)} \cup PF'_{\sigma^{\mathsf{S}}(ws)}$

- $Dom(\sigma^{\mathsf{P}}) = FinSeq(S)$
- for all $w \in FinSeq(S)$, $\sigma^{\mathsf{P}}_w : P_w \to P'_{\sigma^{\mathsf{S}}(w)}$

where, for $w = \langle s_1, \ldots, s_n \rangle$, $\sigma^{\mathsf{S}}(w) = \langle \sigma^{\mathsf{S}}(s_1), \ldots, \sigma^{\mathsf{S}}(s_n) \rangle$ and $\sigma^{\mathsf{S}}(w, s) = (\sigma^{\mathsf{S}}(w), \sigma^{\mathsf{S}}(s))$. If Σ is a subsignature of Σ', we write $\Sigma \hookrightarrow \Sigma'$ for the evident signature morphism. Such a signature morphism is called a *signature inclusion*. Note that a signature extension Δ relative to Σ can be viewed more abstractly as the signature inclusion $\Sigma \hookrightarrow \Sigma \cup \Delta$. However, information about any re-declaration in Δ of symbols in Σ is lost by this abstraction. Therefore Δ is kept explicitly together with the signature inclusion in Part II.

If $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$ are signature morphisms, where $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$, $\rho = (\rho^{\mathsf{S}}, \rho^{\mathsf{TF}}, \rho^{\mathsf{PF}}, \rho^{\mathsf{P}})$ and $\Sigma = (S, TF, PF, P)$, then the composition $\rho \circ \sigma : \Sigma \to \Sigma''$ is the signature morphism $(\delta^{\mathsf{S}}, \delta^{\mathsf{TF}}, \delta^{\mathsf{PF}}, \delta^{\mathsf{P}})$ where

$$
\begin{aligned}
\delta^{\mathsf{S}} &= \rho^{\mathsf{S}} \circ \sigma^{\mathsf{S}}, \\
\delta^{\mathsf{TF}} &= \{ ws \mapsto \rho^{\mathsf{TF}}_{\sigma^{\mathsf{S}}(ws)} \circ \sigma^{\mathsf{TF}}_{ws} \mid ws \in FinSeq(S) \times S \}, \\
\delta^{\mathsf{PF}} &= \{ ws \mapsto (\rho^{\mathsf{TF}}_{\sigma^{\mathsf{S}}(ws)} \cup \rho^{\mathsf{PF}}_{\sigma^{\mathsf{S}}(ws)}) \circ \sigma^{\mathsf{PF}}_{ws} \mid ws \in FinSeq(S) \times S \}, \\
\delta^{\mathsf{P}} &= \{ w \mapsto \rho^{\mathsf{P}}_{\sigma^{\mathsf{S}}(w)} \circ \sigma^{\mathsf{P}}_{w} \mid w \in FinSeq(S) \}
\end{aligned}
$$

Identity morphisms id_Σ are obvious.

Proposition 2 *The composition of signature morphisms does indeed yield a signature morphism.*

PROOF: *In the definition of* δ^{PF}, $\rho^{\mathsf{TF}}_{\sigma^{\mathsf{S}}(ws)} \cup \rho^{\mathsf{PF}}_{\sigma^{\mathsf{S}}(ws)}$ *is a function because* $TF'_{\sigma^{\mathsf{S}}(ws)} \cap PF'_{\sigma^{\mathsf{S}}(ws)} = \emptyset$. *The rest of the proof is straightforward.* □

Proposition 3 *Signatures and signature morphisms form a finitely cocomplete category,* **Sig**.

PROOF: *It is easy to see that* **Sig** *is a category. Regarding finite cocompleteness, see [Mos98b] for a more general result.* □

If $\sigma : \Sigma \to \Sigma'$ is a signature morphism, where $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$, $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, then the function $|\sigma| : |\Sigma| \to |\Sigma'|$ is defined as follows:

$$
\begin{aligned}
|\sigma|(s) &= \sigma^{\mathsf{S}}(s) \qquad \text{for all } s \in S \\
|\sigma|(f_{ws}) &= \begin{cases} \sigma^{\mathsf{TF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} & \text{for } f \in TF_{ws} \\ \sigma^{\mathsf{PF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} & \text{for } f \in PF_{ws} \end{cases} \\
&\qquad \text{for all } ws \in FinSeq(S) \times S \\
|\sigma|(p_w) &= \sigma^{\mathsf{P}}_w(p)_{\sigma^{\mathsf{S}}(w)} \\
&\qquad \text{for all } w \in FinSeq(S) \text{ and } p \in P_w
\end{aligned}
$$

Proposition 4 $|.| : \mathbf{Sig} \to \mathbf{Set}$ *is a faithful functor.*

PROOF: *It is easy to see that* $|.|$ *is a functor. Faithfulness is also obvious:* $|\sigma|$ *(together with the partiality data in* Σ *and* Σ'*) carries no less information than* $\sigma : \Sigma \to \Sigma'$*.* □

Proposition 5 *A signature morphism* $\sigma : \Sigma \to \Sigma'$ *is a signature inclusion iff* $|\sigma|$ *is an inclusion of* $|\Sigma|$ *into* $|\Sigma'|$*.*

PROOF: *Straightforward. It is essential that in a signature inclusion* $\sigma : \Sigma \to \Sigma'$*, a function name may be a partial function symbol in* Σ *but a total function symbol in* Σ'*.* □

1.2 Models

For a many-sorted signature Σ, a *many-sorted model* $M \in \mathbf{Mod}(\Sigma)$ assigns a non-empty *carrier set* to each sort in Σ, a *partial* resp. *total function* to each partial resp. total function symbol, and a *predicate* to each predicate symbol. Requiring carriers to be non-empty simplifies deduction [GM82] and will allow axioms in specifications to be implicitly universally quantified, see Sect. 2.2.

$$
\begin{array}{rcl}
S^M(s) \text{ or } s^M & \in & \mathbf{Carrier} = \text{the class of all sets} \\
S^M & \in & \mathbf{Carriers} = \mathit{Sort} \stackrel{\text{fin}}{\to} \mathbf{Carrier} \\
F^M_{ws}(f) \text{ or } f^M & \in & \mathbf{PartialFun} = \text{the class of all partial functions} \\
F^M & \in & \mathbf{PartialFuns} = \mathit{FunProfile} \rightharpoonup (\mathit{FunName} \stackrel{\text{fin}}{\to} \mathbf{PartialFun}) \\
P^M_w(p) \text{ or } p^M & \in & \mathbf{Pred} = \text{the class of all predicates} \\
P^M & \in & \mathbf{Preds} = \mathit{PredProfile} \rightharpoonup (\mathit{PredName} \stackrel{\text{fin}}{\to} \mathbf{Pred}) \\
(S^M, F^M, P^M) & & \\
\text{or } M & \in & \mathbf{Model} = \mathbf{Carriers} \times \mathbf{PartialFuns} \times \mathbf{Preds} \\
\mathcal{M} & \in & \mathbf{ModelClass} = \mathit{Set}(\mathbf{Model})
\end{array}
$$

Requirements on a Σ-model $M = (S^M, F^M, P^M)$ for $\Sigma = (S, TF, PF, P)$:

- $\mathit{Dom}(S^M) = S$
- for all $s \in S$, $S^M(s) \neq \emptyset$
- $\mathit{Dom}(F^M) = \mathit{FinSeq}(S) \times S$
- for all $w \in \mathit{FinSeq}(S)$ and $s \in S$:
  - $\mathit{Dom}(F^M_{w,s}) = PF_{w,s} \cup TF_{w,s}$

  - for all $f \in TF_{w,s}$, $F^M_{w,s}(f) : w^M \to s^M$ (a *total* function)
  - for all $f \in PF_{w,s}$, $F^M_{w,s}(f) : w^M \rightharpoonup s^M$
- $Dom(P^M) = FinSeq(S)$
- for all $w \in FinSeq(S)$:
  - $Dom(P^M_w) = P_w$
  - for all $p \in P_w$, $P^M_w(p) \subseteq w^M$

where $\langle s_1, \ldots, s_n \rangle^M = s_1^M \times \cdots \times s_n^M$.

Every model in a Σ-model class $\mathcal{M}$ is required to be a valid Σ-model.

Given two Σ-models $M, M' \in \mathbf{Mod}(\Sigma)$, a *many-sorted homomorphism* $h : M \to M'$ maps the values in the carriers of M to values in the corresponding carriers of M' in such a way that the values of functions and their definedness is preserved, as well as the truth of predicates.

$$h : M \to M' \quad \in \quad \mathbf{Homomorphism} = \mathbf{Model} \times (Sort \xrightarrow{\text{fin}} \mathbf{PartialFun}) \times \mathbf{Model}$$

Requirements on a Σ-homomorphism $h : M \to M'$ for $\Sigma = (S, TF, PF, P)$:

- M and M' are valid Σ-models
- $Dom(h) = S$
- for all $s \in S$, $h_s : s^M \to s^{M'}$
- for all $w = \langle s_1, \ldots, s_n \rangle \in FinSeq(S)$, $s \in S$, $f \in TF_{w,s} \cup PF_{w,s}$ and $a_1 \in s_1^M, \ldots, a_n \in s_n^M$, if $f^M(a_1, \ldots, a_n)$ is defined then $f^{M'}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ is defined and $h_s(f^M(a_1, \ldots, a_n)) = f^{M'}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$
- for all $w = \langle s_1, \ldots, s_n \rangle \in FinSeq(S)$, $p \in P_w$ and $a_1 \in s_1^M, \ldots, a_n \in s_n^M$, if $(a_1, \ldots, a_n) \in p^M$ then $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in p^{M'}$.

Composition of homomorphisms is as usual: if $h : M \to M'$ and $h' : M' \to M''$ are Σ-homomorphisms for $\Sigma = (S, TF, PF, P)$, then the Σ-homomorphism $h' \circ h : M \to M''$ is given by $(h' \circ h)_s = h'_s \circ h_s$ for all $s \in S$. Identity homomorphisms are S-sorted identity functions.

Proposition 6 *The composition $h' \circ h : M \to M''$ is indeed a Σ-homomorphism.*

PROOF: *Routine.* □

Proposition 7 *Σ-models and Σ-homomorphisms form a category,* $\mathbf{Mod}(\Sigma)$.

PROOF: *Easy.* □

A signature morphism $\sigma : \Sigma \to \Sigma'$ determines the *many-sorted reduct* of each Σ'-model resp. Σ'-homomorphism to a Σ-model resp. Σ-homomorphism, defined by interpreting symbols of Σ in the reduct in the same way that their images under σ are interpreted.

Let $M' = (S^{M'}, F^{M'}, P^{M'})$ be a Σ'-model and let $\sigma : \Sigma \to \Sigma'$ be a signature morphism where $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$ and $\Sigma = (S, TF, PF, P)$. The *reduct* of M' with respect to σ is the Σ-model $M'|_\sigma = (S^M, F^M, P^M)$ defined as follows:

$$
\begin{aligned}
S^M &= S^{M'} \circ \sigma^{\mathsf{S}} \\
F^M_{ws}(f) &= \begin{cases} F^{M'}_{\sigma^{\mathsf{S}}(ws)}(\sigma^{\mathsf{TF}}_{ws}(f)) & \text{if } f \in TF_{ws} \\ F^{M'}_{\sigma^{\mathsf{S}}(ws)}(\sigma^{\mathsf{PF}}_{ws}(f)) & \text{if } f \in PF_{ws} \end{cases} \\
P^M_w(p) &= P^{M'}_{\sigma^{\mathsf{S}}(w)}(\sigma^{\mathsf{P}}_w(p))
\end{aligned}
$$

Proposition 8 *If $\sigma : \Sigma \to \Sigma'$ is a signature morphism and M' is a Σ'-model then $M'|_\sigma$ is indeed a Σ-model.*

PROOF: *Routine.* □

Suppose that Σ is a subsignature of Σ', so there is a signature inclusion $\Sigma \hookrightarrow \Sigma'$. Then we sometimes write $M'|_\Sigma$ as an abbreviation for $M'|_{\Sigma \hookrightarrow \Sigma'}$, and we say that a Σ'-model M' *extends* a Σ-model M if $M'|_\Sigma = M$.

Let $h' : M1' \to M2'$ be a Σ'-homomorphism and let $\sigma : \Sigma \to \Sigma'$ be a signature morphism where $\Sigma = (S, TF, PF, P)$ and $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$. The *reduct* of h' with respect to σ is the Σ-homomorphism $h'|_\sigma : M1'|_\sigma \to M2'|_\sigma$ defined by $(h'|_\sigma)_s = h'_{\sigma^{\mathsf{S}}(s)}$ for all $s \in S$. If Σ is a subsignature of Σ' then we sometimes write $h'|_\Sigma$ as an abbreviation for $h'|_{\Sigma \hookrightarrow \Sigma'}$.

Proposition 9 *If $\sigma : \Sigma \to \Sigma'$ is a signature morphism and $h' : M1' \to M2'$ is a Σ'-homomorphism then $h'|_\sigma : M1'|_\sigma \to M2'|_\sigma$ is indeed a Σ-homomorphism.*

PROOF: *Easy.* □

Proposition 10 *Reduct of models and homomorphisms extends* **Mod** *to a finitely continuous functor* $\mathbf{Mod} : \mathbf{Sig}^{op} \to \mathbf{Cat}$ *(i.e.* **Mod** *takes finite colimits in* **Sig** *to limits in* **Cat***).*

PROOF: *It is easy to see that* **Mod** *is a functor. For continuity, see [Mos98a] for a sketch of the proof of a more general result; cf. [CGRW95].* □

Let $h : M \to M'$ be a Σ-homomorphism. If there is a Σ-homomorphism $h^{-1} : M' \to M$ such that $h \circ h^{-1}$ is the identity on M' and $h^{-1} \circ h$ is the

identity on M then h is a Σ-*isomorphism* and we write $M \cong M'$.

1.3 Sentences

The *many-sorted terms* on a signature Σ and a set X of variables consist of variables from X together with applications of qualified function symbols 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 explicit qualifications to be omitted when they are determined by the context.

Following the Language Summary, we leave the syntax of variables (Var) unspecified for now. It will be defined in Section 2.2 below.

$$\begin{array}{rcl} x & \in & Var \\ X & \in & Variables = Sort \xrightarrow{\text{fin}} FinSet(Var) \\ x_s & \in & QualVarName = Var \times Sort \end{array}$$

Requirements on an S-sorted set of variables X:

- $Dom(X) = S$
- for all $s, s' \in S$ such that $s \neq s'$, $X_s \cap X_{s'} = \emptyset$.

In a qualified variable name x_s, it is required that $s \in S$.

We write $X + \{x_s\}$ for the $(S \cup \{s\})$-sorted set of variables such that

$$(X + \{x_s\})_s = \begin{cases} X_s \cup \{x\} & \text{if } s \in Dom(X) \\ \{x\} & \text{otherwise} \end{cases}$$

and $(X + \{x_s\})_{s'} = X_{s'} \setminus \{x\}$ for $s' \in S$ such that $s' \neq s$. We write $X + X'$ for the extension of this to arbitrary S'-sorted sets of variables X'.

Proposition 11 *If X is valid for S and X' is valid for S' then $X + X'$ is valid for $S \cup S'$.*

PROOF: *Easy.* □

If $x^i \neq x^j$ for all $1 \leq i \neq j \leq n$ then we use $\{x^1_{s_1}, \ldots, x^n_{s_n}\}$ to abbreviate $\{x^1_{s_1}\} + \cdots + \{x^n_{s_n}\}$. (The pre-condition means that the order is immaterial, as the set notation suggests.)

The definitions of fully-qualified terms and formulae are mutually recursive.

$$\begin{array}{rcl} t & \in & FQTerm = \\ x_s & \in & QualVarName \uplus \\ f_{ws}\langle t_1, \ldots, t_n \rangle & \in & QualFunName \times FinSeq(FQTerm) \uplus \\ \varphi \rightarrow t \mid t' & \in & Formula \times FQTerm \times FQTerm \end{array}$$

For any $t \in FQTerm$, define $sort(t) \in Sort$ as follows:

$$\begin{array}{rcl} sort(x_s) & = & s \\ sort(f_{w,s}\langle t_1, \ldots, t_n\rangle) & = & s \\ sort(\varphi \rightarrow t' \mid t'') & = & \begin{cases} sort(t') & \text{if } sort(t') = sort(t'') \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}$$

Requirements on a fully-qualified Σ-term t over an S-sorted set of variables X, for $\Sigma = (S, TF, PF, P)$:

- if t is x_s, then x_s is valid for S and $x \in X_s$
- if t is $f_{w,s}\langle t_1, \ldots, t_n\rangle$, then:
  - $f_{w,s}$ is a valid qualified function name over Σ
  - $t_1, \ldots, t_n$ are valid fully-qualified Σ-terms over X
  - $|w| = n$
  - $w = \langle sort(t_1), \ldots, sort(t_n)\rangle$
- if t is $\varphi \rightarrow t' \mid t''$, then:
  - φ is a valid Σ-formula over X
  - t' and t'' are valid fully-qualified Σ-terms over X
  - $sort(t') = sort(t'')$

The fully-qualified term $\varphi \rightarrow t \mid t'$ is only needed to deal with the conditional term construct, see Section 2.3.4 below. An alternative is to deal with these by transformation as described in Section 2.3.4.5 of the CASL Language Summary. Then fully-qualified terms of the form $\varphi \rightarrow t \mid t'$ are not required. Since these terms are non-standard, this might be a better choice when it comes to questions like the design of a proof system for CASL.

The *many-sorted sentences* in $\mathbf{Sen}(\Sigma)$ are the usual closed many-sorted first-order logic formulae, built from atomic formulae (application of qualified predicate symbols to argument terms of appropriate sorts, assertions about the definedness of fully-qualified terms, and existential and strong equations between fully-qualified terms of the same sort) using quantification and logical connectives. Predicate application, existential equations, implication and universal quantification are taken as primitive, the other forms being regarded as derived.

$$\begin{array}{rcl}
\varphi & \in & \mathit{Formula} = \\
p_w\langle t_1, \ldots, t_n\rangle & \in & \mathit{QualPredName} \times \mathit{FinSeq}(\mathit{FQTerm}) \uplus \\
t \stackrel{\mathrm{e}}{=} t' & \in & \mathit{FQTerm} \times \mathit{FQTerm} \uplus \\
\mathit{false} & \in & \mathit{unit} \uplus \\
\varphi \Rightarrow \varphi' & \in & \mathit{Formula} \times \mathit{Formula} \uplus \\
\forall x_s.\varphi & \in & \mathit{QualVarName} \times \mathit{Formula}
\end{array}$$

Requirements on a Σ-formula φ over an S-sorted set of variables X, for $\Sigma = (S, TF, PF, P)$:

- if φ is $p_w\langle t_1, \ldots, t_n\rangle$, then:
  - p_w is a valid qualified predicate name over Σ
  - $t_1, \ldots, t_n$ are valid fully-qualified Σ-terms over X
  - $|w| = n$
  - $w = \langle \mathit{sort}(t_1), \ldots, \mathit{sort}(t_n)\rangle$
- if φ is $t \stackrel{\mathrm{e}}{=} t'$, then:
  - t and t' are valid fully-qualified Σ-terms over X
  - $\mathit{sort}(t) = \mathit{sort}(t')$
- if φ is $\varphi' \Rightarrow \varphi''$, then φ' and φ'' are valid Σ-formulae over X
- if φ is $\forall x_s.\varphi'$, then:
  - x_s is valid for S
  - φ' is a valid Σ-formula over $X + \{x_s\}$

Abbreviations are defined as follows:

$$\begin{array}{rll}
\neg\varphi & \text{abbreviates} & \varphi \Rightarrow \mathit{false} \\
\varphi \vee \varphi' & \text{abbreviates} & (\neg\varphi) \Rightarrow \varphi' \\
\varphi \wedge \varphi' & \text{abbreviates} & \neg(\neg\varphi \vee \neg\varphi') \\
\varphi \Leftrightarrow \varphi' & \text{abbreviates} & (\varphi \Rightarrow \varphi') \wedge (\varphi' \Rightarrow \varphi) \\
\mathit{true} & \text{abbreviates} & \neg\mathit{false} \\
D(t) & \text{abbreviates} & t \stackrel{\mathrm{e}}{=} t \\
t \stackrel{\mathrm{s}}{=} t' & \text{abbreviates} & (t \stackrel{\mathrm{e}}{=} t \Rightarrow t \stackrel{\mathrm{e}}{=} t') \wedge (t' \stackrel{\mathrm{e}}{=} t' \Rightarrow t \stackrel{\mathrm{e}}{=} t') \\
\forall\{x^1_{s^1}, \ldots, x^n_{s^n}\}.\varphi & \text{abbreviates} & \forall x^1_{s^1}.\cdots\forall x^n_{s^n}.\varphi \\
\exists X.\varphi & \text{abbreviates} & \neg(\forall X.\neg\varphi) \\
\exists! X.\varphi & \text{abbreviates} & \exists X.(\varphi \wedge \forall \hat{X}.(\varphi[\hat{X}/X] \Rightarrow X \stackrel{\mathrm{e}}{=} \hat{X}))
\end{array}$$

where in the last clause the variables $\hat{X}$ are variants of X chosen to avoid all variable clashes, $\varphi[\hat{X}/X]$ is substitution, and $X \stackrel{\mathrm{e}}{=} \hat{X}$ abbreviates the evident conjunction of equations.

If $n = 0$ then $\varphi_1 \wedge \cdots \wedge \varphi_n$ means *true*, $\varphi_1 \vee \cdots \vee \varphi_n$ means *false*, and $\forall x^1_{s1}. \cdots \forall x^n_{sn}.\varphi$ means φ. (This is metanotation: ellipses are not included in the syntax of sentences.)

Let φ be a Σ-formula over X and let $(\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$ be a signature morphism $\sigma : \Sigma \to \Sigma'$ where $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$. Let X' be the S'-sorted set of variables such that $X'_{s'} = \bigcup_{\sigma^{\mathsf{S}}(s)=s'} X_s$ for all $s' \in S'$. The *translation* of φ along σ is the Σ'-formula $\sigma(\varphi)$ over X' obtained by replacing each qualified variable name x_s in φ by $x_{\sigma^{\mathsf{S}}(s)}$, each qualified function name f_{ws} such that $f \in TF_{ws}$ by $\sigma^{\mathsf{TF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)}$, each qualified function name f_{ws} such that $f \in PF_{ws}$ by $\sigma^{\mathsf{PF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)}$, and each qualified predicate name p_w by $\sigma^{\mathsf{P}}_w(p)_{\sigma^{\mathsf{S}}(w)}$.

Proposition 12 *If $\sigma : \Sigma \to \Sigma'$ is a signature morphism and φ is a Σ-formula over X then $\sigma(\varphi)$ is indeed a Σ'-formula over X'. If X is empty then so is X'.*

PROOF: *Straightforward.* □

The sentences **Sen**(Σ) also include *sort-generation constraints*, used to require that models are reachable on a subset of sorts.

$$(S, F, \sigma) \quad \in \quad \mathit{Constraint} = \mathit{SortSet} \times \mathit{FunSet} \times \mathit{SignatureMorphism}$$

Requirements on a Σ-constraint (S, F, σ):

- $\sigma : \Sigma' \to \Sigma$ where $\Sigma' = (S', TF', PF', P')$, and then:
- $S \subseteq S'$
- $\mathit{Dom}(F) = \mathit{FinSeq}(S') \times S'$
- for all $ws \in \mathit{FinSeq}(S') \times S'$, $F_{ws} \subseteq TF'_{ws} \cup PF'_{ws}$

Let $\sigma' : \Sigma \to \Sigma''$ be a signature morphism. The *translation* of a Σ-constraint (S, F, σ) along σ' is the Σ''-constraint $\sigma'(S, F, \sigma) = (S, F, \sigma' \circ \sigma)$.

Proposition 13 *Translating a Σ-constraint along $\sigma : \Sigma \to \Sigma''$ gives a Σ''-constraint.*

PROOF: *Obvious.* □

We use the abbreviation (S, F) for the Σ-constraint (S, F, id_Σ). Only constraints of this kind are introduced by CASL specifications, see Sections 2.1.4.2 and 2.1.5. Constraints with non-identity third components arise only when constraints introduced by CASL specifications are translated along signature morphisms.

$$\psi \quad \in \quad \mathit{Sentence} = \mathit{Formula} \uplus \mathit{Constraint}$$

Requirements on a Σ-sentence ψ:

- if ψ is a formula, it is required to be a valid Σ-formula over the empty set of variables
- if ψ is a constraint, it is required to be a valid Σ-constraint

Proposition 14 *The mapping from signatures Σ to sets of Σ-sentences, together with translation of sentences along signature morphisms, gives a functor* $\mathbf{Sen} : \mathbf{Sig} \to \mathbf{Set}$.

PROOF: *The requirement that variables cannot be overloaded is crucial because it allows the translated sets of variables X' above to be formed without the use of disjoint union. Given this, the proof is straightforward.* □

$$(\Delta, \Psi) \quad \in \quad \mathit{Enrichment} = \mathit{Extension} \times \mathit{FinSet}(\mathit{Sentence})$$

Requirements on an enrichment (Δ, Ψ) relative to a signature Σ:

- Δ is a signature extension relative to Σ
- Each $\psi \in \Psi$ is a $\Sigma \cup \Delta$-sentence

1.4 Satisfaction

The satisfaction of a Σ-formula in a Σ-model is determined as usual by the holding of its atomic formulae w.r.t. assignments of values to all the variables that occur in them. The value of a term may be undefined, due to the presence of partial functions. Note, however, that the satisfaction of sentences is 2-valued.
A predicate application holds iff the values of all its argument terms are defined and give a tuple that belongs to the predicate. A definedness assertion holds iff the value of the term is defined. 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.

$$\rho \quad \in \quad \mathbf{Assignment} = \mathit{Sort} \xrightarrow{\text{fin}} \mathbf{PartialFun}$$

Let $\Sigma = (S, TF, PF, P)$ be a signature, M a Σ-model, and X an S-sorted set of variables. Requirements on an assignment ρ of X into M:

- $Dom(\rho) = S$
- for all $s \in S$, $\rho_s : X_s \to s^M$

If $a \in s^M$ then we write $\rho[x_s \mapsto a]$ for the assignment of $X + \{x_s\}$ into M such that $\rho[x_s \mapsto a]_s(x) = a$, $\rho[x_s \mapsto a]_s(x') = \rho_s(x')$ for $x' \neq x$, and $\rho[x_s \mapsto a]_{s'}(x') = \rho_{s'}(x')$ for $s' \neq s$ and $x' \neq x$.

We now simultaneously define three things inductively by means of inference rules:

- the *value* $[\![t]\!]_\rho$ of a fully-qualified Σ-term t over X in a Σ-model M with respect to an assignment ρ of X into M;
- *satisfaction* of a Σ-formula φ over X by a Σ-model M under an assignment ρ of X into M, written $M \models_\rho \varphi$; and
- *non-satisfaction* of φ by M under ρ, written $M \not\models_\rho \varphi$.

We define both $\models$ and $\not\models$ so as to avoid negative occurrences of $\models$ in its own definition.

$$\frac{\rho_s(x_s) = a}{[\![x_s]\!]_\rho = a}$$

$$\frac{[\![t_1]\!]_\rho = a_1 \quad \cdots \quad [\![t_n]\!]_\rho = a_n \qquad f^M(a_1, \ldots, a_n) = a}{[\![f_{ws}\langle t_1, \ldots, t_n\rangle]\!]_\rho = a}$$

According to this rule, the value of $f_{ws}\langle t_1, \ldots, t_n\rangle$ is defined only if the values of $t_1, \ldots, t_n$ are defined and the resulting tuple of values is in $Dom(f^M)$.

$$\frac{M \models_\rho \varphi \qquad [\![t]\!]_\rho = a}{[\![\varphi \to t \mid t']\!]_\rho = a}$$

$$\frac{M \not\models_\rho \varphi \qquad [\![t']\!]_\rho = a'}{[\![\varphi \to t \mid t']\!]_\rho = a'}$$

$$\frac{[\![t_1]\!]_\rho = a_1 \quad \cdots \quad [\![t_n]\!]_\rho = a_n \qquad (a_1, \ldots, a_n) \in p^M}{M \models_\rho p_w\langle t_1, \ldots, t_n\rangle}$$

$$\frac{[\![t_j]\!]_\rho \text{ not defined for some } 1 \leq j \leq n}{M \not\models_\rho p_w\langle t_1, \ldots, t_n\rangle}$$

$$\frac{[\![t_1]\!]_\rho = a_1 \quad \cdots \quad [\![t_n]\!]_\rho = a_n \qquad (a_1, \ldots, a_n) \notin p^M}{M \not\models_\rho p_w\langle t_1, \ldots, t_n\rangle}$$

$$\frac{[\![t]\!]_\rho = a \qquad [\![t']\!]_\rho = a}{M \models_\rho t \stackrel{e}{=} t'}$$

$$\frac{[\![t]\!]_\rho \text{ not defined}}{M \not\models_\rho t \stackrel{e}{=} t'}$$

$$\frac{[\![t']\!]_\rho \text{ not defined}}{M \not\models_\rho t \stackrel{e}{=} t'}$$

$$\frac{[\![t]\!]_\rho = a \qquad [\![t']\!]_\rho = a' \qquad a \neq a'}{M \not\models_\rho t \stackrel{e}{=} t'}$$

$$\frac{}{M \not\models_\rho \mathit{false}}$$

$$\frac{M \not\models_\rho \varphi}{M \models_\rho \varphi \Rightarrow \varphi'}$$

$$\frac{M \models_\rho \varphi'}{M \models_\rho \varphi \Rightarrow \varphi'}$$

$$\frac{M \models_\rho \varphi \qquad M \not\models_\rho \varphi'}{M \not\models_\rho \varphi \Rightarrow \varphi'}$$

$$\frac{M \models_{\rho[x_s \mapsto a]} \varphi \text{ for all } a \in s^M}{M \models_\rho \forall x_s . \varphi}$$

$$\frac{a \in s^M \qquad M \not\models_{\rho[x_s \mapsto a]} \varphi}{M \not\models_\rho \forall x_s . \varphi}$$

Proposition 15 *$M \models_\rho \varphi$ iff $\neg M \not\models_\rho \varphi$ and $M \not\models_\rho \varphi$ iff $\neg M \models_\rho \varphi$.*

PROOF: *By induction on the structure of φ (simultaneously for the two forward implications, and simultaneously for the two backward implications).* □

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 from the values in the carriers of sorts not in S. Then $M \models (S, F, \sigma)$ iff $M|_\sigma \models (S, F)$.

Suppose M is a Σ-model and (S, F, σ) is a Σ-constraint with $\sigma : \Sigma' \to \Sigma$. Then M *satisfies* (S, F, σ), written $M \models (S, F, \sigma)$, if the carriers of $M|_\sigma$ of the sorts in S are generated by the function symbols in F, i.e. for every sort

$s \in S$ and every value $a \in s^{M|_\sigma}$, there is a Σ'-term t containing only function symbols from F and variables of sorts not in S such that $[\![t]\!]_\rho = a$ for some assignment ρ into $M|_\sigma$.

A Σ-model M *satisfies* a Σ-sentence ψ, written $M \models \psi$, if:

- ψ is a formula φ and $M \models_\emptyset \varphi$, where $\emptyset$ is here the empty assignment from the empty set of variables
- ψ is a constraint (S, F, σ) and $M \models (S, F, \sigma)$

We write $M \not\models \psi$ for $\neg M \models \psi$.

Proposition 16 *Satisfaction is compatible with reducts of models and translation of sentences: if $\sigma : \Sigma \to \Sigma'$ is a signature morphism, ψ is a Σ-sentence and M' is a Σ'-model, then*

$$M'|_\sigma \models \psi \qquad \textit{iff} \qquad M' \models \sigma(\psi)$$

PROOF: *See Sect. 3.1 of [Mos].* □

Theorem 17 Sig, **Mod**, **Sen** *and $\models$ form an institution [GB92].* **Sig** *is finitely cocomplete and* **Mod** *supports amalgamation of models and homomorphisms.*

PROOF: *Directly from Props. 3, 10, 14 and 16.* □

Proposition 18 *Satisfaction is preserved and reflected by isomorphisms: if M, M' are Σ-models such that $M \cong M'$ and ψ is a Σ-sentence, then $M \models \psi$ iff $M' \models \psi$.*

PROOF: *Straightforward.* □

Chapter 2

Basic Constructs

This chapter gives the abstract syntax of the constructs of *many-sorted* basic specifications, and defines their intended interpretation. Well-formedness of phrases of the abstract syntax is defined by the *static semantics*, which produces a "syntactic" object as result and fails to produce any result for ill-formed phrases. The *model semantics*, which yields a class of models as result, provides the corresponding model-theoretic part of the semantics. In this chapter, only basic specifications themselves (phrases of type `BASIC-SPEC`) are given both static and model semantics; other phrase types are given only static semantics. In this particular case, the result of the static semantics fully determines the result of the model semantics, but that is not the case in other parts of CASL.

A many-sorted basic specification `BASIC-SPEC` is a sequence of `BASIC-ITEMS` constructs. It determines an enrichment containing the sorts, function symbols, predicate symbols and axioms that belong to the specification; these may make reference to symbols in the local environment. This enrichment in turn determines a class of models.

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

$$\Sigma \vdash \texttt{BASIC-SPEC} \rhd (\Delta, \Psi) \qquad\qquad \Sigma, \mathcal{M} \vdash \texttt{BASIC-SPEC} \Rightarrow \mathcal{M}'$$

(Δ, Ψ) is an enrichment relative to Σ. $\mathcal{M}$ is required to be a model class over Σ. Each model in $\mathcal{M}'$ is a valid $\Sigma \cup \Delta$-model that extends a model in $\mathcal{M}$ and satisfies Ψ.

As will become clear in Part II, one use of basic specifications in CASL is in extending existing specifications. Such a basic specification will often make

reference to the sorts, function symbols and predicate symbols of the existing specification (the *local environment*), for instance to declare a new function taking an argument of an existing sort. This context is captured by the signature Σ in the above judgements, with the Σ-models in $\mathcal{M}$ giving all the possible interpretations of these symbols. In contrast, variable declarations are local to basic specifications.

$$\frac{\begin{array}{c}\Sigma, \emptyset \vdash \texttt{BASIC-ITEMS}_1 \rhd (\Delta_1, \Psi_1), X_1 \\ \cdots \\ \Sigma \cup \Delta_1 \cup \cdots \cup \Delta_{n-1}, X_1 + \cdots + X_{n-1} \vdash \texttt{BASIC-ITEMS}_n \rhd (\Delta_n, \Psi_n), X_n\end{array}}{\Sigma \vdash \texttt{basic-spec BASIC-ITEMS}_1 \ \ldots \ \texttt{BASIC-ITEMS}_n \rhd (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n)}$$

Making the incremental information from all the preceding `BASIC-ITEMS` available to the next one in sequence gives linear visibility. The use of $+$ to combine variable sets means that a later declaration of a given variable will override an earlier declaration of the same variable.

$$\frac{\Sigma \vdash \texttt{basic-spec BASIC-ITEMS*} \rhd (\Delta, \Psi)}{\begin{array}{l}\Sigma, \mathcal{M} \vdash \texttt{basic-spec BASIC-ITEMS*} \Rightarrow \\ \quad \{\Sigma \cup \Delta\text{-model } M' \mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta} \in \mathcal{M} \text{ and } \forall \psi \in \Psi . M' \models \psi\}\end{array}}$$

Each `BASIC-ITEMS` construct determines part of a signature and/or some sentences (except for `VAR-ITEMS`, which merely declares some global variables). 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.

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

$$\boxed{\Sigma, X \vdash \texttt{BASIC-ITEMS} \rhd (\Delta, \Psi), X'}$$

X is required to be a valid set of variables over the sorts of Σ. (Δ, Ψ) is an enrichment relative to Σ, and X' is a valid set of variables over the sorts of $\Sigma \cup \Delta$. (Actually, X' will be a valid set of variables over the sorts of Σ since there happens to be no construct of `BASIC-ITEMS` that both declares variables and introduces signature components.)

$$\frac{\Sigma \vdash \texttt{SIG-ITEMS} \rhd (\Delta, \Delta', \Psi)}{\Sigma, X \vdash \texttt{SIG-ITEMS} \text{ qua } \texttt{BASIC-ITEMS} \rhd (\Delta \cup \Delta', \Psi), \emptyset}$$

$$\frac{\Sigma \vdash \texttt{FREE-DATATYPE} \rhd (\Delta, \Psi)}{\Sigma, X \vdash \texttt{FREE-DATATYPE} \text{ qua } \texttt{BASIC-ITEMS} \rhd (\Delta, \Psi), \emptyset}$$

$$\frac{\Sigma \vdash \texttt{SORT-GEN} \rhd (\Delta, \Psi)}{\Sigma, X \vdash \texttt{SORT-GEN}\ \text{qua}\ \texttt{BASIC-ITEMS} \rhd (\Delta, \Psi), \emptyset}$$

$$\frac{S \vdash \texttt{VAR-ITEMS} \rhd X'}{(S, TF, PF, P), X \vdash \texttt{VAR-ITEMS}\ \text{qua}\ \texttt{BASIC-ITEMS} \rhd (\emptyset, \emptyset), X'}$$

$$\frac{\Sigma, X \vdash \texttt{LOCAL-VAR-AXIOMS} \rhd \Psi}{\Sigma, X \vdash \texttt{LOCAL-VAR-AXIOMS}\ \text{qua}\ \texttt{BASIC-ITEMS} \rhd (\emptyset, \Psi), \emptyset}$$

$$\frac{\Sigma, X \vdash \texttt{AXIOM-ITEMS} \rhd \Psi}{\Sigma, X \vdash \texttt{AXIOM-ITEMS}\ \text{qua}\ \texttt{BASIC-ITEMS} \rhd (\emptyset, \Psi), \emptyset}$$

2.1 Signature Declarations

A list SORT-ITEMS of sort declarations determines some sorts. A list OP-ITEMS of operation declarations/definitions determines some operation symbols, and possibly some sentences; similarly for predicate declarations/definitions PRED-ITEMS. A list DATATYPE-ITEMS of datatype declarations determines some sorts together with some constructor and (optional) selector operations, and sentences defining the selector operations.

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

$$\boxed{\Sigma \vdash \texttt{SIG-ITEMS} \rhd (\Delta, \Delta', \Psi)}$$

$(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ.

Here, Δ' are the selectors declared by DATATYPE-DECLs in SIG-ITEMS and Δ is everything else declared in SIG-ITEMS. These need to be kept separate here because they are treated differently by the sort-generation construct, see Section 2.1.5.

$$\frac{\Sigma \vdash \texttt{SORT-ITEMS} \rhd (\Delta, \Psi)}{\Sigma \vdash \texttt{SORT-ITEMS}\ \text{qua}\ \texttt{SIG-ITEMS} \rhd (\Delta, \emptyset, \Psi)}$$

$$\frac{\Sigma \vdash \texttt{OP-ITEMS} \rhd (\Delta, \Psi)}{\Sigma \vdash \texttt{OP-ITEMS}\ \text{qua}\ \texttt{SIG-ITEMS} \rhd (\Delta, \emptyset, \Psi)}$$

$$\frac{\Sigma \vdash \texttt{PRED-ITEMS} \rhd (\Delta, \Psi)}{\Sigma \vdash \texttt{PRED-ITEMS}\ \text{qua}\ \texttt{SIG-ITEMS} \rhd (\Delta, \emptyset, \Psi)}$$

$$\frac{\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W}{\Sigma \vdash \texttt{DATATYPE-ITEMS}\ \text{qua}\ \texttt{SIG-ITEMS} \rhd (\Delta, \Delta', \Psi)}$$

2.1.1 Sorts

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

$$\boxed{\Sigma \vdash \texttt{SORT-ITEMS} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\Sigma \vdash \texttt{SORT-ITEM}_1 \rhd (\Delta_1, \Psi_1) \qquad \cdots \qquad \Sigma \vdash \texttt{SORT-ITEM}_n \rhd (\Delta_n, \Psi_n)}{\Sigma \vdash \texttt{sort-items SORT-ITEM}_1 \ \ldots \ \texttt{SORT-ITEM}_n \rhd (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n)}$$

The only reason why we have $\Sigma \vdash \texttt{SORT-ITEMS} \rhd (\Delta, \Psi)$ rather than simply $\vdash \texttt{SORT-ITEMS} \rhd S$ (and similarly for SORT-ITEM below) is to accommodate the extension to subsorts in Chapters 3–4 where Δ will include a subsorting relation and Ψ will include axioms for defined subsorts.

$$\boxed{\Sigma \vdash \texttt{SORT-ITEM} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\vdash \texttt{SORT-DECL} \rhd S}{\Sigma \vdash \texttt{SORT-DECL qua SORT-ITEM} \rhd ((S, \emptyset, \emptyset, \emptyset), \emptyset)}$$

2.1.1.1 Sort Declarations

A sort declaration SORT-DECL declares each of the sorts given.

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

$$\boxed{\vdash \texttt{SORT-DECL} \rhd S}$$

$$\frac{}{\vdash \texttt{sort-decl } s_1 \ \ldots \ s_n \rhd \{s_1, \ldots, s_n\}}$$

As promised in Section 1.1, we now define the universe *Sort* of sort names.

$$Sort = \texttt{TOKEN-ID}$$

2.1.2 Operations

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

$$\boxed{\Sigma \vdash \texttt{OP-ITEMS} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{OP-ITEM}_1 \rhd (\Delta_1, \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \cdots \cup \Delta_{n-1} \vdash \texttt{OP-ITEM}_n \rhd (\Delta_n, \Psi_n)\end{array}}{\Sigma \vdash \texttt{op-items OP-ITEM}_1 \ \ldots \ \texttt{OP-ITEM}_n \rhd (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n)}$$

Making the signature extensions from all the preceding OP-ITEMs available to the next one in sequence gives linear visibility. This is required here for the sake of UNIT-OP-ATTR attributes and operation definitions, both of which may refer to previously-declared function symbols.

$$\boxed{\Sigma \vdash \texttt{OP-ITEM} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

Rules elided (see Sect. 0.4).

2.1.2.1 Operation Declarations

An operation declaration OP-DECL declares each given operation name as a total or partial operation, with profile as specified, and having the given attributes. If an operation is declared both as total and as partial with the same profile, the resulting signature only contains the total operation.

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

As promised in Section 1.1, we now define the universe *FunName* of operation names.

$$\mathit{FunName} = \texttt{ID}$$

(Recall from Section 1.1 that operations are also referred to as functions, hence *FunName*.)

$$\boxed{\Sigma \vdash \texttt{OP-DECL} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \qquad ws = (\langle s_1, \ldots, s_m \rangle, s) \\ \{s_1, \ldots, s_m, s\} \subseteq S \qquad \Delta = (\emptyset, \{ws \mapsto \{f^1, \ldots, f^n\}\}, \emptyset, \emptyset) \\ \Sigma \cup \Delta, f^1_{ws} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{11} \quad \cdots \quad \Sigma \cup \Delta, f^n_{ws} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{n1} \\ \ldots \qquad\qquad\qquad\qquad \ldots \\ \Sigma \cup \Delta, f^1_{ws} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{1p} \quad \cdots \quad \Sigma \cup \Delta, f^n_{ws} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{np}\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{op-decl } f^1 \ \ldots \ f^n \\ \quad (\texttt{total-op-type } (\texttt{sort-list } s_1 \ \ldots \ s_m)\ s) \\ \quad \texttt{OP-ATTR}_1 \ \ldots \ \texttt{OP-ATTR}_p \rhd \\ \qquad (\Delta, (\Psi_{11} \cup \cdots \cup \Psi_{n1}) \cup \cdots \cup (\Psi_{1p} \cup \cdots \cup \Psi_{np}))\end{array}}$$

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \qquad ws = (\langle s_1, \ldots, s_m \rangle, s) \\ \{s_1, \ldots, s_m, s\} \subseteq S \qquad \Delta = (\emptyset, \emptyset, \{ws \mapsto \{f^1, \ldots, f^n\}\}, \emptyset) \\ \Sigma \cup \Delta, f^1_{ws} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{11} \quad \cdots \quad \Sigma \cup \Delta, f^n_{ws} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{n1} \\ \ldots \qquad\qquad\qquad\qquad \ldots \\ \Sigma \cup \Delta, f^1_{ws} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{1p} \quad \cdots \quad \Sigma \cup \Delta, f^n_{ws} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{np}\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{op-decl } f^1 \ \ldots \ f^n \\ \quad (\texttt{partial-op-type } (\texttt{sort-list } s_1 \ \ldots \ s_m)\ s) \\ \quad \texttt{OP-ATTR}_1 \ \ldots \ \texttt{OP-ATTR}_p \rhd \\ \qquad (\Delta, (\Psi_{11} \cup \cdots \cup \Psi_{n1}) \cup \cdots \cup (\Psi_{1p} \cup \cdots \cup \Psi_{np}))\end{array}}$$

The use of $\cup$ to combine the extensions produced by these rules, in the rules for `OP-ITEMS` and `BASIC-SPEC`, ensure that when an operation is declared both as total and as partial with the same profile, the resulting signature only contains the total operation. This is the purpose of the *reconcile* function in the definition of union, see Section 1.1.

Operation Attributes

Operation attributes assert that the operations being declared (which must be binary) have certain common properties: *associativity*, *commutativity*, *idempotency* and/or having a *unit*. (This can also be used to add attributes to operations that have previously been declared without them.)

```
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
```

$$\boxed{\Sigma, f_{ws} \vdash \texttt{OP-ATTR} \rhd \Psi}$$

f_{ws} is required to be a qualified function name over Σ. Ψ is a set of Σ-sentences.

$$\frac{ws = (\langle s, s\rangle, s)}{\Sigma, f_{ws} \vdash \texttt{assoc-op-attr} \rhd \{\forall x_s.\forall y_s.\forall z_s.f_{ws}\langle x_s, f_{ws}\langle y_s, z_s\rangle\rangle \stackrel{s}{=} f_{ws}\langle f_{ws}\langle x_s, y_s\rangle, z_s\rangle\}}$$

$$\frac{ws = (\langle s, s\rangle, s)}{\Sigma, f_{ws} \vdash \texttt{comm-op-attr} \rhd \{\forall x_s.\forall y_s.f_{ws}\langle x_s, y_s\rangle \stackrel{s}{=} f_{ws}\langle y_s, x_s\rangle\}}$$

$$\frac{ws = (\langle s, s\rangle, s)}{\Sigma, f_{ws} \vdash \texttt{idem-op-attr} \rhd \{\forall x_s.f_{ws}\langle x_s, x_s\rangle \stackrel{s}{=} x_s\}}$$

$$\frac{ws = (\langle s, s\rangle, s) \qquad \Sigma, \emptyset \vdash \texttt{TERM} \rhd t \qquad sort(t) = s}{\Sigma, f_{ws} \vdash \texttt{unit-op-attr TERM} \rhd \{\forall x_s.f_{ws}\langle t, x_s\rangle \stackrel{s}{=} x_s, \forall x_s.f_{ws}\langle x_s, t\rangle \stackrel{s}{=} x_s\}}$$

2.1.2.2 Operation Definitions

A total or partial operation may be *defined* at the same time as it is declared, by giving its value (when applied to a list of argument variables) as a term. The operation name may occur in the term, and may have *any* interpretation satisfying the equation—not necessarily the least fixed point.

```
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
```

$$\boxed{\Sigma \vdash \texttt{OP-DEFN} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \qquad S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \\ w = \langle s_1, \ldots, s_n \rangle \qquad s \in S \\ \Delta = (\emptyset, \{(w, s) \mapsto \{f\}\}, \emptyset, \emptyset) \qquad X = \mathit{complete}(\{x^1_{s_1}, \ldots, x^n_{s_n}\}, S) \\ \Sigma \cup \Delta, X \vdash \texttt{TERM} \rhd t \qquad \mathit{sort}(t) = s\end{array}}{\begin{array}{r}\Sigma \vdash \texttt{op-defn}\ f\ (\texttt{total-op-head ARG-DECL*}\ s)\ \texttt{TERM} \rhd \\ (\Delta, \{\forall X. f_{w,s} \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \stackrel{\mathrm{s}}{=} t\})\end{array}}$$

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \qquad S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \\ w = \langle s_1, \ldots, s_n \rangle \qquad s \in S \\ \Delta = (\emptyset, \emptyset, \{(w, s) \mapsto \{f\}\}, \emptyset) \qquad X = \mathit{complete}(\{x^1_{s_1}, \ldots, x^n_{s_n}\}, S) \\ \Sigma \cup \Delta, X \vdash \texttt{TERM} \rhd t \qquad \mathit{sort}(t) = s\end{array}}{\begin{array}{r}\Sigma \vdash \texttt{op-defn}\ f\ (\texttt{partial-op-head ARG-DECL*}\ s)\ \texttt{TERM} \rhd \\ (\Delta, \{\forall X. f_{w,s} \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \stackrel{\mathrm{s}}{=} t\})\end{array}}$$

$$\boxed{S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle}$$

Each $x^i_{s_i}$ is a qualified variable name over S, and $x^i \neq x^j$ for all $1 \leq i \neq j \leq n$.

$$\frac{S \vdash \texttt{ARG-DECL}_1 \rhd \langle x^{11}, \ldots, x^{1m_1} \rangle, s_1 \qquad \cdots \qquad S \vdash \texttt{ARG-DECL}_p \rhd \langle x^{p1}, \ldots, x^{pm_p} \rangle, s_p \quad \{x^{i1}, \ldots, x^{im_i}\} \cap \{x^{j1}, \ldots, x^{jm_j}\} = \emptyset \text{ for all } 1 \leq i \neq j \leq p}{S \vdash \texttt{ARG-DECL}_1\ \ldots\ \texttt{ARG-DECL}_p \rhd \langle x^{11}_{s_1}, \ldots, x^{1m_1}_{s_1}, \ \ldots, \ x^{p1}_{s_p}, \ldots, x^{pm_p}_{s_p} \rangle}$$

$$\boxed{S \vdash \texttt{ARG-DECL} \rhd \langle x_1, \ldots, x_n \rangle, s}$$

s is a sort in S and $x_i \neq x_j$ for all $1 \leq i \neq j \leq n$.

$$\frac{s \in S \qquad x_i \neq x_j \text{ for all } 1 \leq i \neq j \leq n}{S \vdash \texttt{arg-decl}\ x_1\ \ldots\ x_n\ s \rhd \langle x_1, \ldots, x_n \rangle, s}$$

2.1.3 Predicates

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

$$\boxed{\Sigma \vdash \texttt{PRED-ITEMS} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{PRED-ITEM}_1 \rhd (\Delta_1, \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \cdots \cup \Delta_{n-1} \vdash \texttt{PRED-ITEM}_n \rhd (\Delta_n, \Psi_n)\end{array}}{\Sigma \vdash \texttt{pred-items PRED-ITEM}_1 \ \ldots \ \texttt{PRED-ITEM}_n \rhd (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n)}$$

Making the signature extensions from all the preceding PRED-ITEMs available to the next one in sequence gives linear visibility. This is required here for the sake of predicate definitions which may refer to previously-declared predicate symbols.

$$\boxed{\Sigma \vdash \texttt{PRED-ITEM} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\Sigma \vdash \texttt{PRED-DECL} \rhd \Delta}{\Sigma \vdash \texttt{PRED-DECL qua PRED-ITEM} \rhd (\Delta, \emptyset)}$$

Rule for PRED-DEFN qua PRED-ITEM elided.

As promised in Section 1.1, we now define the universe *PredName* of predicate names.

$$\mathit{PredName} = \texttt{ID}$$

2.1.3.1 Predicate Declarations

A predicate declaration PRED-DECL declares each given predicate name, with profile as specified.

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

$$\boxed{\Sigma \vdash \texttt{PRED-DECL} \rhd \Delta}$$

Δ is a signature extension relative to Σ.

$$\frac{S \vdash \texttt{PRED-TYPE} \rhd w}{(S, TF, PF, P) \vdash \texttt{pred-decl}\ p_1 \ \ldots \ p_n\ \texttt{PRED-TYPE} \rhd (\emptyset, \emptyset, \emptyset, \{w \mapsto \{p_1, \ldots, p_n\}\})}$$

Predicate Types

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

$$\boxed{S \vdash \texttt{PRED-TYPE} \rhd w}$$

All the sorts in w are in S.

$$\frac{\{s_1, \ldots, s_n\} \subseteq S}{S \vdash \texttt{pred-type}\ (\texttt{sort-list}\ s_1\ \ldots\ s_n) \rhd \langle s_1, \ldots, s_n \rangle}$$

2.1.3.2 Predicate Definitions

A predicate may be *defined* at the same time as it is declared, by asserting its equivalence with a formula. The predicate name may occur in the formula, and may have *any* interpretation satisfying the equivalence.

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

$$\boxed{\Sigma \vdash \texttt{PRED-DEFN} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \qquad S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \\ w = \langle s_1, \ldots, s_n \rangle \qquad \Delta = (\emptyset, \emptyset, \emptyset, \{w \mapsto \{p\}\}) \\ X = \mathit{complete}(\{x^1_{s_1}, \ldots, x^n_{s_n}\}, S) \qquad \Sigma \cup \Delta, X \vdash \texttt{FORMULA} \rhd \varphi\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{pred-defn}\ p\ (\texttt{pred-head ARG-DECL*})\ \texttt{FORMULA} \rhd \\ \qquad\qquad (\Delta, \{\forall X. p_w \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \Leftrightarrow \varphi\})\end{array}}$$

2.1.4 Datatypes

The order of the datatype declarations in a list DATATYPE-ITEMS is *not* significant: there is *non-linear visibility* of the declared sorts. A list of datatype declarations must not declare a function symbol both as a constructor and selector with the same profiles.

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

The semantics of datatype declarations is by far the most complicated part of the semantics of basic specifications. Before proceeding, here is an overview.

Some examples of the results produced for free datatypes are given just before Section 2.1.5; these should be helpful in understanding that part of the semantics, and working backwards to see why these results are produced should help in clarifying the semantics of non-free datatypes.

The main judgements are $\Sigma \vdash$ `DATATYPE-ITEMS` $\rhd (\Delta, \Delta', \Psi), W$ and $\Sigma \vdash$ `FREE-DATATYPE` $\rhd (\Delta, \Psi)$. The former, for a list of datatype declarations, is subordinate to the latter, for free datatypes. It is also subordinate to the judgement for `SIG-ITEMS`, when used to declare non-free datatypes. All of the information in its result is required to determine the semantics of free datatypes but some is not required in the case of non-free datatypes. The judgements that are subordinate to `DATATYPE-ITEMS` collect information about declared sorts, constructors and selectors and check that various restrictions are satisfied. A complicating factor in these is non-linear visibility at the `DATATYPE-ITEMS` level.

All metavariables are used consistently in the judgement for `DATATYPE-ITEMS` and all of its subordinate judgements. Here is a summary of what they stand for, where these results are formed, and where and for what they are required.

1. Δ contains the sorts and constructors declared by the list of datatype declarations. It is formed by the rules for `DATATYPE-DECL` (sorts) and `ALTERNATIVE` (constructors).

2. Δ' contains the declared selectors. It is formed by the rules for `COMPONENTS`. The selectors need to be kept separate from the other signature components for the sake of the disjointness condition in the `DATATYPE-ITEMS` rule, to generate sentences in the rule for `FREE-DATATYPE`, and, in the case of a non-free datatypes, to produce the result for `SIG-ITEMS`, where a separation is required for the sake of `SORT-GEN` where selectors receive special treatment.

3. Ψ contains sentences defining the value of each selector on the values produced by the corresponding constructor. It is formed by the rules for `COMPONENTS`.

4. W is a finite map taking each constructor name in Δ to the corresponding set of partial selectors from Δ' (or to $\emptyset$ in case there are none). It is formed in the rule for `DATATYPE-DECL` using information from the `ALTERNATIVE`s it contains. W is needed in the rule for `FREE-DATATYPE` to generate sentences that require a partial selector to return an undefined result when applied to a value produced by a constructor for which it has not been declared.

$$\boxed{\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W}$$

$(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ and W is a finite map taking qualified function names over $\Sigma \cup \Delta$ from Δ to sets of qualified function names over $\Sigma \cup \Delta \cup \Delta'$ from Δ'.

$$\frac{\begin{array}{c}\Sigma' \vdash \texttt{DATATYPE-DECL}_1 \rhd (\Delta_1, \Delta_1', \Psi_1), W_1 \\ \cdots \\ \Sigma' \vdash \texttt{DATATYPE-DECL}_n \rhd (\Delta_n, \Delta_n', \Psi_n), W_n \\ \textit{disjoint-functions}(\Delta_1 \cup \cdots \cup \Delta_n, \Delta_1' \cup \cdots \cup \Delta_n') \\ \Sigma' = \Sigma \cup \Delta_1 \cup \Delta_1' \cup \cdots \cup \Delta_n \cup \Delta_n'\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{datatype-items DATATYPE-DECL}_1 \ldots \texttt{DATATYPE-DECL}_n \rhd \\ \quad (\Delta_1 \cup \cdots \cup \Delta_n, \Delta_1' \cup \cdots \cup \Delta_n', \Psi_1 \cup \cdots \cup \Psi_n), W_1 \cup \cdots \cup W_n\end{array}}$$

where *disjoint-functions*$((S, TF, PF, P), (S', TF', PF', P'))$ means

$$\begin{array}{l}\forall ws \in Dom(TF \cup PF) \cap Dom(TF' \cup PF'). \\ \quad (TF \cup PF)(ws) \cap (TF' \cup PF')(ws) = \emptyset\end{array}$$

The "recursion" in the premises of this rule is what provides non-linear visibility, making the order of the DATATYPE-DECLs not significant. In the subordinate judgements, it is important to remember that the context will already include the signature extensions being produced. The disjointness premise implements the requirement[1] that a list of datatype declarations must not declare a function symbol both as a constructor and selector with the same profile.

2.1.4.1 Datatype Declarations

A datatype declaration DATATYPE-DECL declares the given sort, and for each given alternative construct the given constructor and selector operations, and determines sentences asserting the expected relationship between selectors and constructors.

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

$$\boxed{\Sigma \vdash \texttt{DATATYPE-DECL} \rhd (\Delta, \Delta', \Psi), W}$$

$(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ and W is a finite map taking qualified function names over $\Sigma \cup \Delta$ from Δ to sets of qualified function names over $\Sigma \cup \Delta \cup \Delta'$ from Δ'.

See the beginning of Section 2.1.4 for an explanation of the meaning of Δ, Δ', Ψ and W in this part of the semantics.

[1] In the Language Summary this requirement is in Sect. 2.1.4.1, under Components.

$$\frac{\begin{array}{c}(S,\, TF,\, PF,\, P) = \Sigma \\ \Sigma, s \vdash \texttt{ALTERNATIVE}_1 \rhd (\Delta_1, \Delta'_1, \Psi_1) \qquad (S_1,\, TF_1,\, PF_1,\, P_1) = \Delta_1 \\ \{(w_1, s) \mapsto \{f^1\}\} = TF_1 \cup PF_1 \qquad (S'_1,\, TF'_1,\, PF'_1,\, P'_1) = \Delta'_1 \\ \cdots \\ \Sigma, s \vdash \texttt{ALTERNATIVE}_n \rhd (\Delta_n, \Delta'_n, \Psi_n) \qquad (S_n,\, TF_n,\, PF_n,\, P_n) = \Delta_n \\ \{(w_n, s) \mapsto \{f^n\}\} = TF_n \cup PF_n \qquad (S'_n,\, TF'_n,\, PF'_n,\, P'_n) = \Delta'_n\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{datatype-decl}\ s\ \texttt{ALTERNATIVE}_1\ \texttt{ALTERNATIVE}_2\ \ldots\ \texttt{ALTERNATIVE}_n \rhd \\ \qquad ((\{s\}, \emptyset, \emptyset, \emptyset) \cup \Delta_1 \cup \cdots \cup \Delta_n, \Delta'_1 \cup \cdots \cup \Delta'_n, \Psi_1 \cup \cdots \cup \Psi_n), \\ \qquad \{f^1_{w_1,s} \mapsto \{g_{\langle s\rangle, s'} \mid s' \in S \cup \{s\}, g \in PF'_1(\langle s\rangle, s')\}\} \\ \qquad \cup \cdots \cup \\ \qquad \{f^n_{w_n,s} \mapsto \{g_{\langle s\rangle, s'} \mid s' \in S \cup \{s\}, g \in PF'_n(\langle s\rangle, s')\}\}\end{array}}$$

Note that s will be a sort in Σ because of non-linear visibility.

Alternatives

An `ALTERNATIVE` declares a constructor operation. Each component specifies one or more argument sorts for the profile; the result sort is the one declared by the enclosing datatype declaration.

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

$$\boxed{\Sigma, s \vdash \texttt{ALTERNATIVE} \rhd (\Delta, \Delta', \Psi)}$$

s is required to be a sort in Σ. $(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ where Δ contains exactly one function and this function has result sort s.

See the beginning of Section 2.1.4 for an explanation of the meaning of Δ, Δ' and Ψ in this part of the semantics. In this judgement, s is the sort declared by the enclosing `DATATYPE-DECL`, the function in Δ is the constructor for this alternative, and Δ' are its selectors.

$$\frac{\begin{array}{c}\Sigma, f, ws, 1 \vdash \texttt{COMPONENTS}_1 \rhd \langle s_{11}, \ldots, s_{1m_1}\rangle, (\Delta'_1, \Psi_1) \\ \cdots \\ \Sigma, f, ws, 1 + m_1 + \cdots + m_{n-1} \vdash \texttt{COMPONENTS}_n \rhd \langle s_{n1}, \ldots, s_{nm_n}\rangle, (\Delta'_n, \Psi_n) \\ ws = (\langle s_{11}, \ldots, s_{1m_1}, \ \ldots, \ s_{n1}, \ldots, s_{nm_n}\rangle, s)\end{array}}{\begin{array}{l}\Sigma, s \vdash \texttt{total-construct}\ f\ \texttt{COMPONENTS}_1\ \ldots\ \texttt{COMPONENTS}_n \rhd \\ \qquad ((\emptyset, \{ws \mapsto \{f\}\}, \emptyset, \emptyset), \Delta'_1 \cup \cdots \cup \Delta'_n, \Psi_1 \cup \cdots \cup \Psi_n)\end{array}}$$

Note that f will be a total function in Σ because of non-linear visibility.

$$\frac{\begin{array}{c}\Sigma, f, ws, 1 \vdash \texttt{COMPONENTS}_1 \rhd \langle s_{11}, \ldots, s_{1m_1}\rangle, (\Delta'_1, \Psi_1) \\ \cdots \\ \Sigma, f, ws, 1 + m_1 + \cdots + m_{n-1} \vdash \texttt{COMPONENTS}_n \rhd \langle s_{n1}, \ldots, s_{nm_n}\rangle, (\Delta'_n, \Psi_n) \\ ws = (\langle s_{11}, \ldots, s_{1m_1}, \ \ldots, \ s_{n1}, \ldots, s_{nm_n}\rangle, s)\end{array}}{\begin{array}{l}\Sigma, s \vdash \texttt{partial-construct}\ f\ \texttt{COMPONENTS}_1\ \ldots\ \texttt{COMPONENTS}_n \rhd \\ \qquad ((\emptyset, \emptyset, \{ws \mapsto \{f\}\}, \emptyset), \Delta'_1 \cup \cdots \cup \Delta'_n, \Psi_1 \cup \cdots \cup \Psi_n)\end{array}}$$

Note that f will be a partial function in Σ because of non-linear visibility.

Components

Each COMPONENTS construct specifies one or more argument sorts for the constructor operation declared by the enclosing ALTERNATIVE, and optionally some selector operations with sentences determining their result on values produced by that constructor. All sorts used must be declared in the local environment.

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

$$\boxed{\Sigma, f, ws, m \vdash \texttt{COMPONENTS} \rhd w', (\Delta', \Psi)}$$

f is required to be a function name in Σ with profile $ws = (\langle s_1, \ldots, s_n\rangle, s)$ and $1 \leq m \leq n$. w' is a non-empty sequence of sorts in Σ and (Δ', Ψ) is an enrichment relative to Σ.

See the beginning of Section 2.1.4 for an explanation of the meaning of Δ' and Ψ in this part of the semantics. In this judgement, f is the constructor declared by the enclosing ALTERNATIVE, s is the sort declared by the enclosing DATATYPE-DECL, and m is the first argument position corresponding to these COMPONENTS. Then w' are the sorts of these arguments, so $w' = \langle s_m, \ldots, s_{m+|w'|-1}\rangle$.

$$\frac{s' \in S}{(S, TF, PF, P), f, ws, m \vdash s' \rhd \langle s'\rangle, (\emptyset, \emptyset)}$$

$$\frac{s' \in S \qquad (\langle s_1, \ldots, s_n \rangle, s) = ws \qquad x^i \neq x^j \text{ for all } 1 \leq i \neq j \leq n}{\begin{array}{l}(S, TF, PF, P), f, ws, m \vdash \texttt{total-select}\ f^1\ \ldots\ f^p\ s' \rhd \\ \quad \langle \underbrace{s', \ldots, s'}_{p \text{ times}} \rangle, ((\emptyset, \{(\langle s \rangle, s') \mapsto \{f^1, \ldots, f^p\}\}, \emptyset, \emptyset), \\ \qquad \{\forall \{x^1_{s_1}, \ldots, x^n_{s_n}\}.\ D(f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad f^1_{\langle s \rangle, s'}\langle f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle\rangle \stackrel{\mathrm{s}}{=} x^m_{s_m}, \\ \qquad \ldots, \\ \qquad \forall \{x^1_{s_1}, \ldots, x^n_{s_n}\}.\ D(f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad f^p_{\langle s \rangle, s'}\langle f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle\rangle \stackrel{\mathrm{s}}{=} x^{m+p-1}_{s_{m+p-1}}\})\end{array}}$$

Note that $f^1, \ldots, f^p$ will be in TF because of non-linear visibility.

$$\frac{s' \in S \qquad (\langle s_1, \ldots, s_n \rangle, s) = ws \qquad x^i \neq x^j \text{ for all } 1 \leq i \neq j \leq n}{\begin{array}{l}(S, TF, PF, P), f, ws, m \vdash \texttt{partial-select}\ f^1\ \ldots\ f^p\ s' \rhd \\ \quad \langle \underbrace{s', \ldots, s'}_{p \text{ times}} \rangle, ((\emptyset, \emptyset, \{(\langle s \rangle, s') \mapsto \{f^1, \ldots, f^p\}\}, \emptyset), \\ \qquad \{\forall \{x^1_{s_1}, \ldots, x^n_{s_n}\}.\ D(f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad f^1_{\langle s \rangle, s'}\langle f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle\rangle \stackrel{\mathrm{s}}{=} x^m_{s_m}, \\ \qquad \ldots, \\ \qquad \forall \{x^1_{s_1}, \ldots, x^n_{s_n}\}.\ D(f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad f^p_{\langle s \rangle, s'}\langle f_{ws}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle\rangle \stackrel{\mathrm{s}}{=} x^{m+p-1}_{s_{m+p-1}}\})\end{array}}$$

Note that $f^1, \ldots, f^p$ will be in PF because of non-linear visibility.

2.1.4.2 Free Datatype Declarations

A `FREE-DATATYPE` construct is only well-formed when its constructors are total. The same sorts, constructors, and selectors are declared as in ordinary datatype declarations. Apart from the sentences defining the values of selectors, additional sentences require the constructors to be injective, the ranges of constructors of the same sort to be disjoint, the declared sorts to be generated by the constructors, and that applying a selector to a constructor for which it has not been declared is undefined.

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

$$\boxed{\Sigma \vdash \texttt{FREE-DATATYPE} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W \qquad (S, TF, PF, P) = \Sigma \qquad (S', TF', \emptyset, P') = \Delta \qquad S'' = S \cup S'}{\begin{array}{l}\Sigma \vdash \texttt{free-datatype DATATYPE-ITEMS} \rhd \\ \quad (\Delta \cup \Delta', \Psi \cup \{\mathit{injective}(f_{w,s}) \mid w \in \mathit{FinSeq}(S''), s \in S'', f \in TF'_{w,s}\} \\ \qquad \cup \{\mathit{disjoint\text{-}ranges}(f_{w,s}, g_{w',s}) \\ \qquad\quad \mid w, w' \in \mathit{FinSeq}(S''), s \in S'', f \in TF'_{w,s}, g \in TF'_{w',s} \\ \qquad\quad \text{such that } w \neq w' \text{ or } f \neq g\} \\ \qquad \cup \{\mathit{undefined\text{-}selection}(f_{w,s}, g_{\langle s\rangle,s'}) \\ \qquad\quad \mid f_{w,s}, f'_{w',s} \in \mathit{Dom}(W), g_{\langle s\rangle,s'} \in W(f'_{w',s}) \setminus W(f_{w,s})\} \\ \qquad \cup \{(S', \mathit{complete}(TF', \mathit{FinSeq}(S'') \times S''))\}\ )\end{array}}$$

where:

- *injective*$(f_{w,s})$ is the following $(\Sigma \cup \Delta \cup \Delta')$-sentence which states that $f_{w,s}$ is injective:

$$\begin{array}{l}\forall\{x^1_{s_1}, \ldots, x^n_{s_n}, y^1_{s_1}, \ldots, y^n_{s_n}\}. \\ \qquad f_{w,s}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle \stackrel{s}{=} f_{w,s}\langle y^1_{s_1}, \ldots, y^n_{s_n}\rangle \Rightarrow \\ \qquad\qquad x^1_{s_1} \stackrel{s}{=} y^1_{s_1} \wedge \cdots \wedge x^n_{s_n} \stackrel{s}{=} y^n_{s_n}\end{array}$$

  where $\langle s_1, \ldots, s_n\rangle = w$ and $x^1, \ldots, x^n, y^1, \ldots, y^n$ are distinct variables.

- *disjoint-ranges*$(f_{w,s}, g_{w',s})$ is the following $(\Sigma \cup \Delta \cup \Delta')$-sentence which states that $f_{w,s}$ and $g_{w',s}$ have disjoint ranges:

$$\forall\{x^1_{s_1}, \ldots, x^m_{s_m}, y^1_{s'_1}, \ldots, y^n_{s'_n}\}.\neg(f_{w,s}\langle x^1_{s_1}, \ldots, x^m_{s_m}\rangle \stackrel{s}{=} g_{w',s}\langle y^1_{s'_1}, \ldots, y^n_{s'_n}\rangle)$$

  where $\langle s_1, \ldots, s_m\rangle = w$, $\langle s'_1, \ldots, s'_n\rangle = w'$ and $x^1, \ldots, x^m, y^1, \ldots, y^n$ are distinct variables.

- *undefined-selection*$(f_{w,s}, g_{\langle s\rangle,s'})$ is the following $(\Sigma \cup \Delta \cup \Delta')$-sentence which states that the value of applying the selector $g_{\langle s\rangle,s'}$ to values produced by the constructor $f_{w,s}$ (for which it has not been declared) is undefined:

$$\forall\{x^1_{s_1}, \ldots, x^n_{s_n}\}.\neg D(g_{\langle s\rangle,s'}\langle f_{w,s}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle\rangle)$$

  where $\langle s_1, \ldots, s_n\rangle = w$ and $x^1, \ldots, x^n$ are distinct variables.

See the beginning of Section 2.1.4 for an explanation of Δ, Δ', Ψ and W as produced by the judgement for `DATATYPE-ITEMS`. The third premise imposes the condition that all declared constructors are total. Note that $(S', \mathit{complete}(TF', \mathit{FinSeq}(S'') \times S''))$ in the last line of the rule is a sort generation constraint, and recall that this abbreviates $(S', \mathit{complete}(TF', \mathit{FinSeq}(S'') \times S''), id_{\Sigma\cup\Delta\cup\Delta'})$. This requires that all values of sorts declared by `DATATYPE-ITEMS` are generated by the declared constructors.

The following proposition states that the resulting model class is the same as for a free extension with the datatype declarations.

Proposition 19 *Consider a declaration* `free-datatype DATATYPE-ITEMS`*, a signature* Σ *and a model class* $\mathcal{M}$ *over* Σ*, and suppose*

$$\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W$$
$$\Sigma \vdash \texttt{free-datatype DATATYPE-ITEMS} \rhd (\Delta \cup \Delta', \Psi')$$

such that `DATATYPE-ITEMS` *fulfils the following conditions (all referring to fully qualified symbols):*

- *The sorts in* Δ *are not in the local environment* Σ*, and each* `DATATYPE-DECL` *declares a different sort.*
- *The constructors in* Δ *are distinct from each other.*
- *The selectors within each* `ALTERNATIVE` *are distinct.*
- *The constructors in* Δ *and selectors in* Δ' *are distinct from the symbols in the local environment* Σ*.*
- *Any selector in* Δ' *is total only when the same selector is present in all* `ALTERNATIVE`*s for that sort.*

Let $\mathcal{C}$ *be the full subcategory of* $\mathbf{Mod}(\Sigma \cup \Delta \cup \Delta')$ *containing those* $(\Sigma \cup \Delta \cup \Delta')$*-models* M'' *such that* $\forall \psi \in \Psi . M'' \models \psi$*, and let* $\mathcal{M}'$ *and* $\mathcal{M}''$ *be the* $(\Sigma \cup \Delta \cup \Delta')$*-model classes*

$$\begin{aligned}
\mathcal{M}' &= \{(\Sigma \cup \Delta \cup \Delta')\text{-model } M' \\
&\qquad \mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \in \mathcal{M} \text{ and } M' \text{ is free over } M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \\
&\qquad\quad \text{w.r.t. } .|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} : \mathcal{C} \to \mathbf{Mod}(\Sigma)\} \\
\mathcal{M}'' &= \{(\Sigma \cup \Delta \cup \Delta')\text{-model } M' \\
&\qquad \mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \in \mathcal{M} \text{ and } \forall \psi' \in \Psi' . M' \models \psi'\}
\end{aligned}$$

Then $\mathcal{M}' = \mathcal{M}''$*.*

PROOF: *See Theorem 25 in Section 4.1.2 for a more general result.* □

A few examples should help to clarify the above definitions. Since there is no overloading in these examples, ordinary function names are used instead of qualified function names to reduce clutter, and the usual syntax for variable typing is used.

Here is an example of a free datatype declaration where all alternatives are constants, which corresponds to an unordered enumeration type:

free type *Colour* ::= *red* | *blue*

The result is the following enrichment (relative to the empty signature):

$$\begin{array}{l}(\Sigma, \{\mathit{red}\langle\rangle \stackrel{s}{=} \mathit{red}\langle\rangle \Rightarrow \mathit{true},\\ \quad \mathit{blue}\langle\rangle \stackrel{s}{=} \mathit{blue}\langle\rangle \Rightarrow \mathit{true},\\ \quad \neg(\mathit{red}\langle\rangle \stackrel{s}{=} \mathit{blue}\langle\rangle),\\ \quad \neg(\mathit{blue}\langle\rangle \stackrel{s}{=} \mathit{red}\langle\rangle),\\ \quad (\{Colour\}, TF, id_\Sigma)\}\end{array}$$

where $\Sigma = (S, TF, PF, P)$ is the signature containing the sort *Colour*, the total function symbols *red* and *blue*, and no partial function symbols or predicate symbols. The first two sentences are from the *injective* condition and are tautologous, as always for nullary constructors. The next two sentences are from the *disjoint-ranges* condition and are equivalent (such duplication will always be present but it does no harm). The final sentence is a sort generation constraint which requires every value of sort *Colour* to be produced by either $\mathit{red}\langle\rangle$ or $\mathit{blue}\langle\rangle$. Each model has a carrier of sort *Colour* containing exactly two values.

Here is the standard example of lists, with selectors:

free type $\mathit{List} ::= \mathit{nil} \mid \mathit{cons}(\mathit{first} :?\mathit{Elem};\ \mathit{rest} :?\mathit{List})$

The result is the following enrichment (relative to a signature Σ containing just the sort *Elem*):

$$\begin{array}{l}(\Delta, \{\forall x{:}\mathit{Elem}, x'{:}\mathit{List}.D(\mathit{cons}\langle x, x'\rangle) \Rightarrow \mathit{first}\langle\mathit{cons}\langle x, x'\rangle\rangle \stackrel{s}{=} x,\\ \quad \forall x{:}\mathit{Elem}, x'{:}\mathit{List}.D(\mathit{cons}\langle x, x'\rangle) \Rightarrow \mathit{rest}\langle\mathit{cons}\langle x, x'\rangle\rangle \stackrel{s}{=} x',\\ \quad \mathit{nil}\langle\rangle \stackrel{s}{=} \mathit{nil}\langle\rangle \Rightarrow \mathit{true},\\ \quad \forall x{:}\mathit{Elem}, x'{:}\mathit{List}, y{:}\mathit{Elem}, y'{:}\mathit{List}.\\ \qquad\qquad \mathit{cons}\langle x, x'\rangle \stackrel{s}{=} \mathit{cons}\langle y, y'\rangle \Rightarrow x \stackrel{s}{=} y \wedge x' \stackrel{s}{=} y',\\ \quad \forall x{:}\mathit{Elem}, x'{:}\mathit{List}.\neg(\mathit{nil}\langle\rangle \stackrel{s}{=} \mathit{cons}\langle x, x'\rangle),\\ \quad \forall x{:}\mathit{Elem}, x'{:}\mathit{List}.\neg(\mathit{cons}\langle x, x'\rangle \stackrel{s}{=} \mathit{nil}\langle\rangle),\\ \quad \neg D(\mathit{first}\langle\mathit{nil}\langle\rangle\rangle),\\ \quad \neg D(\mathit{rest}\langle\mathit{nil}\langle\rangle\rangle),\\ \quad (\{\mathit{List}\}, TF, id_{\Sigma\cup\Delta})\}\end{array}$$

where $\Delta = (S, TF, PF, P)$ is the signature extension (relative to Σ) containing the sort *List*, the total function symbols *nil* and *cons*, the partial function symbols *first* and *rest*, and no predicate symbols. The first two sentences are generated by the rules for `COMPONENTS` and specify the relationship between the constructor *cons* and the selectors *first* and *rest*. The next two sentences are from the *injective* condition. The next two sentences are from the *disjoint-ranges* condition; again, they are equivalent. The next two sentences are from the *undefined-selection* condition. The final sentence is a sort generation constraint which requires each value of sort *List* to be

produced by a term of the form

$$cons\langle x_1, \ldots, cons\langle x_n, nil\rangle \ldots\rangle.$$

for some assignment of values of sort *Elem* to the variables $x_1, \ldots, x_n$. Models are as one would expect from this specification, with "no junk" and "no confusion", and the selectors defined only for values produced by *cons*.

Here is a type containing two copies of the natural numbers, with the same selector for both:

free type $Twonats ::= left(get : Nat) \mid right(get : Nat)$

The result is the following enrichment (relative to a signature Σ containing just the sort *Nat*):

$$\begin{array}{l}(\Delta, \{\forall x{:}Nat.D(left\langle x\rangle) \Rightarrow get\langle left\langle x\rangle\rangle \stackrel{s}{=} x,\\ \quad \forall x{:}Nat.D(right\langle x\rangle) \Rightarrow get\langle right\langle x\rangle\rangle \stackrel{s}{=} x,\\ \quad \forall x{:}Nat, x'{:}Nat.left\langle x\rangle \stackrel{s}{=} left\langle x'\rangle \Rightarrow x \stackrel{s}{=} x',\\ \quad \forall x{:}Nat, x'{:}Nat.right\langle x\rangle \stackrel{s}{=} right\langle x'\rangle \Rightarrow x \stackrel{s}{=} x',\\ \quad \forall x{:}Nat, x'{:}Nat.\neg(left\langle x\rangle \stackrel{s}{=} right\langle x'\rangle),\\ \quad \forall x{:}Nat, x'{:}Nat.\neg(right\langle x\rangle \stackrel{s}{=} left\langle x'\rangle),\\ \quad (\{Twonats\}, TF', id_{\Sigma\cup\Delta})\}\end{array}$$

where $\Delta = (S, TF, PF, P)$ is the signature extension (relative to Σ) containing the sort *Twonats*, the total function symbols *left*, *right* and *get*, and no partial function symbols or predicate symbols, and TF' contains the total function symbols *left* and *right*. The first two sentences are generated by the rules for `COMPONENTS` and specify the relationship between the constructors *left* and *right* and the selector *get*. The next two sentences are from the *injective* condition. The next two sentences are from the *disjoint-ranges* condition; once more, they are equivalent. The final sentence is a sort generation constraint which requires each value of sort *Twonats* to be produced by either $left\langle x\rangle$ or $right\langle x\rangle$ for some assignment of a value of sort *Nat* to x. Models are as one would expect. Note that the total selector $get : Twonats \to Nat$ which is present in both `ALTERNATIVE`s becomes a single selector in the rule for `DATATYPE-DECL`: we have

$\Sigma, Twonats \vdash$ `total-construct` *left* (`total-select` *get Nat*)$\triangleright(\Delta_1, \Delta_1', \Psi_1)$

$\Sigma, Twonats \vdash$ `total-construct` *right* (`total-select` *get Nat*)$\triangleright(\Delta_2, \Delta_2', \Psi_2)$

where Δ_1 contains *left*, Δ_2 contains *right*, $\Delta_1' = \Delta_2'$ contains *get*, Ψ_1 contains $\forall x{:}Nat.D(left\langle x\rangle) \Rightarrow get\langle left\langle x\rangle\rangle \stackrel{s}{=} x$ and Ψ_2 contains $\forall x{:}Nat.D(right\langle x\rangle) \Rightarrow get\langle right\langle x\rangle\rangle \stackrel{s}{=} x$.

Changing the declaration to

free type $Twonats ::= left(get : Nat) \mid right(Nat)$

would cause the sentence $\forall x{:}Nat.D(right\langle x\rangle) \Rightarrow get\langle right\langle x\rangle\rangle \stackrel{s}{=} x$ to be omitted from the result, but otherwise there would be no difference. Note that the term $get\langle right\langle x\rangle\rangle$ is still required to have some defined value for every x, since *get* and *right* are total function symbols, but that value is unconstrained.

Finally, here is what happens when an attempt is made to define an empty type as a free datatype:

free type $Empty ::= f(Empty)$

The result is the following enrichment (relative to the empty signature):

$$(\Sigma, \{\forall x{:}Empty, y{:}Empty.f\langle x\rangle \stackrel{s}{=} f\langle y\rangle \Rightarrow x \stackrel{s}{=} y, \\ (\{Empty\}, TF, id_\Sigma)\}$$

where $\Sigma = (S, TF, PF, P)$ is the signature containing the sort *Empty*, the total function symbol f, and no partial function symbols or predicate symbols. The first sentence is from the *injective* condition. The second sentence is a sort generation constraint which requires every value of sort *Empty* to be produced by a Σ-term containing no variables. There are no such terms since there are no constants of sort *Empty*; hence this requires the carrier of sort *Empty* to be empty. But models are required to have non-empty carriers, and therefore there are no models.

2.1.5 Sort Generation

A sort generation `SORT-GEN` determines the same signature elements and sentences as its list of `SIG-ITEMS`s, together with a sort generation constraint requiring the declared sorts to be generated by the declared operations, but excluding operations declared as selectors.

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

$$\Sigma \vdash \texttt{SORT-GEN} \rhd (\Delta, \Psi)$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{SIG-ITEMS}_1 \rhd (\Delta_1, \Delta_1', \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \Delta_1' \cup \cdots \cup \Delta_{n-1} \cup \Delta_{n-1}' \vdash \texttt{SIG-ITEMS}_n \rhd (\Delta_n, \Delta_n', \Psi_n) \\ (S, TF, PF, P) = \Delta = \Delta_1 \cup \cdots \cup \Delta_n \qquad \Delta' = \Delta_1' \cup \cdots \cup \Delta_n' \\ (S', TF', PF', P') = \Sigma \cup \Delta \cup \Delta' \qquad S \neq \emptyset\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{sort-gen SIG-ITEMS}_1 \ \ldots \ \texttt{SIG-ITEMS}_n \rhd \\ \quad (\Delta \cup \Delta', \Psi_1 \cup \cdots \cup \Psi_n \cup \{(S, \mathit{complete}(TF \cup PF, \mathit{FinSeq}(S') \times S'))\}\,)\end{array}}$$

In this rule, Δ represents the signature extension declared by $\texttt{SIG-ITEMS}_1 \ldots \texttt{SIG-ITEMS}_n$, excluding the operations declared as selectors since these do not contribute to the resulting sort generation constraint. The predicate symbols in Δ also make no contribution.

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.

2.2.1 Global Variable Declarations

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

$$\boxed{S \vdash \texttt{VAR-ITEMS} \rhd X}$$

X is a valid set of variables over S.

$$\frac{S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n}{S \vdash \texttt{var-items VAR-DECL}_1 \ \ldots \ \texttt{VAR-DECL}_n \rhd X_1 + \cdots + X_n}$$

A variable declaration VAR-DECL declares the given variables to be of the given sort for use in subsequent axioms. This adds a universal quantification on those variables to the subsequent axioms of the enclosing basic specification.

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

$$\boxed{S \vdash \texttt{VAR-DECL} \rhd X}$$

X is a valid set of variables over S.

$$\frac{s \in S}{S \vdash \texttt{var-decl}\ x_1\ \ldots\ x_n\ \ s \rhd \mathit{complete}(\{s \mapsto \{x_1, \ldots, x_n\}\}, S)}$$

A later declaration for a variable overrides an earlier declaration for the same identifier because of the use of + to combine variable sets in the rules for BASIC-SPEC and VAR-ITEMS. Universal quantification over all declared variables, both global and local, is added in the rule for AXIOM.

$$\mathit{Var} = \texttt{SIMPLE-ID}$$

2.2.2 Local Variable Declarations

A LOCAL-VAR-AXIOMS construct declares variables for local use in the given axioms, and adds a universal quantification on those variables to all those axioms.

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

$$\boxed{\Sigma, X \vdash \texttt{LOCAL-VAR-AXIOMS} \rhd \Psi}$$

X is required to be a valid set of variables over the sorts of Σ. Ψ is a set of Σ-sentences.

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \quad \cdots \quad S \vdash \texttt{VAR-DECL}_m \rhd X_m \\ \Sigma, X + X_1 + \cdots + X_m \vdash \texttt{AXIOM}_1 \rhd \psi_1 \quad \cdots \quad \Sigma, X + X_1 + \cdots + X_m \vdash \texttt{AXIOM}_n \rhd \psi_n\end{array}}{\Sigma, X \vdash \texttt{local-var-axioms VAR-DECL}_1\ \ldots\ \texttt{VAR-DECL}_m\ \texttt{AXIOM}_1\ \ldots\ \texttt{AXIOM}_n \rhd \{\psi_1, \ldots, \psi_n\}}$$

2.3 Axioms

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

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

$$\boxed{\Sigma, X \vdash \texttt{AXIOM-ITEMS} \rhd \Psi}$$

X is required to be a valid set of variables over the sorts of Σ. Ψ is a set of Σ-sentences.

$$\frac{\Sigma, X \vdash \texttt{AXIOM}_1 \rhd \psi_1 \qquad \cdots \qquad \Sigma, X \vdash \texttt{AXIOM}_n \rhd \psi_n}{\Sigma, X \vdash \texttt{axiom-items AXIOM}_1 \ \ldots \ \texttt{AXIOM}_n \rhd \{\psi_1, \ldots, \psi_n\}}$$

$$\boxed{\Sigma, X \vdash \texttt{AXIOM} \rhd \psi}$$

X is required to be a valid set of variables over the sorts of Σ. ψ is a Σ-sentence.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi}{\Sigma, X \vdash \texttt{FORMULA} \text{ qua } \texttt{AXIOM} \rhd \forall X.\varphi}$$

All declared variables are universally quantified. Quantification over variables that do not occur free in the axiom has no effect since carriers are assumed to be non-empty.

A formula is constructed from atomic formulae using quantification and the usual logical connectives.

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

$$\boxed{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

Rules elided, except for the one for ATOM qua FORMULA which is near the beginning of Section 2.3.3 below to keep it together with subordinate rules for atomic formulae.

2.3.1 Quantifications

Universal, existential and unique existential quantification are as usual. 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.

```
QUANTIFICATION ::= quantification QUANTIFIER VAR-DECL+ FORMULA
```

```
QUANTIFIER     ::= universal | existential | unique-existential
```

$$\boxed{\Sigma, X \vdash \texttt{QUANTIFICATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n \\ \Sigma, X + X_1 + \cdots + X_n \vdash \texttt{FORMULA} \rhd \varphi\end{array}}{\begin{array}{r}\Sigma, X \vdash \texttt{quantification universal VAR-DECL}_1 \ldots \texttt{VAR-DECL}_n \texttt{ FORMULA} \rhd \\ \forall X_1 + \cdots + X_n . \varphi\end{array}}$$

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n \\ \Sigma, X + X_1 + \cdots + X_n \vdash \texttt{FORMULA} \rhd \varphi\end{array}}{\begin{array}{r}\Sigma, X \vdash \texttt{quantification existential VAR-DECL}_1 \ldots \texttt{VAR-DECL}_n \texttt{ FORMULA} \rhd \\ \exists X_1 + \cdots + X_n . \varphi\end{array}}$$

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n \\ \Sigma, X + X_1 + \cdots + X_n \vdash \texttt{FORMULA} \rhd \varphi\end{array}}{\begin{array}{r}\Sigma, X \vdash \texttt{quantification unique-existential VAR-DECL}_1 \ldots \texttt{VAR-DECL}_n \texttt{ FORMULA} \rhd \\ \exists! X_1 + \cdots + X_n . \varphi\end{array}}$$

2.3.2 Logical Connectives

The logical connectives are as usual, except that conjunction and disjunction apply to lists of two or more formulae.

2.3.2.1 Conjunction

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

$$\boxed{\Sigma, X \vdash \texttt{CONJUNCTION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA}_1 \rhd \varphi_1 \qquad \Sigma, X \vdash \texttt{FORMULA}_2 \rhd \varphi_2 \qquad \cdots \qquad \Sigma, X \vdash \texttt{FORMULA}_n \rhd \varphi_n}{\Sigma, X \vdash \texttt{conjunction FORMULA}_1 \texttt{ FORMULA}_2 \ldots \texttt{FORMULA}_n \rhd (\cdots(\varphi_1 \wedge \varphi_2) \wedge \cdots) \wedge \varphi_n}$$

2.3.2.2 Disjunction

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

$$\boxed{\Sigma, X \vdash \texttt{DISJUNCTION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA}_1 \rhd \varphi_1 \qquad \Sigma, X \vdash \texttt{FORMULA}_2 \rhd \varphi_2 \qquad \cdots \qquad \Sigma, X \vdash \texttt{FORMULA}_n \rhd \varphi_n}{\Sigma, X \vdash \texttt{disjunction FORMULA}_1\,\texttt{FORMULA}_2\,\ldots\,\texttt{FORMULA}_n \rhd (\cdots(\varphi_1 \vee \varphi_2) \vee \cdots) \vee \varphi_n}$$

2.3.2.3 Implication

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

$$\boxed{\Sigma, X \vdash \texttt{IMPLICATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi \qquad \Sigma, X \vdash \texttt{FORMULA}' \rhd \varphi'}{\Sigma, X \vdash \texttt{implication FORMULA FORMULA}' \rhd \varphi \Rightarrow \varphi'}$$

2.3.2.4 Equivalence

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

$$\boxed{\Sigma, X \vdash \texttt{EQUIVALENCE} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi \qquad \Sigma, X \vdash \texttt{FORMULA}' \rhd \varphi'}{\Sigma, X \vdash \texttt{equivalence FORMULA FORMULA}' \rhd \varphi \Leftrightarrow \varphi'}$$

2.3.2.5 Negation

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

$$\boxed{\Sigma, X \vdash \texttt{NEGATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi}{\Sigma, X \vdash \texttt{negation FORMULA} \rhd \neg\varphi}$$

2.3.3 Atomic Formulae

An atomic formula is well-formed 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 captured by the rules below.

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

(The following rule really belongs just before Section 2.3.1 above. It is here in order to keep it together with the subordinate rules for atomic formulae, because of the complications introduced by the "unique expansion" requirement.)

$$\frac{\begin{array}{c}\exists!\varphi \text{ such that } \Sigma, X \vdash \texttt{ATOM} \rhd \varphi \\ \Sigma, X \vdash \texttt{ATOM} \rhd \varphi\end{array}}{\Sigma, X \vdash \texttt{ATOM qua FORMULA} \rhd \varphi}$$

The first premise of this rule imposes the requirement that ATOM expands to a unique (fully-qualified) atomic formula. In this premise, the static semantics of ATOM occurs in a negative position (introduced by $\exists!$). This is potentially problematic, especially since there is a circularity: the judgement $\Sigma, X \vdash \texttt{ATOM} \rhd \varphi$ depends on the judgement $\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi'$ if ATOM contains a conditional term. But since FORMULA will then be strictly contained within ATOM, there is no problem: we can (implicitly) impose a stratification on the judgements for FORMULA and ATOM where the semantics of larger formulae/atoms is based on the (fixed) semantics of strictly smaller formulae/atoms.

$$\boxed{\Sigma, X \vdash \texttt{ATOM} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

Rules elided, except for the following one:

$$\frac{\vdash \texttt{TRUTH} \rhd \varphi}{\Sigma, X \vdash \texttt{TRUTH qua ATOM} \rhd \varphi}$$

2.3.3.1 Truth

The atomic formulae for truth and falsity are always well-sorted, and expand to primitive sentences.

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

$$\boxed{\vdash \texttt{TRUTH} \rhd \varphi}$$

φ is a Σ-formula over X for any Σ and X.

$$\frac{}{\vdash \texttt{true-atom} \rhd \mathit{true}}$$

$$\frac{}{\vdash \texttt{false-atom} \rhd \mathit{false}}$$

2.3.3.2 Predicate Application

The application of a predicate symbol is well-sorted when there is a declaration of the 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.

```
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*
```

$$\boxed{\Sigma, X \vdash \texttt{PREDICATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{PRED-SYMB} \rhd p, \langle s_1, \ldots, s_n \rangle \\ \Sigma, X \vdash \texttt{TERMS} \rhd \langle t_1, \ldots, t_n \rangle \\ sort(t_1) = s_1 \quad \cdots \quad sort(t_n) = s_n\end{array}}{\Sigma, X \vdash \texttt{predication PRED-SYMB TERMS} \rhd p_{\langle s_1,\ldots,s_n \rangle}\langle t_1, \ldots, t_n \rangle}$$

$$\boxed{\Sigma \vdash \texttt{PRED-SYMB} \rhd p, w}$$

p is a predicate symbol in Σ with profile w.

$$\frac{\{s_1, \ldots, s_n\} \subseteq S \qquad p \in P_{\langle s_1,\ldots,s_n \rangle}}{(S, TF, PF, P) \vdash p \rhd p, \langle s_1, \ldots, s_n \rangle}$$

$$\frac{S \vdash \texttt{PRED-TYPE} \rhd w \qquad p \in P_w}{(S, TF, PF, P) \vdash \texttt{qual-pred-name } p \texttt{ PRED-TYPE} \rhd p, w}$$

$$\boxed{\Sigma, X \vdash \texttt{TERMS} \rhd \langle t_1, \ldots, t_n \rangle}$$

X is required to be a valid set of variables over the sorts of Σ. $t_1, \ldots, t_n$ are fully-qualified Σ-terms over X.

$$\frac{\Sigma, X \vdash \texttt{TERM}_1 \rhd t_1 \quad \cdots \quad \Sigma, X \vdash \texttt{TERM}_n \rhd t_n}{\Sigma, X \vdash \texttt{terms TERM}_1 \ \ldots \ \texttt{TERM}_n \rhd \langle t_1, \ldots, t_n \rangle}$$

2.3.3.3 Definedness

A definedness formula 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.

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

$$\boxed{\Sigma, X \vdash \texttt{DEFINEDNESS} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t}{\Sigma, X \vdash \texttt{definedness TERM} \rhd D(t)}$$

2.3.3.4 Equations

An equation is well-sorted if both terms are well-sorted for some sort. It then expands to the corresponding equation on the fully-qualified expansions of the terms for that sort.

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

$$\boxed{\Sigma, X \vdash \texttt{EXISTL-EQUATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad \Sigma, X \vdash \texttt{TERM}' \rhd t' \qquad sort(t) = sort(t')}{\Sigma, X \vdash \texttt{existl-equation TERM TERM}' \rhd t \stackrel{\text{e}}{=} t'}$$

$$\boxed{\Sigma, X \vdash \texttt{STRONG-EQUATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad \Sigma, X \vdash \texttt{TERM}' \rhd t' \qquad sort(t) = sort(t')}{\Sigma, X \vdash \texttt{strong-equation TERM TERM}' \rhd t \stackrel{\text{s}}{=} t'}$$

2.3.4 Terms

A term is constructed from 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.

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

$$\boxed{\Sigma, X \vdash \texttt{TERM} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

Rules elided, except for the two rules in the next subsection which are for the case SIMPLE-ID.

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.

$$\frac{s \in S \qquad x \in X_s}{(S, TF, PF, P), X \vdash x \rhd x_s}$$

$$\frac{s \in S \qquad f \in TF_{\langle\rangle,s} \cup PF_{\langle\rangle,s}}{(S, TF, PF, P), X \vdash f \rhd f_{\langle\rangle,s}\langle\rangle}$$

2.3.4.2 Qualified Variables

A qualified variable is well-sorted for the given sort.

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

$$\boxed{\Sigma, X \vdash \texttt{QUAL-VAR} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{s \in S \qquad x \in X_s}{(S, TF, PF, P), X \vdash \texttt{qual-var}\ x\ s \rhd x_s}$$

2.3.4.3 Operation Application

An application is well-sorted for some sort s when there is a declaration of the operation name such that all the argument terms are well-sorted for the respective argument sorts, and the result sort is s. It then expands to an application of the qualified operation name to the fully-qualified expansions of the argument terms for those sorts.

```
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*
```

$$\boxed{\Sigma, X \vdash \texttt{APPLICATION} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{OP-SYMB} \rhd f, (\langle s_1, \ldots, s_n\rangle, s) \\ \Sigma, X \vdash \texttt{TERMS} \rhd \langle t_1, \ldots, t_n\rangle \\ sort(t_1) = s_1 \quad \cdots \quad sort(t_n) = s_n\end{array}}{\Sigma, X \vdash \texttt{application OP-SYMB TERMS} \rhd f_{\langle s_1,\ldots,s_n\rangle,s}\langle t_1, \ldots, t_n\rangle}$$

$$\boxed{\Sigma \vdash \texttt{OP-SYMB} \rhd f, ws}$$

f is a function symbol in Σ with profile ws.

$$\frac{\{s_1, \ldots, s_n, s\} \subseteq S \qquad f \in TF_{\langle s_1,\ldots,s_n\rangle,s} \cup PF_{\langle s_1,\ldots,s_n\rangle,s}}{(S, TF, PF, P) \vdash f \rhd f, (\langle s_1, \ldots, s_n\rangle, s)}$$

$$\frac{\{s_1, \ldots, s_n, s\} \subseteq S \qquad f \in TF_{\langle s_1,\ldots,s_n\rangle,s}}{(S, TF, PF, P) \vdash \texttt{qual-op-name}\ f\ (\texttt{total-op-type}\ (\texttt{sort-list}\ s_1\ \ldots\ s_n)\ s) \rhd f, (\langle s_1, \ldots, s_n\rangle, s)}$$

$$\frac{\{s_1, \ldots, s_n, s\} \subseteq S \qquad f \in PF_{\langle s_1,\ldots,s_n\rangle,s}}{(S, TF, PF, P) \vdash \texttt{qual-op-name}\ f\ (\texttt{partial-op-type}\ (\texttt{sort-list}\ s_1\ \ldots\ s_n)\ s) \rhd f, (\langle s_1, \ldots, s_n\rangle, s)}$$

$$\boxed{\Sigma, X \vdash \texttt{TERMS} \rhd \langle t_1, \ldots, t_n\rangle}$$

X is required to be a valid set of variables over the sorts of Σ. $t_1, \ldots, t_n$ are fully-qualified Σ-terms over X.

$$\frac{\Sigma, X \vdash \texttt{TERM}_1 \rhd t_1 \quad \cdots \quad \Sigma, X \vdash \texttt{TERM}_n \rhd t_n}{\Sigma, X \vdash \texttt{terms}\ \ \texttt{TERM}_1\ \ldots\ \texttt{TERM}_n \rhd \langle t_1, \ldots, t_n\rangle}$$

2.3.4.4 Sorted Terms

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

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

$$\boxed{\Sigma, X \vdash \texttt{SORTED-TERM} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad sort(t) = s}{\Sigma, X \vdash \texttt{sorted-term TERM } s \rhd t}$$

2.3.4.5 Conditional Terms

A conditional term is well-sorted for some sort when both given terms are well-sorted for that sort and the given formula is well-formed. It then expands to a fully-qualified term built from that formula and the fully-qualified expansions of the given terms for that sort.

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

$$\boxed{\Sigma, X \vdash \texttt{CONDITIONAL} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad \Sigma, X \vdash \texttt{FORMULA} \rhd \varphi \qquad \Sigma, X \vdash \texttt{TERM}' \rhd t' \qquad sort(t) = sort(t')}{\Sigma, X \vdash \texttt{conditional TERM FORMULA TERM}' \rhd \varphi \rightarrow t \mid t'}$$

Conditional terms are interpreted as fully-qualified terms, as explained in Section 1.3, rather than being handled by transformation of the enclosing atomic formula as is suggested in the Language Summary; such a transformation would be difficult to define using this style of semantics.

2.4 Identifiers

The internal structure of identifiers ID is insignificant in the context of basic specifications. (ID is extended with compound identifiers, whose structure is significant, in connection with generic specifications in Section 6.5.)

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

Chapter 3

Subsorting Concepts

Chapter 4

Subsorting Constructs

Part II

Structured Specifications

Chapter 5

Structuring Concepts

Chapter 6

Structuring Constructs

Part III

Architectural Specifications

Chapter 7

Architectural Concepts

Chapter 8

Architectural Constructs

Part IV

Specification Libraries

Chapter 9

Library Concepts

Chapter 10

Library Constructs

Bibliography

[CGRW95] Ingo Claßen, Martin Große-Rhode, and Uwe Wolter. Categorical concepts for parameterized partial specification. *Mathematical Structures in Computer Science*, 5:153–188, 1995.

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

[CoF01] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [CoF], March 2001.

[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.

[GM82] Joseph A. Goguen and José Meseguer. Completeness of many-sorted equational logic. *ACM SIGPLAN Notices*, 17(1):9–17, 1982.

[HS73] Horst Herrlich and George Strecker. *Category Theory*. Allyn and Bacon, 1973.

[Kah88] Gilles Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. North Holland, 1988.

[Mos] Till Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*. To appear.

[Mos98a] Till Mossakowski. Cocompleteness of the CASL signature category. Note S-7, in [CoF], February 1998.

[Mos98b] Till Mossakowski. Colimits of order-sorted specifications. In *Recent Trends in Algebraic Development Techniques, Proc. 12th International Workshop, WADT '97, Tarquinia, 1997, Selected*

Papers, volume 1376 of *LNCS*, pages 316–332. Springer-Verlag, 1998.

[Mos00] Till Mossakowski. Specification in an arbitrary institution with symbols. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*, pages 252–270. Springer-Verlag, 2000.

[Pie91] Benjamin Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[Prz88] Teodor Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[vG96] Rob van Glabbeek. The meaning of negative premises in transition system specifications II. In F. Meyer auf der Heide and B. Monien, editors, *Proc. 23rd Intl. Colloq. on Automata, Languages and Programming, ICALP'96, Paderborn*, volume 1099 of *Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 1996.

[Wag99] Eric Wagner. On the category of signatures. Presentation at WADT'99, Bonas, 1999.

Appendices

Appendix A

Abstract Syntax

The entire abstract syntax as in Appendix A of the CASL Language Summary [CoF01] is included here for convenience.

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 notation `COMPONENT*` indicates repetition of `COMPONENT` any number of times; `COMPONENT+` indicates repetition at least once.

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
```