HasCasl
Integrated functional specification and programming
Summary

CoFI Tentative Document: HasCasl Summary
Version: 0.3 23 March 2004

Lutz Schröder Till Mossakowski Christian Maeder
E-mail address for comments: {lschrode,till,maeder}@informatik.uni-bremen.de

CoFI: The Common Framework Initiative
http://www.brics.dk/Projects/CoFI

Abstract

The development of programs in modern functional languages such as Haskell calls for a wide-spectrum specification formalism that supports the type system of such languages, in particular higher order types, type constructors, and polymorphism, and that contains a functional language as an executable subset in order to facilitate rapid prototyping. We lay out the design of HasCasl, a higher order extension of Casl that is geared towards precisely this purpose. Its semantics is tuned to allow program development by specification refinement, while at the same time staying close to the set-theoretic semantics of first order Casl. The number of primitive concepts in the logic has been kept as small as possible; advanced concepts, in particular general recursion, can be formulated within the language itself. This document provides a detailed definition of the HasCasl syntax and an informal description of the semantics, building on the existing Casl Summary [CoF].

Contents

About this document

This document gives a detailed summary of the syntax and intended semantics of HasCasl. It is intended for readers already familiar with Casl. In particular, since HasCasl reuses the institution-independent mechanisms of Casl for structured and architectural specifications and libraries, these concepts will not be described here; for a summary of these language features, see [CoF]. Like [CoF], this document provides little or nothing in the way of discussion or motivation of design decisions; for such matters, see in particular [SM02].

In principle, HasCasl extends full Casl, i.e. all Casl features are also contained in HasCasl, with their meaning being preserved except in the case of sort generation constraints, and the same syntax can be used to invoke them. This does not, however, necessarily mean that all Casl specifications are parsable in HasCasl, since due to the additional language features of HasCasl, in particular the implicit application operator customary in higher order languages, the parsing of mixfix terms may require more brackets in HasCasl than in Casl.

Structure

The document consists of only one part, still called Part I in correspondence to the numbering of the parts of [CoF], dealing with *basic specifications*. Chapters 1 and 2 describe the basic logic of HasCasl including subtyping; since even in the most basic version, total function types are subtypes of partial function types, a separate treatment of subtyping as in [CoF] is not really feasible for HasCasl. Chapters 3 and 4 introduce the so-called internal logic, and Chapters 5 and 6 deal with general recursive functions, i.e. with functional programming, thus reflecting the bootstrap design of HasCasl. For each step, the first of the two chapters summarizes the main *semantic concepts*, and the second presents the (concrete and abstract) syntax of the associated HasCasl *language constructs* and indicates their intended semantics.

Like [CoF], this document provides appendices containing the abstract syntax (Appendix A) and the concrete syntax (Appendix C) of basic HasCasl specifications.

Future extensions of HasCasl will also cover a definite description operator, as well as special syntax for the specification of monadic programs (using the so-called do-notation and a monad independent modal logic, see [SM03, SM]). Another important feature currently missing in HasCasl is existential types. Support for concise specification of datatypes like infinite lazy lists would also be desirable; to this end, HasCasl could possibly import constructs from CoCasl [MSRR].

Part I

Basic Specifications

Chapter 1

Basic Concepts

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

A *basic specification framework* may be characterized by:

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

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

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

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

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

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

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

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

The rest of this chapter introduces the HasCasl logic in three steps: firstly, the partial λ-calculus is recalled, secondly, product types are added, and finally, polymorphism, type constructors, type constructor classes, subtypes and subkinds are defined on top of this.

The subsequent chapter considers many-sorted basic specifications of the HasCasl specification framework, and indicates the underlying signatures, models, and sentences. 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 The partial λ-calculus

The natural generalization of the simply typed λ-calculus to the setting of partial functions is the partial λ-calculus as introduced in [Mog86, Mog88, Ros86]. The basic idea is that function types are replaced by partial function types, and λ-abstractions denote partial functions instead of total ones.

A ***simple signature*** consists of a set of ***sorts*** and a set of partial ***operators*** with given ***profiles*** (or arities) written $f : \bar{s} \rightharpoonup t$, where t is a ***type*** and $\bar{s}$ is a ***multi-type***, i.e. a (possibly empty) list of types. A type is either a sort or a ***partial function type*** (product types will be introduced in Section 1.1.1)

$$\bar{s} \to ?t,$$

with $\bar{s}$ and t as above (one cannot resort to currying for multi-argument partial functions; e.g., $s \to ?(t \to ?u)$ is not isomorphic to $st \to ?u$ [Mog86]). Following [Mog86], we assume for each multi-type $\bar{s}$ and each type t an application operators with profile $(\bar{s} \to ?t)\bar{s} \rightharpoonup t$ in the signature, so that

$$\frac{x : s \text{ in } \Gamma}{\Gamma \rhd x : s} \qquad \frac{\Gamma \rhd \bar{\alpha} : \bar{t} \quad f : \bar{t} \rightharpoonup u}{\Gamma \rhd f(\bar{\alpha}) : u} \qquad \frac{\Gamma, \bar{y} : \bar{t} \rhd \alpha : u}{\Gamma \rhd \lambda\, \bar{y} : \bar{t} \bullet \alpha : \bar{t} \to ?u}$$

Figure 1.1: Typing rules for the partial λ-calculus

application does not require extra typing or deduction rules. This operator is denoted as usual by juxtaposition, while application of other operators in the signature is written with brackets. For $\bar{t} = (t_1, \dots, t_m)$, $\bar{s} \to ?\bar{t}$ denotes the multi-type $(\bar{s} \to ?t_1, \dots, \bar{s} \to ?t_m)$, not to be confused with the (non-existent) 'type' $\bar{s} \to ?t_1 \times \dots \times t_m$. A ***morphism*** between two simple signatures is a pair of maps between the corresponding sets of sorts and operators, respectively, that is compatible with operator profiles.

A signature gives rise to a notion of typed terms in context according to the typing rules given in Figure 1.1, where a context Γ is a list $(x_1 : s_1, \dots, x_n : s_n)$, shortly $(\bar{x} : \bar{s})$, of type assignments for distinct variables. More precisely, we speak simultaneously about terms and ***multi-terms***, i.e. lists of terms also denoted shortly in the form $\bar{\alpha}$ instead of $(\alpha_i, \dots, \alpha_n)$. The judgement $\Gamma \rhd \alpha : t$ reads '(multi-)term α has (multi-)type t in context Γ'. The empty multi-term () doubles as a term of 'type' (), where the latter is also denoted as *Unit*.

A ***partial λ-theory*** $\mathcal{T}$ is a signature Σ together with a set $\mathcal{A}$ of ***axioms*** that take the form of existentially conditioned equations: an (existential) ***equation*** $\bar{\alpha}_1 \overset{e}{=} \bar{\alpha}_2$ is read '$\bar{\alpha}_1$ and α_2 are defined and equal'. Equations $\bar{\alpha} \overset{e}{=} \bar{\alpha}$ are abbreviated as $\operatorname{def} \bar{\alpha}$ and called ***definedness judgements***. An ***existentially conditioned equation (ECE)*** is a sentence of the form $\Gamma \rhd \operatorname{def} \bar{\alpha} \Rightarrow \phi$, where $\bar{\alpha}$ is a multi-term and ϕ is an equation in context Γ, to be read 'ϕ holds on the domain of $\bar{\alpha}$'. By equations between multi-terms, we can express conjunction of equations (e.g. $\operatorname{def}(\alpha, \beta) \equiv \operatorname{def} \alpha \wedge \operatorname{def} \beta$); true will denote $\operatorname{def}()$.

Remark 1 In the simple signature associated to a HasCasl signature according to the translation given in Section 1.4, all operators except the application operators will be ***total constants*** $f : t$, i.e. operators $f : () \rightharpoonup t$ together with an axiom $\operatorname{def} f()$.

In Figure 1.2, we present a set of proof rules for existential equality in a partial λ-theory. The rules are parametrized over a fixed context Γ. We write $\Gamma \rhd \operatorname{def} \bar{\alpha} \vdash \phi$ if an equation ϕ can be deduced from $\operatorname{def} \bar{\alpha}$ in context Γ by means of these rules; in this case, $\Gamma \rhd \operatorname{def} \bar{\alpha} \Rightarrow \phi$ is a ***theorem***. The rules

$$\text{(var)}\ \frac{x : s \text{ in } \Gamma}{\operatorname{def} x} \qquad \text{(st)}\ \frac{\operatorname{def} f(\bar\alpha)}{\operatorname{def} \bar\alpha} \qquad \text{(unit)}\ \frac{x : \mathit{Unit} \text{ in } \Gamma}{x \overset{e}{=} ()}$$

$$\text{(sym)}\ \frac{\bar\alpha \overset{e}{=} \bar\beta}{\bar\beta \overset{e}{=} \bar\alpha} \qquad \text{(tr)}\ \frac{\begin{array}{c}\bar\alpha \overset{e}{=} \bar\beta \\ \bar\beta \overset{e}{=} \bar\gamma\end{array}}{\bar\alpha \overset{e}{=} \bar\gamma} \qquad \text{(cong)}\ \frac{\begin{array}{c}\bar\alpha \overset{e}{=} \bar\beta \\ \operatorname{def} f(\bar\alpha)\end{array}}{f(\bar\alpha) \overset{e}{=} f(\bar\beta)}$$

$$\text{(ax)}\ \frac{\begin{array}{c}(\bar y : \bar t \rhd \operatorname{def} \bar\alpha \Rightarrow \phi) \in \mathcal{A} \\ \bar y : \bar t \text{ in } \Gamma \\ \operatorname{def} \bar\alpha\end{array}}{\phi} \qquad \text{(sub)}\ \frac{\begin{array}{c}\bar y : \bar t \rhd \operatorname{def} \bar\alpha \vdash \phi \\ \operatorname{def}(\bar\alpha[\bar y/\bar\beta]) \\ \operatorname{def}(\bar\beta)\end{array}}{\phi[\bar y/\bar\beta]}$$

$$(\eta)\ \frac{x : \bar t \rightarrow ?u \text{ in } \Gamma}{(\lambda\, \bar y : \bar t \bullet x\, \bar y) \overset{e}{=} x} \qquad (\beta)\ \frac{\bar y : \bar t \text{ in } \Gamma}{(\lambda\, \bar y : \bar t \bullet \alpha)\, \bar y \overset{s}{=} \alpha}$$

$$(\xi)\ \frac{\bar y : \bar t \rhd \alpha \overset{s}{=} \beta}{\lambda\, \bar y : \bar t \bullet \alpha \overset{e}{=} \lambda\, \bar y : \bar t \bullet \beta}$$

Figure 1.2: Deduction rules for existential equality in context Γ

are essentially a version of the calculus presented in [Mog86], adapted for existential (rather than strong) equations. Of course, there is no reflexive law, since $\alpha \overset{e}{=} \alpha$ is false if α is undefined. For conciseness, subderivations are denoted in the form $\Delta \rhd \operatorname{def} \bar\alpha \vdash \phi$, where the context Δ and the assumption $\operatorname{def} \bar\alpha$ are to be understood as *extending* the ambient context and assumptions. E.g. the first premise of rule (sub) reads 'in the context enlarged by $\bar y : \bar t$ and under the additional assumption $\operatorname{def} \bar\alpha$, ϕ is derivable'. ***Strong equations*** $\Delta \rhd \alpha \overset{s}{=} \beta$, or just $\alpha \overset{s}{=} \beta$, are abbreviations for '$\Delta \rhd \operatorname{def} \alpha \vdash \operatorname{def} \beta$ and $\Delta \rhd \operatorname{def} \beta \vdash \alpha \overset{e}{=} \beta$'; in particular, rule (β) is really two rules. Rule (ξ) implies that all λ-terms are defined.

The higher order rules (ξ) and (β) show a slight preference for strong equations. Note, however, that the usual form of the η-equation, $\lambda\, \bar y : \bar t \bullet \alpha(\bar y) = \alpha$, is an ECE, not a strong equation.

A ***translation*** between partial λ-theories $\mathcal{T}_1$ and $\mathcal{T}_2$ with signatures Σ_1 and Σ_2, respectively, is a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ which transforms axioms into theorems — i.e. for every axiom $\Gamma \rhd \operatorname{def} \bar\alpha \Rightarrow \phi$ in $\mathcal{T}_1$, $\sigma\Gamma \rhd \operatorname{def} \sigma\bar\alpha \vdash \sigma\phi$ in $\mathcal{T}_2$.

Remark 2 So-called ***conditioned terms*** $\bar\alpha \operatorname{res} \bar\beta$, which denote the restriction of a multi-term $\bar{\alpha}$ to the domain of a multi-term $\bar{\beta}$ [Bur93, Mog88], can in our setting be coded using projection operators: we can put $\bar{\alpha} \operatorname{res} \bar{\beta} = (\lambda\, \bar{x}, \bar{y} \bullet \bar{x})\,(\bar{\alpha}, \bar{\beta})$. Conditioned terms, in turn, provide a coding for λ-abstraction of multi-terms.

Remark 3 A notion of ***predicates*** is provided in the shape of terms $\Gamma \rhd \alpha : \mathit{Unit}$, for which we write α in place of $\operatorname{def} \alpha$. The sentence $\operatorname{def} \beta$ can be coded as the predicate $(\lambda\, x \bullet *)(\beta)$.

The expressive power of ECEs is greatly increased in the presence of an equality predicate (see also [CO89, Mog86]):

Definition 4 A partial λ-theory has ***internal equality*** if there exists, for each type s, a binary predicate (cf. Remark 3) eq_s such that

$$x, y : s \rhd eq_s(x, y) \Rightarrow x \overset{e}{=} y \quad \text{and} \quad x : s \rhd \mathsf{true} \Rightarrow eq_s(x, x)$$

Such a predicate allows coding conditional equations as ECEs. In combination with λ-abstraction, it gives rise to a full-fledged intuitionistic logic [CO89, LS86, SM02]; this is explained in more detail in Chapter 3.

The notion of model we choose for the partial λ-calculus and thus, in effect, for HasCasl, is that of ***intensional Henkin model***. Briefly, this means that not only may the sets interpreting partial function types fail to contain all set-theoretic partial functions, but they may also contain several elements describing the same set-theoretic function. Henkin models may be described as syntactic λ-algebras modeled on the corresponding notion defined for the total λ-calculus in [BTM85]:

Definition 5 A ***syntactic λ-algebra*** for a partial λ-theory $\mathcal{T}$ is a family of sets $[\![s]\!]$, indexed over all *types* of $\mathcal{T}$, together with partial interpretation functions

$$[\![\Gamma.\, \alpha]\!] : [\![\Gamma]\!] \rightharpoonup [\![t]\!]$$

for each term $\Gamma \rhd \alpha : t$ in $\mathcal{T}$, where $[\![\Gamma]\!]$ denotes the extension of the interpretation to contexts via the cartesian product. This interpretation is subject to the following conditions:

(i) $[\![\Gamma.\, x_i]\!]$, where $\Gamma = (\bar{x} : \bar{s})$, is the i-th projection;

(ii) $[\![\bar{y} : \bar{t}.\, \gamma]\!] \circ [\![\Gamma.\, \beta]\!] = [\![\Gamma.\, \gamma[\bar{y}/\beta]]\!]$, where $\Gamma \rhd \beta : \bar{t}$ is a multi-term, with the interpretation extended to multi-terms in the obvious way;

(iii) whenever $\Gamma \rhd \phi \vdash \alpha \overset{e}{=} \beta$ in $\mathcal{T}$ and $[\![\Gamma.\, \phi]\!](x)$ holds (i.e. is defined), then $[\![\Gamma.\, \alpha]\!](x) = [\![\Gamma.\, \beta]\!](x)$ are defined.

A ***model morphism*** between two syntactic λ-algebras is a family of functions h_s, where s ranges over all types, that satisfies the usual homomorphism condition for partial algebras w.r.t. all terms. A syntactic λ-algebra *satisfies* an ECE $\Gamma \rhd \operatorname{def} \bar{\alpha} \Rightarrow \beta \overset{e}{=} \gamma$ if

$$\llbracket (). \lambda \Gamma \bullet \beta \operatorname{res} \bar{\alpha} \rrbracket = \llbracket (). \lambda \Gamma \bullet \gamma \operatorname{res} \bar{\alpha} \rrbracket \quad \text{and}$$
$$\llbracket (). \lambda \Gamma \bullet \operatorname{def}(\beta, \bar{\alpha}) \rrbracket = \llbracket (). \lambda \Gamma \bullet \operatorname{def} \bar{\alpha} \rrbracket.$$

It is shown in [Sch03] that such models are essentially equivalent to categorical models involving partial cartesian closed categories.

Remark 6 Having to interpret all λ-terms can be avoided by using combinators instead of λ-abstraction. In fact, it is implicit in [Mog88] that syntactic λ-algebras are equivalent to the combinatorically defined λ_p-*algebras* considered there; however, it is unclear whether λ_p-algebras can be finitely axiomatized.

A translation $\sigma : \mathcal{T}_1 \to \mathcal{T}_2$ of partial λ-theories gives rise to a ***reduct functor*** from the model category of $\mathcal{T}_2$ to that of $\mathcal{T}_1$: given a model M of $\mathcal{T}_2$, $M|_\sigma$ interprets each type and each term in $\mathcal{T}_1$ by the interpretation of its translation along σ in M.

1.1.1 Product types

In a first extension step, we add product types to the partial λ-calculus, i.e. essentially promote multi-types to types. In a ***signature with product types*** Σ, the set T of ***types*** is generated from the set S of sorts by the grammar

$$T ::= S \mid T \times \cdots \times T \mid T \to ?T,$$

with types of the form $\bar{s} \to ?t$ coded as $s_1 \times \cdots \times s_n \to ?t$; operators, however, have profiles consisting as before of a multi-type describing their arguments and a result type. Product types $s_1 \times \cdots \times s_n$ are equipped with tuple formation and projection operators $(\_, \ldots, \_) : \bar{s} \rightharpoonup s_1 \times \cdots \times s_n$ and $pr_i : s_1 \times \cdots \times s_n \rightharpoonup s_i$. Terms for Σ are defined as before, but using these new operators. Morphisms of such signatures are defined as for simple signatures; of course, compatibility with operator profiles now refers to an extension of the translation that takes product types into account. A ***partial λ-theory with product types*** is a signature with product types equipped with a set of ECEs, where ECEs can now be restricted to have a definedness condition for a term (rather than a multi-term) as their premise.

The semantics of a partial λ-theory $\mathcal{T}$ with product types is given by a translation into a partial λ-theory $\mathcal{T}'$: the sorts of $\mathcal{T}'$ are the sorts and the non-trivial product types of $\mathcal{T}$. This gives rise to an obvious translation of types in $\mathcal{T}$ to types in $\mathcal{T}'$. The operators of $\mathcal{T}'$ are, then, the operators of $\mathcal{T}$ with accordingly translated profiles. This, in turn, induces to a translation of terms; the axioms of $\mathcal{T}'$ are the correspondingly translated axioms of $\mathcal{T}$, extended by axioms stating that tuple formation and projections are mutual inverses, i.e.

$$\begin{aligned} \bar{x} : \bar{s} \rhd \mathsf{true} \Rightarrow pr_i(x_1, \dots, x_n) \overset{e}{=} x_i \quad \text{and} \\ x : s_1 \times \cdots \times s_n \rhd \mathsf{true} \Rightarrow (pr_1(x), \dots, pr_n(x)) \overset{e}{=} x. \end{aligned}$$

The models of $\mathcal{T}$ are defined to be the models of $\mathcal{T}'$, and an ECE in $\mathcal{T}$ *holds* in such a model M iff its translation to an ECE in $\mathcal{T}'$ holds in M. Translations of partial λ-theories give rise to translations of the corresponding simple signatures and hence to ***reduct functors***.

1.2 Signatures

We now proceed to define actual HasCasl signatures, which will then be translated into simple signatures as defined in the previous section.

As it is standard in higher-order logic, operations are just constants of an appropriate (possibly higher-order) type. Moreover, the type of constants may be *polymorphic*, containing type variables that may be instantiated later on. *Type constructors* map types to types. More generally, also higher-order type constructors are allowed, mapping type constructors to type constructors (where types are regarded as nullary type constructors). Declaration and application of type constructors is subject to correct *kinding.* Kinds can be regarded as sets of type constructors. Higher-order type constructors have higher-order kinds.

As in Haskell, a *type constructor class* is a user-declared subkind of a given kind, such that all members of the constructor class come with a bunch of operations (also called methods). Type constructors may be overloaded with several kinds built from different classes, but only if all these kinds are of the same shape, formalized as *raw kind.*

Finally, *type synonyms* just are abbreviations of more complex types by names.

A gentle introduction into polymorphic types, type constructors, kinds, and type classes is given in [HPF99].

A ***signature*** $\Sigma = (C, \leq_C, T, A, O, \leq)$ consists of:

- a set C of ***type constructor classes*** (or just ***classes***) with assigned ***raw kinds***;

- a ***subclass*** relation $\leq_C$ between classes and ***kinds***
- a set T of ***type constructors*** consisting of their name and a set of kinds called ***profiles***;
- a set A of ***type synonyms***, where a type synonym associates a name to a ***pseudotype*** (i.e. a λ-term at the level of types; see below), its ***expansion***; and
- a set O of ***constant*** symbols with assigned ***type schemes*** (i.e types, possibly with a universal quantification over type variables at the outermost level; see below).
- a ***subtype*** relation between type constructors and pseudotypes.

The sets K and RK of *kinds* and *raw kinds*, respectively, are defined by the grammar

$$\begin{aligned} K &::= C \mid \{V \bullet V \leq P\} \mid \mathit{Type} \mid K \cap K \mid K \to K \mid K^+ \to K \mid K^- \to K \\ RK &::= \mathit{Type} \mid RK \to RK \mid RK^+ \to RK \mid RK^- \to RK, \end{aligned}$$

where *Type* is the kind of all types, P is the set of all pseudotypes (see below), and V is a set of type variables. The ***downset kind*** $\{a \bullet a \leq t\}$ denotes the kind of all subtypes of a given pseudotype t. The $+/-$ superscripts indicate covariant or contravariant dependency on the type arguments, respectively, for purposes of subtyping. A class Cl is associated to its raw kind Kd by writing $Cl : Kd$. The *raw kind* of a kind Kd is obtained by replacing each class occurring in Kd with its raw kind and each downset kind with the raw kind of the corresponding pseudotype as defined below. The formation of the ***intersection kind*** $Kd_1 \cap Kd_2$ is allowed only when Kd_1 and Kd_2 have the same raw kind, which is then also the raw kind of $Kd_1 \cap Kd_2$. We require that, whenever $Cl \leq_C Kd$, then the raw kinds of Cl and Kd are in the subkind relation. The subclass relation is extended to an order relation $\leq_K$ on kinds by the rules shown in Figure 1.3; note that co- and contravariant constructor kinds are subkinds of the corresponding constructor kind without variance information. The rule for intersection kinds works in both directions. By induction over the derivation length, it is shown that $Kd_1 \leq_K Kd_2$ implies that the same relation holds for the associated raw kinds, i.e. that the latter are identical up to possible removal of variance annotations. Note that a class or kind is not necessarily a subkind of its raw kind (e.g., given a class *Ord* of ordered types, $\mathit{Ord} \to \mathit{Ord}$ has raw kind $\mathit{Type} \to \mathit{Type}$, but is not a subkind of that kind.); however, for a class Cl of raw kind *Type*, it is required that $Cl \leq_C \mathit{Type}$.

By writing $t : Kd$, we express that a type t is associated to a kind Kd. We require that all the kinds assigned to a type constructor are of the same raw kind, which is then regarded as the ***raw kind*** of the type constructor (kinds

$$\frac{Cl \leq_C Kd}{Cl \leq_K Kd} \qquad \frac{Kd_1 \leq_K Kd_2 \quad Kd_3 \leq_K Kd_4}{Kd_2^{(+/-/\cdot)} \to Kd_3 \leq_K Kd_1^{(+/-/\cdot)} \to Kd_4}.$$

$$\begin{array}{c} Kd \leq_K Kd_i, i = 1, 2 \\ \hline\hline Kd \leq_K Kd_1 \cap Kd_2 \end{array} \qquad \frac{t_1 \leq t_2}{\{a \bullet a \leq t_1\} \leq_K \{a \bullet a \leq t_2\}}$$

$$\frac{}{Kd_1^{(+/-)} \to Kd_2 \leq_K Kd_1 \to Kd_2}$$

$$\frac{}{Kd \leq_K Kd} \qquad \frac{Kd_1 \leq_K Kd_2 \quad Kd_2 \leq_K Kd_3}{Kd_1 \leq_K Kd_3}.$$

Figure 1.3: Subkinding rules

derivable for the type constructor may have a greater raw kind). There are built-in type constructors

$$\begin{aligned} \_ \times \_ &: \mathit{Type}^+ \to \mathit{Type}^+ \to \mathit{Type}, \\ \_ \to ?\_,\ \_ \to \_ &: \mathit{Type}^- \to \mathit{Type}^+ \to \mathit{Type}, \text{ and} \\ \mathit{Unit} &: \mathit{Type} \end{aligned}$$

for products, partial and total function spaces, and the singleton type, respectively (with, in fact, an n-ary product type constructor $\_ \times \cdots \times \_$, covariant in all arguments, for each n).

A signature induces a set P of ***pseudotypes***, where a pseudotype, formed in a ***type context*** Θ of ***type variables***, is either a type variable, a type constructor, an application, or an abstraction. The type context consists of distinct type variables with assigned ***extended kinds***, denoted $(a_1 : Kd_1, \dots, a_n : Kd_n)$ or, briefly, $(\bar{a} : \overline{Kd})$. Here, an extended kind is a kind, possibly annotated with a variance $(+/-)$ (called its ***outer variance***), as used in argument kinds of constructor kinds $Kd_1 \to Kd_2$.

More precisely, pseudotypes are formed and kinded according to the rules shown in Figure 1.4. A judgement of the form $\Theta \rhd t : Kd$ is to be read 't is a type constructor of kind Kd in context Θ which depends on the variables in Θ with the indicated variance'. The contexts Θ^{-1} and Θ^0 denote Θ with all outer variances reversed or removed, respectively. In the kinding rule for type abstraction, the variance of the abstracted variable in the premise must, of course, be identical to the variance of the argument kind in the conclusion. A pseudotype of kind *Type* is called a ***type***. The (unique) ***raw kind*** of a pseudotype can be calculated by the essentially the same set of rules, with the following modifications:

$$\frac{\begin{array}{c} t : Kd_1 \text{ in } \Sigma \\ Kd_1 \le_K Kd_2 \end{array}}{\Theta \rhd t : Kd_2} \qquad \frac{\begin{array}{c} a : Kd_1^{(+/-/\cdot)} \text{ in } \Theta \\ Kd_1 \le_K Kd_2 \end{array}}{\Theta \rhd a : Kd_2}$$

$$\frac{\begin{array}{c} \Theta \rhd t : Kd_1 \\ \Theta \rhd s : Kd_1 \to Kd_2 \end{array}}{\Theta^0 \rhd s\ t : Kd_2} \qquad \frac{\begin{array}{c} \Theta \rhd t : Kd_1 \\ \Theta \rhd s : Kd_1^+ \to Kd_2 \end{array}}{\Theta \rhd s\ t : Kd_2}$$

$$\frac{\begin{array}{c} \Theta^{-1} \rhd t : Kd_1 \\ \Theta \rhd s : Kd_1^- \to Kd_2 \end{array}}{\Theta \rhd s\ t : Kd_2} \qquad \frac{\begin{array}{c} \Theta, a : Kd_1^{(+/-/\cdot)} \rhd t : Kd_2 \\ Kd_3 \le_K Kd_1 \end{array}}{\Theta \rhd \lambda\, a : Kd_1 \bullet t : Kd_3^{(+/-/\cdot)} \to Kd_2}$$

Figure 1.4: Kinding rules for type constructors

- type constructors are introduced with their raw kind instead of with one of their profiles
- type contexts contain only variables of raw kinds
- exact fits are required where the kinding rules have subkinding constraints, i.e. $\le_K$ is replaced by $=$ throughout.

Note that the raw kind of a type constructor or pseudotype t need not be a derivable kind for t! The corresponding raw kinding judgements are written $\Theta \rhd_{\mathsf{raw}} t : Kd$.

It is easy to show that the kinds derivable for a pseudotype are upwards closed w.r.t the subkind relation (which is why we can require exact fits in the application rules). All kinds derivable for a pseudotype are of its raw kind. Moreover, kinding is invariant under substitution and hence under β-equality (but not under η-equality, which is therefore not imposed on type constructors).

The subtype relation $\le$ between type constructors and pseudotypes is extended to two preorders $\le$ and $\le_*$ on pseudotypes. The intuition behind this distinction is that certain subtypes will be mapped injectively into a supertype (recall that this is assumed for all subtypes in first order CASL), while others may have non-injective coercion functions (e.g., function restriction). The former type of subtype relation is denoted by $\le$, the latter by $\le_*$. In Figure 1.5, this difference shows up in the rule for application of contravariant type constructors, which applies only to the relation $\le_*$.

$$\frac{s \le t \text{ in } \Sigma}{\Theta \rhd s \le t} \qquad \frac{\Theta \rhd s \le t}{\Theta \rhd s \le_* t} \qquad \frac{\begin{array}{c}\Theta \rhd_{\mathsf{raw}} t : Kd_1^+ \to Kd_2 \\ \Theta \rhd s_1 \le s_2\end{array}}{\Theta \rhd t\ s_1 \le t\ s_2}$$

$$\frac{\begin{array}{c}\Theta \rhd_{\mathsf{raw}} t : Kd_1^+ \to Kd_2 \\ \Theta \rhd s_1 \le_* s_2\end{array}}{\Theta \rhd t\ s_1 \le_* t\ s_2} \qquad \frac{\begin{array}{c}\Theta \rhd_{\mathsf{raw}} t : Kd_1^- \to Kd_2 \\ \Theta \rhd s_2 \le_* s_1\end{array}}{\Theta \rhd t\ s_1 \le_* t\ s_2}$$

$$\frac{\Theta \rhd t_1 \le t_2}{\Theta \rhd t_1\ s \le t_2\ s} \qquad \frac{\Theta \rhd t_1 \le_* t_2}{\Theta \rhd t_1\ s \le_* t_2\ s}$$

$$\frac{\begin{array}{c}\Theta, a : Kd_1 \rhd t \le s \\ Kd_1 \le_K Kd_2\end{array}}{\lambda\, a : Kd_2 \bullet t \le \lambda\, a : Kd_1 \bullet s} \qquad \frac{\begin{array}{c}\Theta, a : Kd_1 \rhd t \le_* s \\ Kd_1 \le_K Kd_2\end{array}}{\lambda\, a : Kd_2 \bullet t \le_* \lambda\, a : Kd_1 \bullet s}$$

Figure 1.5: Subtyping rules for pseudotypes

The subtyping relation implicitly contains

$$\_ \to \_ \le \_ \to ?\_,$$

i.e. total functions can be regarded as partial when required. From this extended relation, the preorders on the set of pseudotypes are defined by the rules in Figure 1.5. Like kinding judgements, subtyping judgements are parametrized by a type context; however, for subtyping, the outer variances of type variables are irrelevant.

Type synonyms are intended as shorthands for pseudotypes; they are *not* meant as a means of constructing recursive types. More formally, expansion of type synonyms is required to be non-recursive, i.e. the relation 'the expansion of a contains b' on synonyms must be well-founded. A ***named pseudotype*** (as opposed to an *anonymous* pseudotype) is a pseudotype that can be constructed from type constructors, type synonyms, and its type context using only application (*not* λ-abstraction). Of course, any pseudotype can be made into a named pseudotype by just introducing suitable synonyms.

HasCasl features ***ML-polymorphism***, i.e. constants of types that contain type variables, implicitly or explicitly universally quantified *on the outermost level*. Thus, the types are complemented by ***type schemes***: A type scheme consists of a type context Θ and a *named* type t in that context (the variables of the type scheme will stem either from an explicit quantification or from a global or local variable declaration), together with a ***coherence flag*** stating whether or not instances are required to be coherent w.r.t. subtyping (typically, recursively defined polymorphic functions will be coherent, while predicates and functions used only for specification purposes may fail to be so); see also Section 1.3. Such a type scheme is written $\forall\Theta \bullet t$, with the coherence flag left implicit. Types will be regarded as type schemes with empty type context.

The ***constant symbols*** are given, like symbols in first order CASL, by their ***names*** with associated ***profiles***, the difference being that a profile is now represented by a single type scheme. A constant symbol with name f and profile t is written $f : t$. An operator is called ***monomorphic*** if t is a type, otherwise ***polymorphic***. The set O of constants contains the following ***distinguished constants***:

- the unique inhabitant of the unit type, $() : \mathit{Unit}$;
- for each partial or total function type $s \to ?\ t$ or $s \to t$ an implicit ***application operator*** of profile $((s \to ?\ t) \times s) \to ?\ t$ or $((s \to t) \times s) \to t$, respectively;
- for each pair $s \leq t$ of types, a ***downcast*** operator $\_\ \mathit{as}\ s : t \to ?s$

Note that constant symbols may be ***overloaded***, i.e. different profiles can be associated to the same name. To ensure that there is no ambiguity in sentences at this level, constant symbols f are always ***qualified*** by their profile t when used, written f_t. (The language considered in Chapter 2 allows the omission of such qualifications when these are unambiguously determined by the context.) In fact, we require signatures to be *embedding-closed* (see also [SMT$^+$01]), i.e. the profiles associated to a given name must be upwards closed under $\leq_*$.[1] (Of course, embedding-closure is provided *implicitly*, so that the user is not actually required to specify all these profiles). This also makes sense of the profile of the upcast operator: $\_ : s$ implicitly has profiles $u \to t$ for all u, t with $u \leq s \leq t$.

A ***signature morphism***

$$\sigma : (C_1, \leq_C, T_1, A_1, O_1, \leq) \to (C_2, \leq_C, T_2, A_2, O_2, \leq)$$

consists of mappings from C_1 to C_2, from T_1 to $T_2 + A_2$, from A_1 to A_2, and from O_1 to O_2. These maps are required to preserve

- raw kinds of classes and type constructors
- the subclass relation in the sense that $\mathit{Cl} \leq_C \mathit{Kd}$ implies $\mathit{Cl} \leq_K \mathit{Kd}$.
- kinding judgements for type constructors in the sense that assigned kinds are mapped to derivable kinding judgements,
- expansions of type synonyms,

[1]This requirement makes it superfluous to define overloading relations as in CASL.

- profiles of constant symbols,
- all distinguished constants, and
- the subtyping relation $\leq$, again in the sense that subtyping judgements must be derivable in the target signature.

Moreover, distinguished constants must also be reflected (this in order to avoid ambiguities in the notation of signature morphisms). (This means that we could have omitted them for purposes of describing the signature category; they are included in the set of constants mainly in order to simplify the presentation of the typing and deduction rules.)

Remark 7 Note that the above definition explicitly allows type constructors to be mapped to type synonyms; this allows instantiating type constructors with pseudotypes, albeit at the cost of having to define an synonym first. A consequence is that the signature category fails to be cocomplete (while its non-full subcategory consisting of the signature morphisms that map type constructors to type constructors *is* cocomplete, being essentially the category of models of a Horn theory). However, the pushouts required for instantiating parametrized specifications do exist, which is all that is needed for HasCasl structured specifications.

1.3 Models

The model semantics of HasCasl is split into two levels. The first level yields an institution only for the fragment of HasCasl without polymorphism, while in the general case one obtains an *rps preinstitution* [SS93], i.e. the satisfaction condition holds only in the direction leading from the extended to the reduced model. This is remedied at the second level by means of a general mechanism for transforming rps preinstitutions into institutions, which we define but do not discuss in detail here; a fuller presentation is given in [SM04].

At the ***first level***, the models of a signature Σ are defined by a translation of Σ into a partial λ-theory with products $\mathsf{Th}(\Sigma)$ to be defined in Section 1.4 — i.e. the models of Σ are defined to be the syntactic λ-algebras for $\mathsf{Th}(\Sigma)$, correspondingly for model morphisms. In the interest of compatibility with first-order Casl, the carrier sets of all types are additionally required to be non-empty. Note that every Casl model of a signature Σ can be extended to a HasCasl model of Σ regarded as a HasCasl signature. However, this extension will in general not be unique; in particular, there is always a free extension, where function types are in a sense minimal, and a standard extension with function types interpreted maximally, i.e. by full function sets.

At the ***second level***, models of a signature Σ are pairs (N, σ), where σ is a ***derived signature morphism*** [Sch] $\Sigma \to \Sigma_2$ into a further signature Σ_2, and N is a model of Σ_2 at the first level. Here, derived signature morphisms generalize signature morphisms in that they may map type constructors to pseudotypes, including types formed by subtype comprehension, and constant symbols to terms. ***Subtype comprehension***, in turn, refers to the formation of types of the form $\{x : t \mid \phi\}$, where t is a type and ϕ is a formula. The reduct of (N, σ) along a signature morphism τ into Σ is $(N, \sigma \circ \tau)$. This construction, together with the definition of satisfaction given in Section 1.6, ensures that the satisfaction condition holds.

1.4 Translation of HasCasl signatures into partial λ-theories

An ***instance*** of a kind is essentially a closed named pseudotype of that kind, taken modulo β-equality; in addition to the usual type forming operators, an instance may however include the use of subtype formation by definedness assertions for terms. The sorts of $\mathsf{Th}(\Sigma)$ are the ***loose types*** of Σ, where a loose type is an application of a type constructor other than the built-in type constructors $\times$, $\to?$, $\to$, and *Unit*, to an instance of its argument kind. This gives rise to a recursively defined translation of kind instances in Σ to types in $\mathsf{Th}(\Sigma)$ (and conversely), since in β-normal types, λ-abstractions can only occur nested inside loose types. We leave this translation implicit in the notation.

The operators of $\mathsf{Th}(\Sigma)$ are defined to be

- for each operator f with profile $\forall \bar{a} : \overline{Kd} \bullet t$ a family of total constants (cf. Remark 1) $f_{\bar{s}} : t[\bar{s}/\bar{a}]$, indexed over all instances $\bar{s} : \overline{Kd}$;
- for each pair (s, t) of types in $\mathsf{Th}(\Sigma)$ such that $s \leq_* t$ holds for the corresponding types in Σ, an ***embedding operator***

$$em_{s,t} : s \rightharpoonup t.$$

Finally, $\mathsf{Th}(\Sigma)$ has the following axioms (all expressible as ECEs):

- ***coherence of subtyping*** essentially as in Casl;
- ***overloading axioms*** stating for each pair (s, t) of types in $\mathsf{Th}(\Sigma)$ and each constant $c : s$ that

$$em_{s,t}(c : s) = c : t$$

(where the profile $c : t$ is in Σ by embedding closure).

- injectivity of subtype embeddings $em_{s,t}$ for $s \leq t$ (*not*, more generally, for $s \leq_* t$), expressed by their mutual inverse property with the corresponding downcast operators (also as in CASL);
- ***coherence*** of correspondingly flagged polymorphic operators w.r.t. subtyping: if $f : \forall \bar{a} : \overline{Kd} \bullet t$ is a coherent polymorphic operator, and $\bar{s}$ and $\bar{u}$ are instances of $\overline{Kd}$ such that $t[\bar{s}/\bar{a}] \leq_* t[\bar{u}/\bar{a}]$, then

$$f_{\bar{u}} = em_{t[\bar{s}/\bar{a}],t[\bar{u}/\bar{a}]}(f_{\bar{s}}).$$

A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ induces a morphism $\mathsf{Th}(\sigma) : \mathsf{Th}(\Sigma_1) \to \mathsf{Th}(\Sigma_2)$ of simple signatures with products. The ***reduct functor*** for σ is defined to be that of $\mathsf{Th}(\sigma)$.

1.5 Sentences

Sentences for a signature Σ are built from atomic formulae using quantification (over sorted variables) and logical connectives, as well as *outer* universal quantification over type constructor variables. An inner quantification over a variable makes a hole in the scope of an outer quantification over the same variable, regardless of the types (or kinds) of the variables. Quantification over type variables produces a local type context. Implication may be taken as primitive (together with the always-false formula), the other connectives being regarded as derived.

The ***atomic formulae*** are:

- fully-qualified terms of sort *Unit*, regarded as predicates qua implicit definedness assertions
- definedness assertions def _ for fully-qualified terms
- existential and strong equations $\_ \overset{e}{=} \_$ and $\_ = \_$, respectively, between fully-qualified terms of the same sort.

Here, a *fully-qualified term* (or, when no confusion is likely, just a ***term***) is a term in $\mathsf{Th}(\Sigma)$ (with the context determined by enclosing quantifications). A fully-qualified term in type context Θ (arising from enclosing universal quantifications over type variables) is a fully-qualifed term in $\Sigma + \Theta$, where $\Sigma + \Theta$ is obtained by extending Σ with the variables in Θ, regarded as type constructors of the appropriate kinds (with raw kinds determined by their unique kinds).

As syntactical sugar over these sentences, one has the following additional features:

- for each pair (s,t) of types in $\mathsf{Th}(\Sigma)$, an elementhood operator $\_ \in s : t \to ?\,\mathit{Unit}$ abbreviating $\lambda\, x : t \bullet \operatorname{def} x \ \mathit{as}\ s$;
- a total λ-abstraction $\lambda\, \bar{x} : \bar{s} \bullet !\, \alpha$ which abbreviates a downcast of the partial λ-abstraction to the type of total functions;
- syntactical support for emulation of ***non-strict functions*** by the ***procedural lifting method***: let $?t$ abbreviate the type $\mathit{Unit} \to ?t$. We admit terms formed using two additional typing rules: a function that expects an argument of type t (possibly as part of a product type) may be applied to a term α of type $?t$, which is then implicitly replaced by $\alpha()$; conversely, a function that expects an argument of type $?t$ accepts arguments β of type t, which are implicitly replaced by $\lambda\, x : \mathit{Unit} \bullet \beta$ (where x is a fresh variable).
- ***let-terms***: for a term $\alpha : t$ in type context $\Theta = (\bar{a} : \overline{Kd})$, a variable x, and a term β in ***operator context*** $x : \forall \Theta \bullet t$, one has a term

  $$\mathit{let}\ \forall \Theta \bullet x = \alpha\ \mathit{in}\ \beta;$$

  here, a term in operator context $x : \forall \Theta \bullet t$ is a term in the signature obtained from Σ by adding a constant $x : \forall \Theta \bullet t$. Such a let-term abbreviates β with all occurrences $x_{\bar{s}}$ of x substituted by $\alpha[\bar{s}/\bar{a}]$. Repeated bindings $\mathit{let}\ \forall \Theta_1 \bullet x_1 = \alpha_1\ \mathit{in}\ \mathit{let}\ \forall \Theta_2 \bullet x_2 = \alpha_2\ \mathit{in}\ \dots$ are abbreviated as $\mathit{let}\ \forall \Theta_1 \bullet x_1 = \alpha_1;\ \forall \Theta_2 \bullet x_2 = \alpha_2\ \mathit{in}\ \dots$.
- ***recursive datatypes*** are syntactical sugar for the usual no-junk-no-confusion axioms; details are laid out in Section 2.7.

1.6 Satisfaction

At the first level of the model semantics (cf. Section 1.3), the ***satisfaction*** of a sentence in a model M is determined as usual by the holding of its atomic formulae w.r.t. assignments of (defined) values to all the variables that occur in them, the values assigned to variables of sort s being in s^M. The value of a term w.r.t. a variable assignment may be undefined, due to the application of a partial function during the evaluation of the term. Note, however, that the satisfaction of sentences is 2-valued (as is the holding of open formulae with respect to variable assignments). The satisfaction of a universal quantification over type variables is defined as satisfaction of all instances of that formula (this is possible because quantifications over type variables are allowed only at the outermost level).

A term of type *Unit* holds as an atomic formula if it is defined in M. A definedness assertion concerning a term holds iff the value of the term is defined (thus it corresponds to the application of an operator $a \rightharpoonup \mathit{Unit}$ to the term). An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined.

Since the type context has been 'substituted away', every term α occurring in the expanded formulae is a term in $\mathsf{Th}(\Sigma)$. The value of α is determined as follows: the given variable assigment for the context Γ is an element x of $[\![\Gamma]\!]$ (cf. Definition 5); the value of α is defined to be $[\![\Gamma.\,\alpha]\!](x)$.

At the second level of the semantics, a model (N, σ) of Σ satisfies a sentence ϕ iff N satisfies the translated sentence $\sigma\phi$ at the first level.

1.7 Embedding of Casl into HasCasl

There is a canonical embedding of CASL into HASCASL. It preserves the semantics of structured specifications, including free specifications when restricted to CASL without subsorting (but excluding sort generation constraints), in the sense that the set of HASCASL models of a CASL specification, when restricted to the basic types, coincides with the original model semantics in CASL.

Chapter 2

Basic Constructs

This chapter indicates the abstract and concrete syntax of the constructs of basic specifications without internal logic and general recursion, and describes their intended interpretation.

For an introduction to the form of grammar used here to define the abstract syntax of language constructs, see Appendix A, which also provides the complete grammar defining the abstract syntax of the entire HasCasl specification language. For the ASCII input of mathematical symbols displayed in LaTeX we refer to Section C.4.

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

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

$$BI_1 \ldots BI_n$$

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

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

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

```
BASIC-ITEMS      ::= CLASS-ITEMS | SIG-ITEMS
                   | GENERATED-ITEMS | FREE-DATATYPES
                   | GENERIC-VARS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS
```

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

plural-keyword X_1; ... X_n;

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

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

A list of class declarations `CLASS-ITEMS` determines part of a signature. A list of signature declarations and definitions `SIG-ITEMS` determines part of a signature and possibly some sentences. A generation construct `GENERATED-ITEMS` determines part of a signature, together with some sentences stating term generatedness. A `FREE-DATATYPE` construct determines part of a signature together with some sentences. A list of variable declaration items `GENERIC-VARS` determines object variables (each with a type) and type variables (each with a kind) that are implicitly universally quantified in the subsequent axioms of the enclosing basic specification; note that variable declarations do not contribute to the signature of the specification in which they occur. A `LOCAL-VAR-AXIOMS` construct restricts the scope of the variable declarations to the indicated list of axioms. (Variables may also be declared locally in individual axioms, by explicit quantification.) An `AXIOM-ITEMS` construct determines a set of sentences.

Signature Declarations

```
SIG-ITEMS ::= TYPE-ITEMS | OP-ITEMS | FUN-ITEMS | PRED-ITEMS
```

A list `TYPE-ITEMS` of type declarations determines one or more type constructors. A list `OP-ITEMS` or a list `FUN-ITEMS` of operation declarations and/or definitions determines one or more constant symbols, and possibly some sentences; similarly for a list `PRED-ITEMS` of predicate declarations and/or definitions. Constant and predicate symbols may be overloaded, being declared with several different profiles in the same local environment.

The difference between `OP-ITEMS` and `FUN-ITEMS` is that the former entail coherence of different polymorphic instances with respect to subtyping, whereas the latter (as well as `PRED-ITEMS`) do not.

2.1 Classes

A *type constructor class* is a user-declared subkind of a given kind, such that all members of the constructor class come with a bunch of operations (also called methods) and satisfy certain *interface axioms*.

```
       CLASS-ITEMS ::= SIMPLE-CLASS-ITEMS | INSTANCE-CLASS-ITEMS
SIMPLE-CLASS-ITEMS ::= class-items CLASS-ITEM+
```

A list `SIMPLE-CLASS-ITEMS` of class declarations determines one or more classes. It is written:

classes $CI_1; \ldots CI_n;$

`INSTANCE-CLASS-ITEMS` are described below.

2.1.1 Class Declarations

```
CLASS-DECL        ::= SIMPLE-CLASS-DECL | SUBCLASS-DECL
SIMPLE-CLASS-DECL ::= class-decl CLASS-NAME+
SUBCLASS-DECL     ::= subclass-decl CLASS-NAME+ KIND
```

A simple class declaration `SIMPLE-CLASS-DECL` is written:

$c_1, \ldots, c_n$

It declares classes $c_1, \ldots, c_n$ as subkinds of the kind *Type*.

A subclass declaration `SUBCLASS-DECL` is written:

$c_1, \ldots, c_n < k$

It declares classes $c_1, \ldots, c_n$ as subkinds of the kind k. In the case that k is another class, the c_i are also called ***subclasses*** of k.

2.1.2 Class Items

```
CLASS-ITEM ::= class-item CLASS-DECL BASIC-ITEMS*
```

A class item `CLASS-ITEM` consisting of a class declaration *CD* and a list of basic items *BIs* is written:

CD{ *BIs* }

It expands to the class declaration *CD* followed by the basic items *BIs*, and attaches all the axioms generated by *BIs* as interface axioms to the classes in *CD*.

2.1.3 Class instance declarations

```
INSTANCE-CLASS-ITEMS ::= instance-class-items CLASS-ITEM+
```

A class instance declaration `INSTANCE-CLASS-ITEMS` is written:

class instances CI_1; ... CI_n;

Like a class declaration, a class instance declaration declares one or more classes, but additionally generates a semantic annotation corresponding to Casl's **%implies** annotation, namely that each class being declared as a subclass of another class satisfies all interface axioms of the superclass. More precisely, the interface axioms have to be satisfied for all models of the local environment and (in the case of type constructor classes) all argument types satisfying the interface axioms of their respective classes.

2.2 Kinds

A kind can be regarded as a set of type constructors.

```
KIND             ::= TYPE-UNIVERSE | CLASS-KIND | INTERSECTION-KIND
                   | DOWNSET-KIND | FUN-KIND
TYPE-UNIVERSE    ::= type-universe
CLASS-KIND       ::= class-name CLASS-NAME
INTERSECTION-KIND::= intersection KIND+
DOWNSET-KIND     ::= downset TYPE
FUN-KIND         ::= fun-kind EXT-KIND KIND
EXT-KIND         ::= ext-kind KIND VARIANCE
VARIANCE         ::= covariant | contravariant | invariant
```

The kind `type-universe` of all ground types is written *Type*. A `CLASS-KIND` is written c and just denotes the class c. An intersection kind `INTERSECTION-KIND` is written

$(k_1, \ldots, k_n)$

and denotes the intersection of the kinds $k_1, \ldots, k_n$. A `DOWNSET-KIND` is written

$\{a \bullet a \le t\}$

and denotes the set of all type constructors that are less than or equal to (in the subtype relation) the type constructor t.

Finally, a `FUN-KIND` is written

$k_1^+ \to k_2$, or $k_1^- \to k_2$ or $k_1 \to k_2$

It denotes the kind of all type constructors from k_1 to k_2. The ***variances*** `covariant`, `contravariant` and `invariant` are written as $+$, $-$ or no superscript to the argument kind. (In ISO Latin-1 or ASCII input syntax, instead of superscripts, the $+$ or $-$ is written directly after the kind.) The variance is used in the subtyping rules when inheriting subtyping from arguments to results of type constructors. Kinds with $+$ and $-$ annotations are also called ***extended kinds***.

2.3 Type constructors

```
          TYPE-ITEMS ::= SIMPLE-TYPE-ITEMS | INSTANCE-TYPE-ITEMS
   SIMPLE-TYPE-ITEMS ::= type-items TYPE-ITEM+
 INSTANCE-TYPE-ITEMS ::= instance-type-items TYPE-ITEM+
```

A list `SIMPLE-TYPE-ITEMS` of type constructor declarations is written:

types TI_1; ... TI_n;

while a list `INSTANCE-TYPE-ITEMS` is written:

type instances TI_1; ... TI_n;

Both forms declare one or more type constructors and/or type synonyms, and possibly some subtype relations and axioms. A type instance declaration additionally generates a semantic annotation, namely that each type constructor being declared as a member of some class satisfies all interface axioms of that class. More precisely, the interface axioms have to be satisfied for all models of the local environment and (in the case of true type constructors) all argument types satisfying the interface axioms of their respective classes.

```
TYPE-ITEM ::= TYPE-DECL | SYNONYM-TYPE | DATATYPE-DECL
            | SUBTYPE-DECL | ISO-DECL | SUBTYPE-DEFN
```

For the description of `DATATYPE-DECL` see Section 2.7.

2.3.1 Type Constructor Declarations

```
TYPE-DECL ::= type-decl TYPE-NAME+ KIND
```

A type constructor declaration `TYPE-DECL` is written:

$tn_1, \ldots, tn_n : k$

It declares each of the type constructors in the list tn_1, ..., tn_n with kind k.

The concrete syntax for `TYPE-DECL` further supports a notation where the kind may be omitted. In this case all type constructors are assumed to have kind *Type*. Furthermore a type constructor declaration may be written via a `TYPE-PATTERN`, that is:

$tn : k_1 \rightarrow \ldots \rightarrow k_n \rightarrow k$

may be written:

$tn\ a_1\ \ldots\ a_n : k$

where a_i are type arguments with extended kinds k_i.

2.3.2 Type Synonym Declarations

```
SYNONYM-TYPE ::= synonym-type TYPE-NAME TYPE-ARG* TYPE
    TYPE-ARG ::= type-arg TYPEVAR EXT-KIND
```

A type synonym declaration `SYNONYM-TYPE` is written:

$tn\ a_1\ \ldots\ a_n := t$

It declares tn with type arguments $a_1\ \ldots\ a_n$ to be a synonym for the type t, where the a_i may occur within t.

A type argument `TYPE-ARG` is written $tv : k$ and declares the type variable tv to have the extended kind k. The concrete syntax for `TYPE-ARG` allows to omit the kind for variables of kind *Type*. Furthermore the variance may be part of the type variable.

2.3.3 Subtype Declarations

```
SUBTYPE-DECL ::= subtype-decl TYPE-NAME+ TYPE
```

A subtype declaration `SUBTYPE-DECL` is written

$tn_1, \ldots, tn_n < t$

It declares all the type constructors $tn_1, \ldots, tn_n$. The kind for these type constructors is taken from the type t which must already be declared in the local environment. The tn_i must be distinct and must not occur in the type t.

For compatibility with Casl, an undeclared type name t will be treated as declaration with kind *Type*.

When a subtype declaration occurs in a generation construct, the embedding and projection operations between the subtype(s) and the supertype are treated as declared operations with regard to the generation of types.

Introducing an embedding relation between two types may cause operation symbols to become related by the overloading relation, so that values of terms become equated when the terms are identical up to embedding. Moreover, new operation profiles are generated while closing the signature under composition with embeddings.

2.3.4 Isomorphism Declarations

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

An isomorphism declaration `ISO-DECL` is written:

$$tn_1 = \ldots = tn_n = tn$$

It declares all the type constructors $tn_1, \ldots, tn_n$, as well as their embeddings as subtypes of each other including tn. The tn_i must be distinct. The kind is taken from tn that must have been declared before.

Again, for compatibilty with CASL, an undeclared type name tn will be declared with kind *Type*.

2.3.5 Subtype Definitions

```
SUBTYPE-DEFN ::= subtype-defn TYPE-NAME TYPE-ARG* VAR TYPE FORMULA
```

A subtype definition `SUBTYPE-DEFN` is written:

$$tn\ a_1\ \ldots\ a_n = \{v : t \bullet F\}$$

It provides an explicit specification of the values of the subtype $tn\ a_1\ \ldots\ a_n$ of t, in contrast to the implicit specification provided by using subtype declarations and overloaded operation symbols. The a_i may occur in t.

The subtype definition declares the type constructor tn; it declares the embedding of $tn\ a_1\ \ldots\ a_n$ as a subtype of t, which must already be declared in the local environment; and it asserts that the values of $tn\ a_1\ \dots\ a_n$ are precisely (the projection of) those values of the variable v from t for which the formula F holds.

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

Note that the terms of type t cannot generally be used as terms of type $tn\ a_1\ \dots\ a_n$. But they can be explicitly projected to $tn\ a_1\ \dots\ a_n$, using a cast.

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

2.4 Types

```
TYPE        ::= TYPE-NAME | TYPE-APPL
              | PRODUCT-TYPE | LAZY-TYPE | KINDED-TYPE | FUN-TYPE
```

Types are constructed from type constructors and their applications. A type name is uniquely identified as variable or constructor and uniquely associated to a raw kind, because type names can only be overloaded for kinds with the same raw kind. PRODUCT-TYPE, LAZY-TYPE and FUN-TYPE are special type applications and all their type components are expected to have the raw kind *Type*.

2.4.1 Type Applications

```
TYPE-APPL  ::= type-appl TYPE TYPE
```

Types can be applied by juxtaposition. The concrete syntax allows to put types in parentheses; furthermore, *Unit* is a built-in type consisting of the emtpy tuple only, *Pred a* is a built-in type synonym for $a \to ?$ *Unit*, and *Logical* is a built-in type synonym for *Pred Unit*.

Product Types

```
PRODUCT-TYPE ::= product-type TYPE+
```

Product types denote the types for tuples and are written:

$t_1 \times \ldots \times t_n$

In contrast to Casl, product types (and tuples) may be nested using parentheses.

Lazy Types

```
LAZY-TYPE    ::= lazy-type TYPE
```

A lazy type is built using an application of a special type constructor '?', see Section 1.5. It is written:

$?t$

Note that in function and datatype definitions a '?' following a colon is interpreted as partiality of the corresponding function type.

Function Types

```
FUN-TYPE   ::= fun-type TYPE ARROW TYPE
ARROW      ::= partial-fun | total-fun
             | cont-partial-fun | cont-total-fun
```

Function types denote the single argument type and the result type of a function. By using a product type as argument, multi-argument first order functions can be typed. A function arrow as infix symbol is right associative and binds weaker than the symbol $\times$ of product types – for compatibility with first-order Casl. Other prefix applications bind even stronger. The four different flavors of functions are written:

$t_a \rightarrow ?\ t_r$

$t_a \rightarrow t_r$

$t_a \xrightarrow{\text{cont}} ?\ t_r$

$t_a \xrightarrow{\text{cont}} t_r$

The long arrows denote total or partial *continuous* functions.

2.4.2 Kinded Types

```
KINDED-TYPE ::= kinded-type TYPE KIND
```

A type can be restricted to a specific kind instance k that must conform to the inferred raw kind. A kinded type is written:

$t : k$

or

$t < d$

for t of the downset kind $\{a \bullet a \leq d\}$.

Type Schemes

```
TYPESCHEME ::= typescheme TYPE-ARG* TYPE
```

A type scheme `TYPESCHEME` with some type variables is written:

forall a_1 ; . . . ; a_n $\bullet$ t

When the list of type variables is empty, the type is simply written 't'.

The type scheme is polymorphic over the type variables a_1,. . . ,a_n, which may occur in t.

The concrete syntax for `TYPE-VAR-DECLS` in type schemes supports an abbreviated notation. Type variables with the same kind may be comma separated.

2.5 Operations

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

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

ops OI_1; . . . OI_n;

Similarly, a list `FUN-ITEMS` of non-coherent operation declarations and definitions is written:

funs OI_1; . . . OI_n;

The difference between `OP-ITEMS` and `FUN-ITEMS` is that the former entail coherence of different polymorphic instances with respect to subtyping, whereas the latter do not.

A declaration or definition of an operation symbol implicitly leads to declarations of the same operation name with all profiles that are obtained from the declared profile by composing with subtype injections (embedding-closure).

2.5.1 Operation Declarations

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

An operation declaration OP-DECL is written:

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

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

$f_1, \ldots, f_n : t$

It declares each operation name f_1, ..., f_n as a constant with profile as specified by the type scheme t, and as having the attributes A_1, ..., A_m (if any).

Operation Attributes

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

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

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

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

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

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

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

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

$f(x, x) = x$

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

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

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

The declaration enclosing an operation attribute is ill-formed unless the operation has exactly one pair argument, with both components of the same sort, which, except in the case of commutativity, also has to be the same as the result sort.

2.5.2 Operation Definitions

```
OP-DEFN          ::= op-defn OP-NAME TYPE-ARG* OP-HEAD TERM
OP-HEAD          ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD    ::= total-op-head   TUPLE-ARG* TYPE
PARTIAL-OP-HEAD  ::= partial-op-head TUPLE-ARG* TYPE
TUPLE-ARG        ::= tuple-arg VAR-DECL*
VAR-DECL         ::= var-decl VAR TYPE
```

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

$$f(vd_{11}\ ;\ \ldots;\ vd_{1m_1}) \ldots (vd_{n1}\ ;\ \ldots;\ vd_{nm_n}) : t = T$$

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

$$f : t = T$$

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

$$f(vd_{11}\ ;\ \ldots;\ vd_{1m_1}) \ldots (vd_{n1}\ ;\ \ldots;\ vd_{nm_n}) :?t = T$$

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

$$f :?t = T$$

It declares the operation name f as a total, respectively partial operation, with a profile

$$t_{11} \times \ldots \times t_{1m_1} \to \ldots \to t_{n1} \times \ldots \times t_{nm_n} \to t$$

respectively

$$t_{11} \times \ldots \times t_{1m_1} \to \ldots \to t_{n1} \times \ldots \times t_{nm_n} \to ?\ t$$

It also asserts the strong equation:

$$f(v_{11}, \ldots, v_{1m_1}) \ldots (v_{n1}, \ldots, v_{nm_n}) = T$$

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

As in Casl variable declarations within a tuple may be abbreviated. The concrete syntax for `VAR-DECL` allows comma separated variables having the same type (see Section 2.9).

For polymorphic operations the type variables for the final type scheme may be given in square brackets following the name f.

$$f[a_1\ ;\ \ldots;\ a_n](vd_{11}\ ;\ \ldots;\ vd_{1m_1})\ldots(vd_{n1}\ ;\ \ldots;\ vd_{nm_n}) : t = T$$

Again, type variables with the same kind may also be comma separated.

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

2.6 Predicates

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

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

preds $PI_1;\ \ldots\ PI_n;$

A declaration or definition of a predicate symbol implicitly leads to declarations of the same predicate name with all profiles that are obtained from the declared profile by composing with subtype injections (embedding-closure). Predicate declarations and definitions are provided only for the sake of compatibility with first-order Casl; they are equivalent to declarations or definitions of operators of type *Pred t*.

2.6.1 Predicate Declarations

```
PRED-DECL ::= pred-decl OP-NAME+ TYPESCHEME
```

A predicate declaration `PRED-DECL` is written:

$p_1, \ldots, p_n : t$

It declares each name p_1, …, p_n as a predicate. Such a predicate takes exactly one argument, that may be a tuple.

2.6.2 Predicate Definitions

```
PRED-DEFN ::= pred-defn OP-NAME TYPE-ARG* TUPLE-ARG FORMULA
```

A definition `PRED-DEFN` of a polymorphic predicate with a single tuple arguments is written:

$p[a_1\ ;\ \ldots;\ a_n](vd_1\ ;\ \ldots;\ vd_m) \Leftrightarrow F$

When the predicate definition is not polymorphic, it is written:

$p(vd_1\ ;\ \ldots;\ vd_m) \Leftrightarrow F$

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

$p \Leftrightarrow F$

It also asserts the equivalence:

$p(v_1, \ldots, v_m) \Leftrightarrow F$

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

2.7 Datatype Declarations

The order of datatype declarations as part of a `TYPE-ITEMS` or a `FREE-DATATYPE` is *not* significant: there is ***non-linear visibility*** of the declared data types in a list. The visibility exactly corresponds to that of Casl.

Datatype Declarations

```
DATATYPE-DECL ::= datatype-decl TYPE-NAME TYPE-ARG* ALTERNATIVE+
                                                    CLASS-NAME*
```

A polymorphic datatype declaration `DATATYPE-DECL` is written:

$t\ a_1 \ldots a_m ::= A_1 \mid \ldots \mid A_n$

It declares the data type constructor t such that $t\ a_1 \ldots a_m$ has kind *Type*. For each alternative construct $A_1, \ldots, A_n$, it declares the specified constructor and selector operations, and determines sentences asserting the expected relationship between selectors and constructors. All types used in an alternative construct must be declared in the local environment (which always includes the type declared by the datatype declaration itself).

Note that a datatype declaration as part of `TYPE-ITEMS` allows models where the ranges of the constructors are not disjoint, and where not all values are the results of constructors. This looseness can be eliminated in a general way by use of free extensions in structured specifications, or by use of free datatypes. Some of the unreachable values can be eliminated also by the use of generation constraints.

Like in Haskell, free data types may be declared to belong to one or more type classes, with automatically generated axiomatization of the instances (this is a limited form of polytypic specification). In HasCasl, only the special type classes introduced in Chapter 6 may be used here.

$t\ a_1 \ldots a_m ::= A_1 \mid \ldots \mid A_n$ **deriving** $c_1, \ldots, c_l$

Alternatives

```
       ALTERNATIVE ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT | SUBTYPES
   TOTAL-CONSTRUCT ::= total-construct   OP-NAME TUPLE-COMPONENT*
 PARTIAL-CONSTRUCT ::= partial-construct OP-NAME TUPLE-COMPONENT+
   TUPLE-COMPONENT ::= tuple-component COMPONENTS+
```

A total constructor `TOTAL-CONSTRUCT` is written:

$f(C_{11}\ ;\ \ldots;\ C_{1m_1})\ldots(C_{n1}\ ;\ \ldots;\ C_{nm_n})$

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

$f(C_{11}\ ;\ \ldots;\ C_{1m_1})\ldots(C_{n1}\ ;\ \ldots;\ C_{nm_n})?$

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

The alternative declares f as an operation. Each tuple $C_{i1}, \ldots, C_{im_i}$ specifies a curried argument for the profile; the result type is the type declared by the enclosing datatype declaration.

Subtypes

```
SUBTYPES ::= subtypes TYPE+
```

A subtypes alternative is written:

types $t_1, \ldots, t_n$

As with type declarations, the plural keyword may be written in the singular. For compatibility with CASL also the keyword sorts followed by type names may be used.

The types t_i, which must be already declared in the local environment, are declared to be embedded as subtypes of the type declared by the enclosing datatype declaration. ('types $t_1, \ldots, t_n$' and 'type t_1 | ... | type t_n' are equivalent.)

Components

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

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

$f_1, \ldots, f_n : t$

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

$f_1, \ldots, f_n :? \; t$

The remaining case is a component without a selector, simply written 't'.

The comma separated list of selectors is again an abbreviation for n tuple components of the same type and with the same partiality. In the first case, each selector operation is declared as total, and in the second case, as partial. It also determines sentences that define the value of each selector on the values given by the constructor of the enclosing alternative.

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

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

Free Datatypes

```
FREE-DATATYPE ::= free-datatype DATATYPE-DECL+
```

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

free types DD_1 ; . . . ; DD_n;

(The terminating '`;`' is optional.)

This construct is only well-formed when all the constructors declared by the datatype declarations are total.

The free datatype declarations declare the same types, constructors, and selectors as ordinary datatype declarations. Apart from the sentences that define the values of selectors, the free datatype declarations determine additional sentences requiring that the constructors are injective, that the ranges of constructors of the same sort are disjoint, that all the declared types are inductively generated by the constructors in the sense of Section 2.8, and that the value of applying a selector to a constructor for which it has not been declared is always undefined.

Besides these axioms, free datatype declarations additionally give rise to a ***case operator*** (cf. Section 2.11.7) which takes a tuple of functions, one on each alternative of the datatype, and returns a function on the whole datatype which behaves in the prescribed way on all alternatives.

When the alternatives of a free datatype declaration are all subtypes, and none of these subtypes have common subtypes, the declared type corresponds to the disjoint union of the subtypes. When the alternatives of a free datatype declaration are all constants, the declared type corresponds to an (unordered) enumeration type.

Note that the axioms generated by a free datatype are by default understood to belong to the logic defined in this chapter, later to be called the external logic as opposed to the internal logic of Chapter 3. This means that they constrain only the 'visible' elements of a datatype, i.e. so to speak its extension. If this is undesired, datatype declarations can be placed inside so-called internal logic blocks, cf. Chapter 4, so that the implicit axioms become formulae of the internal logic.

Moreover, even with the internal logic, there is no hope at all that free datatype declarations will be equivalent to free extensions, the difference being that a free extension would also require all newly arising function types to be freely term generated; this effect will not normally be desired. In fact, if a free datatype contains newly arising function spaces as arguments for its constructors, then its semantics will depend on the loose interpretation of these function spaces, so that the datatype itself will in this respect be a loose type.

2.8 Generated Items

```
GENERATED-ITEMS ::= generated SIG-ITEMS+
```

A generated items `GENERATED-ITEMS` is written:

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

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

generated types $DD_1; \ldots\ DD_n$;

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

A `GENERATED-ITEMS` is ill-formed if it does not declare any types. It determines the same elements of signature and sentences as $SI_1, \ldots, SI_n$, together with a corresponding ***induction axiom***. The declared operations (but *excluding* operations declared as *selectors* by datatype declarations) are called ***constructors***.

For `GENERATED-ITEMS` that have neither function spaces nor applications of type constructors as arguments of constructors, the induction axiom states that for any sequence of predicates over the sequence of declared types (called the ***induction predicates***), the inductive hypothesis implies the inductive conclusion. The inductive hypothesis expresses that the induction predicates are closed under the constructors, and the inductive conclusion expresses that they are everywhere-holding predicates. Note that the induction axiom is a higher-order reformulation of the corresponding sort generation constraint in CASL.[1]

For constructors with functional arguments, the notion of closedness of predicates under the constructor only makes sense if the induction predicates are extended to higher types. This is done as follows: the *extended induction predicate* on a function space type is satisfied by a function if the function takes values satisfying the relevant (extended or non-extended) induction predicate on its argument type to values satisfying the relevant induction predicate on its result type. The extended induction predicates on product types are taken componentwise, and those on types of the local environment

[1]However, due to the flexibility of interpretation of higher types in Henkin models, the higher-order reformulation is weaker than the sort generation constraint in CASL. In particular, not all non-standard models are excluded. However, proof-theoretically, this difference disappears — at least if the standard CASL proof system with the usual finitary induction rule is used. Only if stronger (e.g. infinitary) forms of induction are used, the difference becomes relevant. It also becomes relevant for monomorphicity: due to possible non-standard interpretations of higher types, the usual free datatypes are no longer monomorphic in HASCASL.

are taken to be constantly true. The extended induction predicates are only used in the inductive hypothesis, not in the conclusion.

Furthermore, there may be types of form $c\ t_1\ \ldots\ t_n$, where c is a non built-in type constructor from the local environment and at least one t_i is being declared in the `GENERATED-ITEMS`. Also for these types, extended induction predicates are introduced. However, they are not uniquely defined in terms of other induction predicates, but just stated to be closed under the operations with result type $c\ t_1\ \ldots\ t_n$ (which are necessarily newly arising instances of polymorphic operators), using the (extended) induction predicates for the t_i whenever appropriate. Again, the extended induction predicates are added to the inductive hypothesis, and not used in the conclusion.

Finally, types of form $c\, t_1\ \ldots\ t_n$, where c is a type constructor being declared in the `GENERATED-ITEMS`, are only considered to be well-formed if the t_i are type variables and $c\ t_1\ \ldots\ t_n$ is being declared in the `GENERATED-ITEMS` as well (and then the induction predicate is defined as above). That is, datatypes defined by polymorphic recursion are not allowed in HasCasl.

Note that the induction axiom does *not* imply that elements of a generated datatype containing functional constructors are reachable by the constructors and λ-abstraction. In particular, induction axioms do not preclude a standard interpretation of functional types (i.e. using the full function space, which cannot be term generated for infinite types).

2.9 Variables

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

Note that universal quantification over a variable that does not occur free in an axiom is semantically irrelevant, due to the assumption that all carriers are non-empty (cf. Section 1.3).

Type variables may also be declared globally, locally, or with explicit quantification. Globally or locally declared type variables are implicitly universally quantified in subsequent basic items of the enclosing basic specification. Type variables are not included in the declared signature.

2.9.1 Global Variable Declarations

```
GENERIC-VARS     ::= var-items GEN-VAR-DECL+
GEN-VAR-DECL     ::= VAR-DECL | TYPE-ARG
```

A list `GENERIC-VARS` of variable declarations is written:

vars VD_1; ... VD_n;

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

A `GEN-VAR-DECL` either declares a type variable with its kind or an ordinary variable (to be used within terms) with its type (of kind *Type*).

```
TYPEVAR  ::= SIMPLE-ID
VAR      ::= ID
```

A `TYPE-ARG` or `VAR-DECL` variable declaration is written as within operator definitions (without parentheses). Variables with the same kind or same type may be comma separated.

$v_1, \ldots, v_n : k$

It declares the type variables v_1, ..., v_n of kind k, if k is a legal kind, otherwise k is assumed to be a type and v_1, ..., v_n are declared as ordinary variables. All variables do *not* contribute to the declared signature.

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

Type variables and other variables have separate name spaces, thus the same identifier may be a type variable and an ordinary variable. Locally declared type variables will shadow type constructors with the same identifier, since there is no overloading of type names (while globally declared type variables must have names distinct from those of type constructors).

2.9.2 Local Variable Declarations

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

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

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

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

2.10 Formulae and Axioms

As in CASL, formulae (denoting truth values) are distinguished from terms (denoting data values). In higher-order logic, usually formulae and terms are identified, such that formulae are terms of a Boolean type. In HASCASL, the latter happens in the internal logic, see Chapters 3 and 4.

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

A list `AXIOM-ITEMS` of axioms is written:

$\bullet\ F_1 \ldots \bullet\ F_n$

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

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

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

2.10.1 Quantifications

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

A quantification with the `universal` quantifier is written:

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

A quantification with the `existential` quantifier is written:

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

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

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

The first case is universal quantification, holding when the body F holds for all values of the quantified variables; only an outermost universal quantification may declare type variables (see `GEN-VAR-DECL` in Section 2.9). The second case is existential quantification, holding when the body F holds for some values of the quantified variables; and the last case is unique existential quantification, abbreviating a formula that holds when the body F holds for unique values of the quantified variables. Type variable declarations are illegal within existential quantifications.

Provided type variable declarations precede other variable declarations, the formula $\forall VD_1;\ \ldots;\ VD_n\ \bullet\ F$ is equivalent to $\forall VD_1\ \bullet\ \ldots \forall VD_n\ \bullet\ F$. The scope of a variable declaration in a quantification is the component formula F, and an inner declaration for a variable with the same identifier as in an outer declaration overrides the outer declaration. There are never two variables in scope with the same identifier, however, possibly overloaded operations may have the same identifier as a variable and can be distinguished within terms by qualification. Note that the body of a quantification extends as far as possible.

2.10.2 Logical Connectives

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

Implication (which may be written in two different ways) has higher precedence than equivalence but weaker precedence than conjunction and disjunction. When the 'forward' version of implication is iterated, it is implicitly grouped to the right; the 'backward' version is grouped to the left. When these constructs are mixed, they have to be explicitly grouped. The equivalence has no associativity. The negation as a prefix operator binds stronger than the infix connectives.

Conjunction

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

A conjunction is written:

$$F_1 \wedge \ldots \wedge F_n$$

Disjunction

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

A disjunction is written:

$F_1 \vee \ldots \vee F_n$

Implication

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

An implication is written:

$F_1 \Rightarrow F_2$

An implication may also be written in reverse order:

F_2 *if* F_1

Equivalence

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

An equivalence is written:

$F_1 \Leftrightarrow F_2$

Negation

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

A negation is written:

$\neg F_1$

2.10.3 Atomic formulae

Atomic formula are the truth values, equations, definedness and membership tests and terms of type *Logical*, or (by procedural lifting, see Section 1.5) of type *Unit*. Note that due to intensionality, *Logical* generally may contain more than two truth values. In order to obtain a two-valued logic at the level of logical connectives and quantifiers, terms of type *Logical* (or, by procedural lifting, *Unit*) are implicitly coerced into a two-valued set of Booleans by comparing them with *true*. This means that every truth value of type *Logical* that is not *true* is collapsed to *false*. If this collapsing is not desired, one needs to use the internal logic, see Chapters 3 and 4.

An *atomic formula* is well-formed (with respect to the local environment and variable declarations) if it is well-typed and expands to an atomic formula for constructing sentences that is unique up to embedding-closure. Here, the latter means replacing an operation symbol together with appropriate injection(s) by the corresponding operation symbol for the composition (which exists by embedding-closure), or vice versa.

The notions of when an atomic formula is ***well-typed***, or when a term is *well-typed for a particular type*, and of the ***expansions*** of atomic formulae and terms, are indicated below for the various constructs.

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

Moreover, for non-coherent operation and predicate symbols, always the maximal type is chosen.

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

2.10.3.1 Truth

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

The atomic formulae `true-atom` and `false-atom` are written '*true*', '*false*' and expand to primitive sentences, such that the sentence for '*true*' always holds, and the sentence for '*false*' never holds.

2.10.3.2 Definedness

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

A definedness formula is written:

def T

It expands to a definedness assertion on the fully-qualified expansion of the term. The symbols *def* and $\neg$ are prefixes and therefore '$\neg$ *def* T' may be written without parentheses for a primitive T, although application by juxtaposition associates to the left.

An alternative notation is '$\neg \_\_(def \_\_ \ T)$'. Placeholders *must* be used if $\neg$ or *def* are not applied to an argument. (The same applies to all mixfix connectives including '$\_\_$ *when* $\_\_$ *else* $\_\_$' and $\_\_ \in s$.)

2.10.3.3 Equations

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

An existential equation `EXISTL-EQUATION` is written:

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

A strong equation is written:

$$T_1 = T_2$$

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

An equation is well-typed if the types of both terms are unifiable. It then expands to the corresponding existential or strong equation on the fully-qualified expansions of the terms.

2.10.3.4 Membership

```
MEMBERSHIP ::= membership TERM TYPE
```

A membership formula is written:

$$T \in s$$

It is well-typed if the term T is well-typed for a supertype t of the specified subtype s. It expands to an application of the pre-declared predicate symbol for testing t values for membership in the embedding of s.

2.11 Terms

Terms are built of variables, operation and predicate names (possibly qualified), applications, tuples, conditional terms, λ-abstraction, let and case. A term may also be annotated with a type or downcast to a subtype.

```
TERM ::= QUAL-VAR | INST-QUAL-NAME
       | TERM-APPL | TUPLE-TERM
       | TYPED-TERM
       | CONDITIONAL
       | TOTAL-LAMBDA | PARTIAL-LAMBDA
       | LET-TERM | CASE-TERM
       | CAST
```

2.11.1 Qualified Names

Atomic terms are qualified names, either bound variables or operations declared in the local signature that need to be instantiated if they are polymorphic. The concrete syntax does not require to explicitly qualify and instantiate names; it is the task of the static analysis to recognize variables and to infer the instance of polymorphic operations. Note that type inference can infer only types containing explicitly declared type variables; no type variables are generated by the inference process.

Qualified Variables

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

A qualified variable `QUAL-VAR` is written:

$(var\ \ v : t)$

Instantiated Qualified Names

```
INST-QUAL-NAME ::= inst-qual-name OP-NAME TYPE* TYPESCHEME
```

An instantiated qualified operation is written:

$(op\ \ o[t_1, \ldots, t_n] : \forall a_1\ ;\ \ldots;\ a_n\ \bullet\ t)$

The types for instantiation must correspond in number and order to the bound type variables of the type scheme. Note that the operation name o may be a compound identifier containing identifiers in square brackets. Proper resolution of compound lists and instance annotation is left to the static analysis.

Without type variables the square brackets are omitted:

(op $o : t$)

Instead of the keyword op, also fun or pred may be used. For a qualified predicate, the given type is the type of the argument tuple.

2.11.2 Applications

Application terms consist of a function term applied to a single argument term. Classical first-order application is a special case if the function term is a name and the argument is a tuple.

```
 TERM-APPL ::= term-appl TERM TERM
TUPLE-TERM ::= tuple-term TERM*
```

An application is given by mere juxtaposition and treated like an invisible and left-associative infix identifier '__ __' with highest precedence.

A tuple is written:

$(T_1, \ldots, T_n)$

The binding strength between the function name and the tuple argument is weaker in HasCasl than in Casl. In Casl, such an application has highest priority, but in HasCasl its priority is lower than that of ordinary prefix application.

As in Casl, the application of a mixfix identifier to its first tuple argument may be written in mixfix notation.

The empty tuple is a term of type *Unit*, and a singleton tuple is simply a term put in parentheses.

2.11.3 Typed Terms

```
TYPED-TERM ::= typed-term TERM TYPE
```

A typed term is written:

$T : t$

It is well-typed for some type if the component term T is well-typed for the specified type t. It then expands to those of the fully-qualified expansions of the component term that have the specified type.

2.11.4 Conditional Terms

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

A conditional term is written:

T_1 *when* F *else* T_2

The types of T_1 and T_2 must be unifiable. The enclosing *atomic* formula '$A[T_1$ *when* F *else* $T_2]$' expands to '$(A[T_1]$ *if* $F) \wedge (A[T_2]$ *if* $\neg F)$'.

2.11.5 Lambda Terms

```
PARTIAL-LAMBDA ::= partial-lambda PATTERN* TERM
TOTAL-LAMBDA   ::= total-lambda   PATTERN* TERM
```

Lambda terms are written:

$\lambda\ p_1 \ldots p_n\ \bullet\ T$

or:

$\lambda\ p_1 \ldots p_n\ \bullet!\ T$

The latter denotes a total function. Each constructor or tuple pattern p_i corresponds to a curried argument. An abstraction of a single unused variable of type *Unit* may also be written '$\lambda\ \bullet\ T$'.

When-else, logical connectives and quantifiers may not be used within λ-terms, unless the internal logic (see Chapter 3) is used; when-else remains excluded even within the internal logic.

2.11.6 Let-Terms

```
  LET-TERM ::= let-term PATTERN-EQ+ TERM
PATTERN-EQ ::= pattern-eq PATTERN TERM
```

Let-terms are written:

let PE_1 ; $\ldots$; PE_n *in* T

or equivalently:

T *where* PE_1 ; …; PE_n

The pattern equations PE_i bind new variables and local polymorphic operations that may be used in the term T. A pattern equation is written $p_i = T_i$. The pattern p_i is either a constructor or tuple pattern merely binding variables or the left hand side of a local polymorphic operation definition, i.e. a pattern application.

2.11.7 Case Terms

```
CASE-TERM ::= case-term TERM CASE+
     CASE ::= case PATTERN TERM
```

Case terms are written:

case T *of* C_1 | … | C_n

Each case C_i is written $p_i \rightarrow T_i$. The type of all T_i must be unifiable and will be the overall type of the case term. The type of T must be unifiable with the types of the patterns p_i. All the patterns are constructor or tuple patterns or (typed) variables. The concrete syntax allows a placeholder '__' as wild card pattern that corresponds to an unused variable. Variables in p_i may occur in T_i (but not in another T_j).

2.11.8 Casts

```
CAST ::= cast TERM TYPE
```

A cast term is written:

T *as* s

It is well-typed if the term T is well-typed for a supertype t of s. It expands to an application of the pre-declared operation symbol for projecting t to s.

Term formation is also extended by letting a well-typed term of a subtype s be regarded as a well-typed term of a supertype t as well, which provides implicit embedding. It expands to the explicit application of the pre-declared operation symbol for embedding s into t. (There are no implicit projections.) Also a typed term $T : t$ expands to an explicit application of an embedding, provided that the apparent type s of the component term T is a subtype of the specified type t.

2.11.9 Patterns

Patterns are special expressions that are used in left hand sides of equations in let terms and clauses in case terms. They introduce variable bindings. All variables of one pattern must be distinct. Like terms, patterns are subject to mixfix resolution. The syntax of pattern applications, tuple patterns and typed patterns is identical to that of the corresponding terms, and hence it is omitted here. Only as-patterns do not have a counterpart as terms.

```
PATTERN          ::= QUAL-VAR | INST-QUAL-NAME | PATTERN-APPL
                   | TUPLE-PATTERN | TYPED-PATTERN | AS-PATTERN
PATTERN-APPL     ::= pattern-appl PATTERN PATTERN
TUPLE-PATTERN    ::= tuple-pattern PATTERN*
TYPED-PATTERN    ::= typed-pattern PATTERN TYPE
```

In the concrete syntax, variables in patterns appearing *within a program block* (Chapter 6) need not be given types; types are then inferred by the static analysis. Outside program blocks, only explicitly typed variables are allowed.

As-Patterns

```
AS-PATTERN ::= as-pattern VAR PATTERN
```

An as-pattern is written $v@p$. It introduces a further variable v that names the value of the pattern term p. The type of the variable is that of the pattern.

2.12 Identifiers

Identifiers in HasCasl are those of Casl without the additional keywords (cf. Appendix C.3). In contrast to Casl, variables and types may be mixfix identifiers. Only type variables are restricted to be simple words. Classes also only have simple names but may be compound identifiers (like sorts in Casl).

'Invisible' identifiers, consisting entirely of two or more place-holders (separated by spaces), are no longer allowed.

An identifier `ID` may be used simultaneously to identify different kinds of entities (classes, types, and functions) in the same local environment.

Chapter 3

The Internal Logic — Concepts

The basic logic of HasCasl as laid out in Chapter 1 does not admit the use of equality or logical operators within λ-terms. (Recall that conditional terms are not allowed to be λ-abstracted. One can, however, emulate conjunction, the constant true proposition, and universal quantification, the latter via the elementhood predicate for total function types as subtypes of partial function types). However, equality can be sneaked back in by means of an ***internal equality***. Let *Pred* a abbreviate the type $a \rightarrow ? \, \mathit{Unit}$, and call the inhabitants of (*Pred* a) ***predicates***. A predicate

$$eq : \forall a \bullet \mathit{Pred}\ (a \times a)$$

in a partial λ-theory (with products) is called an internal equality (see also [Mog86]) if $eq(x, y)$ is equivalent to $x \overset{e}{=} y$ in the deduction system of Figure 1.2 (due to intensionality, this is a stronger property than equivalence of the two formulae for each pair (x, y) of elements of a in a model).

In fact, internal equality can be specified in HasCasl. The introduction of internal equality is highly non-conservative, since it makes the logic available within λ-abstracted predicates substantially richer: one can define all quantifiers and logical operators of intuitionistic higher order logic similarly as in [LS86]. The specification of internal equality and the new connectives is given in Figure 3.1. The specification uses the type of truth values *Logical* = *Pred Unit.* Moreover, it liberally applies the support for non-strict functions to pass back and forth between predicate applications, i.e. partial terms of type *Unit*, and terms of type *Logical.*

In order to improve readability, the equality symbol $\overset{e}{=}$ can, after all, be used within λ-terms, but is, then, implicitly replaced by eq. It may come as a

```
spec InternalLogic =
var a : Type
funs tt      : Logical = λ x : Unit • ()
     all     : Pred(Pred a) = λ p : Pred a • p ∈ (a → Unit)
     __ & __: Pred(Logical × Logical) = λ x, y : Logical • def(x(), y())
then
fun eq : Pred(a × a)
 • all(λ x : a • eq(x, x))
 • λ x, y : a • fst(x, eq(x, y)) = λ x, y : a • fst(y, eq(x, y))
then %def
funs __impl__, __or__ : Pred(Logical × Logical)
     ff :               Logical
     neg :              Pred Logical
     ex :               Pred(Pred a)
 • __impl__ = λ x, y : Logical • eq[Logical](x, x & y)
 • __or__ = λ x, y : Logical • all(λ r : Logical •
                         ((x impl r) & (y impl r)) impl r)
 • ff = λ y : Unit • all(λ x : Logical • x)
 • neg = λ x : Logical • x impl ff
 • ex[a] = λ p : Pred a • all(λ r : Logical •
                         all(λ x : a • p(x) impl r)) impl r
then %implies
var a, b : Type
 • all(λ f, g : a →? b • all(λ x : a • eq[?b](f(x), g(x))) impl eq(f, g))
```

Figure 3.1: Specification of the internal logic

surprise that the last formula shown in Figure 3.1 as a consequence of the definitions expresses a form of extensionality; however, it is well-known that all categorical models are 'internally extensional' [MS89].

The internal logic is intuitionistic: there may be more than two truth values, and *neg*(*neg* A) is in general different from A. The obvious deduction rules can be proved as lemmas; e.g., it is not hard to show that the rule

$$\frac{\phi \textit{ impl } \psi; \qquad \phi}{\psi}$$

is derivable from the rules in Figure 1.2 and the definitions in Figure 3.1. The *external* logic, i.e. the logic introduced in Sections 1.5 and 1.6, remains classical: as soon as a predicate appears as an atomic formula, all internal truth values except *tt* are collapsed into the external *false*.

The internal logic is used in specifications by implicitly importing its specification; this import is invoked by suitable built-in syntactic mechanisms.

Chapter 4

The Internal Logic — Constructs

This chapter treats a single construct used to invoke the internal logic described in the previous chapter.

```
BASIC-ITEMS      ::= ... | INTERNAL-ITEMS
INTERNAL-ITEMS   ::= internal BASIC-ITEMS+
```

An ***internal logic block*** is written

internal **{**$BI_1 \ldots BI_n$ **}**;

The enclosed specification has the same semantics as before, except that all formulae it contains, *including implicit ones* (such as the implicit axioms arising from subtype definitions or datatype declarations) are converted into formulae of the internal logic, the specification of which is automatically imported. In particular, enclosing universal quantifications $\forall x \bullet \phi$ implicit in the use of global or local variables are converted to internal universal quantifications $all(\lambda x \bullet \phi)$. Existential equality is converted to internal equality, while strong equality is coded via existential equality, definedness assertions, and implication.

Within the scope of an internal logic block, formulae can be used arbitrarily in terms; in particular, formulae can be λ-abstracted. Of course, such formulae within terms are also implicitly converted into internal formulae, which are really terms anyway.

Chapter 5

Recursive Functions — Concepts

In order to give a unique meaning to general recursive definitions and to ensure at the same time that the latter are actually definitions in the sense that they constitute definitional extensions, HasCasl offers the option of imposing a cpo structure on the relevant types. Just as in the case of the internal logic, the cpo structure is introduced by means of suitable specifications, building on the specification of internal logic (cf. Chapter 3) and making heavy use of the class mechanism. The overall concept is closely related to that of HOLCF [Reg95].

The specification of the cpo structure and the fixed point operator is given in Figure 5.1. Nat is a specification of the natural numbers with a sort Nat and operations $0 : Nat$ and $suc : Nat \to Nat$, including the usual induction axiom and a primitive recursion operator (which does not, of course, make use of the cpo structure). We introduce type classes *Cpo* and *Cppo* of cpos and cpos with bottom, respectively, with generic instantiations that extend the ordering to products and partial and total continuous function types; the subclass *FlatCpo* restricts the order to be equality. The continuous function types are subtypes of the built-in function types; partial continuous functions are required to have Scott open domains. Elements of function types are compared pointwise, and elements of product types are compared componentwise. For continuous functions, we can introduce a least fixed point operator Y as a definitional extension (since Y is expressible as the supremum of a chain which can be defined by primitive recursion); a recursive definition such as $f\ x = \alpha$ is then interpretable as $f = Y(\lambda\, f \bullet !\, \lambda\, x \bullet \alpha)$ (more precisely, with $\lambda\, x \bullet \alpha$ and $\lambda\, f \bullet !\, \lambda\, x \bullet \alpha$ downcast to the appropriate continuous function types).

A further instance of the class *Cpo* are free datatypes with constructor

spec RECURSION $=$ **internal** {NAT **then**
class *Cpo* { **var** $a : Cpo$
fun $\_\_ \leq \_\_ : pred(a \times a)$
var $x, y, z : a$
• $x \leq x$
• $(x \leq y \ \wedge y \leq z) \Rightarrow x \leq z$
• $(x \leq y \ \wedge y \leq x) \ \Rightarrow x = y$
type $Chain\ a = \{s : Nat \to a \bullet \forall n : Nat \bullet s(n) \leq s(suc(n))\}$
fun $sup : Chain\ a \to a$
var $x : a;\ c : Chain\ a \bullet\ sup(c) \leq x \Leftrightarrow \forall n : Nat \bullet c(n) \leq x$
}
class $Cppo < Cpo$ { **var** $a : Cppo$
fun $bottom : a$
var $x : a \bullet bottom \leq x$
}
class instance $FlatCpo < Cpo$
  {**var** $a : FlatCpo \bullet\ \_\_ \leq \_\_[a] = eq[a]$}
vars $a, b : Cpo;\ c : Cppo;\ x, y : a;\ z, w : b$
type instance $\_\_ \times \_\_ : Cpo \to Cpo \to Cpo$
• $(x, y) \leq (z, w) \Leftrightarrow x \leq z \wedge y \leq w$
type instance $\_\_ \times \_\_ : Cppo \to Cppo \to Cppo$
type instance $Unit : Cppo, FlatCpo$
• $() \leq ()$
type $a \overset{\text{cont}}{\longrightarrow}?\ b = \{f : a \to ?\ b \bullet$
    $\forall x, y : a \bullet (def\ \ f(x) \wedge x \leq y \Rightarrow def\ \ f(y)) \wedge$
    $\forall c : Chain\ a \bullet\ def\ \ f(sup(c)) \Rightarrow \exists m : Nat \bullet$
      $def\ \ f(c(m)) \wedge$
      $sup((\lambda\, n : Nat \bullet ! f(c(n + m)))\ \ as\ \ Chain\ a) \overset{e}{=} f(sup(c))\}$
type $a \overset{\text{cont}}{\longrightarrow} b = \{f : a \overset{\text{cont}}{\longrightarrow}?\ b \bullet f \in a \to b\}$
type instance $\_\_ \overset{\text{cont}}{\longrightarrow}?\ \_\_ : Cpo \to Cpo \to Cppo$
var $f, g : a \overset{\text{cont}}{\longrightarrow}?\ b \bullet\ f \leq g \Leftrightarrow \forall x : a \bullet\ def\ \ f(x) \Rightarrow f(x) \leq g(x)$
type instance $\_\_ \overset{\text{cont}}{\longrightarrow} \_\_ : Cpo \to Cpo \to Cpo$
type instance $\_\_ \overset{\text{cont}}{\longrightarrow} \_\_ : Cpo \to Cppo \to Cppo$
then %def
fun $Y : (c \overset{\text{cont}}{\longrightarrow} c) \overset{\text{cont}}{\longrightarrow} c$
var $f : c \overset{\text{cont}}{\longrightarrow} c;\ x : c$
• $f(Y(f)) = Y(f)$
• $f(x) = x\ \ \Rightarrow\ Y(f) \leq x$
}

Figure 5.1: Specification of the cpo structure and the fixed point operator

arguments of class *Cpo*; such instances can be automatically generated: applications of different constructors are incomparable, while applications of the same constructor are compared argument-wise. There is no circularity here: the definition of the ordering is recursive, but does not use the fixed point operator. Rather, it imposes a particular equation on the ordering, and this equation determines the ordering uniquely thanks to the induction axioms for generated datatypes (Section 2.7). It is easy to see that the case operation, restricted to continuous arguments, is continuous w.r.t. this ordering, and hence can be used in definitions of recursive functions.

Actual recursive definitions will be expressions which involve Y and a partial downcast to the total continuous function type. As long as operators are given the right types, such expressions actually denote functions: call a term α in context $(\bar{x} : \bar{s})$ that has a type of class *Cpo continuous* if $\lambda\, \bar{x} : \bar{s} \bullet \alpha$ is continuous. Moreover, call a type a *cpo-type* if it is built from loose types and type variables of class *Cpo* by means of the instance declarations for type constructors given in Figure 5.1 (in particular, cpo-types are of class *Cpo*). Then we have

Proposition 8 *If, for* $\Gamma \rhd \alpha : t$*, all operator constants (besides application) that occur in* α*, as well as the variables in* Γ*, have cpo-types, and* t *is a cpo-type, then* α *is continuous.*

As a consequence, functions of the form, say, $\lambda\, f \bullet !\, \lambda\, x \bullet \alpha$ with α as in the proposition are continuous total functions and hence possess a least fixed point, so that the recursive definition $f\ x = \alpha$ is a definitional extension. Note that the fixed point operator itself is of a cpo-type, provided that its parameter a is instantiated with a cpo-type.

Chapter 6

Recursive Functions — Constructs

This chapter treats the syntax of general recursive function definitions, i.e. in a sense functional programs.

```
BASIC-ITEMS       ::= ... | PROG-ITEMS
PROG-ITEMS        ::= prog-items PATTERN-EQ+
```

A ***program block*** is written as a sequence of pattern equations in the form

program {$PE_1 \dots PE_n$ };

The enclosed pattern equations are implicitly replaced by recursive function definitions using the least fixed point operator as indicated in Chapter 5. It is statically checked that all involved types are cpo-types; the specification is ill-formed if this check fails. All occurring λ-abstractions, implicit or explicit, are equipped with a downcast to the appropriate continuous function type (so that the user does not have to write these casts explicitly). By consequence, recursively defined functions are undefined if one of the functions involved in their definition fails to be continuous (a sufficient criterion for continuity can be statically checked; cf. Chapter 5). Recursive functions on free datatypes can be defined by giving a recursive equation for each constructor. This is coded by means of the case operator; an attempt to use this mechanism for non-free datatypes (which do not have case operators) makes the specification ill-formed. On missing constructor patterns, functions are implicitly undefined; in this case, a warning ('non-exhaustive match') is produced. Of course, the case operator may also be used explicity if desired.

Derived type classes The syntax for derived type classes for free datatypes introduced in Section 2.7:

$$t\ a_1 \ldots a_m ::= A_1 \mid \ldots \mid A_n\ \textbf{deriving}\ c_1, \ldots, c_l$$

may be used for the type classes *Ord*, *Cpo* and *Cppo*. An axiomatization of appropriate **type instances** is generated, corresponding to the usual cpos on datatypes.

Bibliography

[BTM85] V. Breazu-Tannen and A. R. Meyer. Lambda calculus with constrained types. In *Logic of Programs*, volume 193 of *LNCS*, pages 23–40. Springer, 1985.

[Bur93] P. Burmeister. Partial algebras — an introductory survey. In *Algebras and Orders*, NATO ASI Series C, pages 1–70. Kluwer, 1993.

[CO89] P.-L. Curien and A. Obtułowicz. Partiality, Cartesian closedness and toposes. *Inform. and Comput.*, 80:50–95, 1989.

[CoF] CoFI Language Design Group. CASL summary. Part I of [Mos04]. Edited by B. Krieg-Brückner and P. D. Mosses.

[GB92] J. A. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39:95–146, 1992.

[HM03] K. Hoffmann and T. Mossakowski. Algebraic higher order nets: Graphs and petri nets as tokens. In *Recent Trends in Algebraic Development Techniques, 16th International Workshop (WADT 02)*, volume 2755 of *LNCS*, pages 253–267. Springer, 2003.

[HPF99] P. Hudak, J. Peterson, and J. H. Fasel. A gentle introduction into Haskell 98. Available at `http://www.haskell.org`, 1999.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

[MMS03] C. Maeder, T. Mossakowski, and L. Schröder. From requirements to functional programs — the HASCASL development paradigm. Technical report, Universität Bremen, 2003. Available as `http://www.tzi.de/~till/papers/paradigm.ps`.

[Mog86] E. Moggi. Categories of partial morphisms and the λ_p-calculus. In *Category Theory and Computer Programming*, volume 240 of *LNCS*, pages 242–251. Springer, 1986.

[Mog88] E. Moggi. *The Partial Lambda Calculus.* PhD thesis, University of Edinburgh, 1988.

[Mos04] P. D. Mosses, editor. CASL *Reference Manual*, volume 2960 of *LNCS (IFIP Series).* Springer, 2004.

[MS89] J. C. Mitchell and P. J. Scott. Typed lambda models and cartesian closed categories. In *Categories in Computer Science and Logic*, volume 92 of *Contemp. Math.*, pages 301–316. Amer. Math. Soc., 1989.

[MSRR] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-coalgebraic specification in CoCASL. *J. Logic and Algebraic Programming.* To appear. Earlier version in *Recent Developments in Algebraic Development Techniques, 16th International Workshop (WADT 02)*, volume 2755 of *LNCS*, pp. 376–392. Springer, 2003.

[Reg95] F. Regensburger. HOLCF: Higher order logic of computable functions. In *Theorem Proving in Higher Order Logics*, volume 971 of *LNCS*, pages 293–307, 1995.

[Ros86] G. Rosolini. *Continuity and effectiveness in topoi.* PhD thesis, University of Oxford, 1986.

[Sch] L. Schröder. The HASCASL prologue: categorical syntax and semantics of the partial λ-calculus. Available as `http://www.informatik.uni-bremen.de/~lschrode/hascasl/plam.ps`

[Sch03] L. Schröder. Henkin models of the partial λ-calculus. In *Computer Science Logic*, volume 2803 of *LNCS*, pages 498–512. Springer, 2003.

[SM] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HASCASL. *J. Logic Comput.* To appear. Earlier version in *Recent Developments in Algebraic Development Techniques, 16th International Workshop (WADT 02)*, volume 2755 of *LNCS*, pages 425-442. Springer, 2003.

[SM02] L. Schröder and T. Mossakowski. HASCASL: Towards integrated specification and development of functional programs. In *Algebraic Methodology and Software Technology*, volume 2422 of *LNCS*, pages 99–116. Springer, 2002.

[SM03] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HASCASL. In *Fundamental Aspects of Software Engineering*, volume 2621 of *LNCS*, pages 261–277, 2003.

[SM04] L. Schröder and T. Mossakowski. Type class polymorphism in an institutional framework, 2004. Available as `http://www.informatik.uni-bremen.de/~lschrode/papers/genpoly.ps`

[SMT$^+$01] L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman, and B. Klin. Semantics of architectural specifications in Casl. In *Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS*, pages 253–268. Springer, 2001.

[SS93] A. Salibra and G. Scollo. A soft stairway to institutions. In *Recent Trends in Data Type Specification, 8th International Workshop (WADT 91)*, volume 655 of *LNCS*, pages 310–329. Springer, 1993.

Appendices

Appendix A

Abstract Syntax

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

The abstract syntax has the following objectives:

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

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

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

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

or in terms of its constructor and components:

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

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

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

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

The following nonterminal symbols correspond to the CASL syntax, and are left unspecified here: ID, SIMPLE-ID, PLACE and LITERAL.

A.1 Basic Specifications

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

BASIC-ITEMS      ::= CLASS-ITEMS | SIG-ITEMS
                   | GENERATED-ITEMS | FREE-DATATYPES
                   | GENERIC-VARS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS

CLASS-ITEMS          ::= SIMPLE-CLASS-ITEMS | INSTANCE-CLASS-ITEMS
SIMPLE-CLASS-ITEMS   ::= class-items CLASS-ITEM+
INSTANCE-CLASS-ITEMS ::= instance-class-items CLASS-ITEM+

CLASS-ITEM       ::= class-item CLASS-DECL BASIC-ITEMS*

CLASS-DECL       ::= SIMPLE-CLASS-DECL | SUBCLASS-DECL
SIMPLE-CLASS-DECL::= class-decl CLASS-NAME+
SUBCLASS-DECL    ::= subclass-decl CLASS-NAME+ KIND

KIND             ::= TYPE-UNIVERSE | CLASS-KIND | INTERSECTION-KIND
                   | DOWNSET-KIND | FUN-KIND
TYPE-UNIVERSE    ::= type-universe
CLASS-KIND       ::= class-name CLASS-NAME
INTERSECTION-KIND::= intersection KIND+
DOWNSET-KIND     ::= downset TYPE
FUN-KIND         ::= fun-kind EXT-KIND KIND
EXT-KIND         ::= ext-kind KIND VARIANCE
VARIANCE         ::= covariant | contravariant | invariant

SIG-ITEMS        ::= TYPE-ITEMS | OP-ITEMS | FUN-ITEMS | PRED-ITEMS

TYPE-ITEMS          ::= SIMPLE-TYPE-ITEMS | INSTANCE-TYPE-ITEMS
SIMPLE-TYPE-ITEMS   ::= type-items TYPE-ITEM+
INSTANCE-TYPE-ITEMS ::= instance-type-items TYPE-ITEM+

TYPE-ITEM        ::= TYPE-DECL | SYNONYM-TYPE | DATATYPE-DECL
                   | SUBTYPE-DECL | ISO-DECL | SUBTYPE-DEFN
TYPE-DECL        ::= type-decl TYPE-NAME+ KIND
SYNONYM-TYPE     ::= synonym-type TYPE-NAME TYPE-ARG* TYPE
SUBTYPE-DECL     ::= subtype-decl TYPE-NAME+ TYPE
```

```
ISO-DECL          ::= iso-decl TYPENAME+
SUBTYPE-DEFN      ::= subtype-defn TYPE-NAME TYPE-ARG* VAR TYPE FORMULA

TYPE-ARG          ::= type-arg TYPEVAR EXT-KIND

OP-ITEMS          ::= op-items OP-ITEM+
FUN-ITEMS         ::= fun-items OP-ITEM+
OP-ITEM           ::= OP-DECL | OP-DEFN
OP-DECL           ::= op-decl OP-NAME+ TYPESCHEME OP-ATTR*

TYPESCHEME        ::= typescheme TYPE-ARG* TYPE

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 TYPE-ARG* OP-HEAD TERM
OP-HEAD           ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD     ::= total-op-head   TUPLE-ARG* TYPE
PARTIAL-OP-HEAD   ::= partial-op-head TUPLE-ARG* TYPE
TUPLE-ARG         ::= tuple-arg VAR-DECL*
VAR-DECL          ::= var-decl VAR TYPE

PRED-ITEMS        ::= pred-items PRED-ITEM+
PRED-ITEM         ::= PRED-DECL | PRED-DEFN
PRED-DECL         ::= pred-decl OP-NAME+ TYPESCHEME
PRED-DEFN         ::= pred-defn OP-NAME TYPE-ARG* TUPLE-ARG FORMULA

GENERATED-ITEMS   ::= generated SIG-ITEMS+
FREE-DATATYPE     ::= free-datatype DATATYPE-DECL+
GENERIC-VARS      ::= var-items GEN-VAR-DECL+
LOCAL-VAR-AXIOMS  ::= local-var-axioms GEN-VAR-DECL+ FORMULA+
AXIOM-ITEMS       ::= axiom-items FORMULA+

GEN-VAR-DECL      ::= VAR-DECL | TYPE-ARG

DATATYPE-DECL     ::= datatype-decl TYPE-NAME TYPE-ARG* ALTERNATIVE+
                                                        CLASS-NAME*

ALTERNATIVE       ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT | SUBTYPES
TOTAL-CONSTRUCT   ::= total-construct   OP-NAME TUPLE-COMPONENT*
PARTIAL-CONSTRUCT::= partial-construct OP-NAME TUPLE-COMPONENT+
TUPLE-COMPONENT   ::= tuple-component COMPONENTS+
COMPONENTS        ::= TOTAL-SELECT | PARTIAL-SELECT | TYPE
TOTAL-SELECT      ::= total-select   OP-NAME+ TYPE
PARTIAL-SELECT    ::= partial-select OP-NAME+ TYPE
SUBTYPES          ::= subtypes TYPE+

TYPE              ::= TYPE-NAME | TYPE-APPL
                    | PRODUCT-TYPE | LAZY-TYPE | KINDED-TYPE | FUN-TYPE
TYPE-APPL         ::= type-appl TYPE TYPE
PRODUCT-TYPE      ::= product-type TYPE+
LAZY-TYPE         ::= lazy-type TYPE
KINDED-TYPE       ::= kinded-type TYPE KIND
```

```
FUN-TYPE         ::= fun-type TYPE ARROW TYPE
ARROW            ::= partial-fun | total-fun
                   | cont-partial-fun | cont-total-fun

FORMULA          ::= QUANTIFICATION | CONJUNCTION | DISJUNCTION
                   | IMPLICATION | EQUIVALENCE
                   | NEGATION | ATOM
QUANTIFICATION   ::= quantification QUANTIFIER GEN-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 | DEFINEDNESS
                   | EXISTL-EQUATION | STRONG-EQUATION
                   | MEMBERSHIP | PREDICATION
TRUTH            ::= true-atom | false-atom
DEFINEDNESS      ::= definedness TERM
EXISTL-EQUATION  ::= existl-equation TERM TERM
STRONG-EQUATION  ::= strong-equation TERM TERM
MEMBERSHIP       ::= membership TERM TYPE
PREDICATION      ::= predication TERM

TERM             ::= QUAL-VAR | INST-QUAL-NAME
                   | TERM-APPL | TUPLE-TERM
                   | TYPED-TERM
                   | CONDITIONAL
                   | TOTAL-LAMBDA | PARTIAL-LAMBDA
                   | LET-TERM | CASE-TERM
                   | CAST
QUAL-VAR         ::= qual-var VAR TYPE
INST-QUAL-NAME   ::= inst-qual-name OP-NAME TYPE* TYPESCHEME
TERM-APPL        ::= term-appl TERM TERM
TUPLE-TERM       ::= tuple-term TERM*
TYPED-TERM       ::= typed-term TERM TYPE
CONDITIONAL      ::= conditional TERM TERM TERM
PARTIAL-LAMBDA   ::= partial-lambda PATTERN* TERM
TOTAL-LAMBDA     ::= total-lambda   PATTERN* TERM
LET-TERM         ::= let-term PATTERN-EQ+ TERM
CASE-TERM        ::= case-term TERM CASE+
CAST             ::= cast TERM TYPE

PATTERN          ::= QUAL-VAR | INST-QUAL-NAME | PATTERN-APPL
                   | TUPLE-PATTERN | TYPED-PATTERN | AS-PATTERN
PATTERN-APPL     ::= pattern-appl PATTERN PATTERN
TUPLE-PATTERN    ::= tuple-pattern PATTERN*
TYPED-PATTERN    ::= typed-pattern PATTERN TYPE
AS-PATTERN       ::= as-pattern VAR PATTERN

PATTERN-EQ       ::= pattern-eq PATTERN TERM
CASE             ::= case PATTERN TERM
```

```
CLASS-NAME       ::= ID
TYPE-NAME        ::= ID
OP-NAME          ::= ID
VAR              ::= ID
TYPEVAR          ::= SIMPLE-ID
```

A.2 Basic Specifications with Internal Logic

```
BASIC-ITEMS      ::= ... | INTERNAL-ITEMS

INTERNAL-ITEMS   ::= internal BASIC-ITEMS+
```

A.3 Basic Specifications with Recursive Programs

```
BASIC-ITEMS      ::= ... | PROG-ITEMS

PROG-ITEMS       ::= prog-items PATTERN-EQ+
```

Appendix B

Abbreviated Abstract Syntax

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

BASIC-ITEMS      ::= class-items CLASS-ITEM+
                   | instance-class-items CLASS-ITEM+
                   | SIG-ITEMS
                   | generated SIG-ITEMS+
                   | internal BASIC-ITEMS+
                   | free-datatype DATATYPE-DECL+
                   | var-items GEN-VAR-DECL+
                   | local-var-axioms GEN-VAR-DECL+ FORMULA+
                   | axiom-items FORMULA+
                   | internal-items BASIC-ITEMS+
                   | prog-items PATTERN-EQ+

CLASS-ITEM       ::= class-item CLASS-DECL BASIC-ITEMS*

CLASS-DECL       ::= class-decl CLASS-NAME+
                   | subclass-decl CLASS-NAME+ KIND

KIND             ::= type-universe
                   | class-name CLASS-NAME
                   | intersection KIND+
                   | downset TYPE
                   | fun-kind EXT-KIND KIND

EXT-KIND         ::= ext-kind KIND VARIANCE

VARIANCE         ::= covariant | contravariant | invariant

SIG-ITEMS        ::= type-items TYPE-ITEM+
                   | instance-type-items TYPE-ITEM+
                   | op-items OP-ITEM+
                   | fun-items OP-ITEM+
                   | pred-items PRED-ITEM+

TYPE-ITEM        ::= type-decl TYPE-NAME+ KIND
```

```
                  | synonym-type TYPE-NAME TYPE-ARG* TYPE
                  | DATATYPE-DECL
                  | subtype-decl TYPE-NAME+ TYPE
                  | iso-decl TYPENAME+
                  | subtype-defn TYPE-NAME TYPE-ARG* VAR TYPE FORMULA

TYPE-ARG        ::= type-arg TYPEVAR EXT-KIND

OP-ITEM         ::= op-decl OP-NAME+ TYPESCHEME OP-ATTR*
                  | op-defn OP-NAME TYPE-ARG* OP-HEAD TERM

TYPESCHEME      ::= typescheme TYPE-ARG* TYPE

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

OP-HEAD         ::= total-op-head   TUPLE-ARG* TYPE
                  | partial-op-head TUPLE-ARG* TYPE

TUPLE-ARG       ::= tuple-arg VAR-DECL*
VAR-DECL        ::= var-decl VAR TYPE

PRED-ITEM       ::= pred-decl OP-NAME+ TYPESCHEME
                  | pred-defn OP-NAME TYPE-ARG* TUPLE-ARG FORMULA

GEN-VAR-DECL    ::= VAR-DECL | TYPE-ARG

DATATYPE-DECL   ::= datatype-decl TYPE-NAME TYPE-ARG* ALTERNATIVE+
                                                      CLASS-NAME*

ALTERNATIVE     ::= total-construct   OP-NAME TUPLE-COMPONENT*
                  | partial-construct OP-NAME TUPLE-COMPONENT+
                  | subtypes TYPE+
TUPLE-COMPONENT ::= tuple-component COMPONENTS+
COMPONENTS      ::= total-select   OP-NAME+ TYPE
                  | partial-select OP-NAME+ TYPE
                  | TYPE

TYPE            ::= TYPE-NAME
                  | type-appl TYPE TYPE
                  | product-type TYPE+
                  | lazy-type TYPE
                  | kinded-type TYPE KIND
                  | fun-type TYPE ARROW TYPE

ARROW           ::= partial-fun | total-fun
                  | cont-partial-fun | cont-total-fun

QUANTIFIER      ::= universal | existential | unique-existential
FORMULA         ::= quantification QUANTIFIER GEN-VAR-DECL+ FORMULA
                  | conjunction FORMULA+
                  | disjunction FORMULA+
                  | implication FORMULA FORMULA
                  | equivalence FORMULA FORMULA
                  | negation FORMULA
                  | true-atom | false-atom
                  | definedness TERM
                  | existl-equation TERM TERM
                  | strong-equation TERM TERM
                  | membership TERM TYPE
                  | predication TERM

TERM            ::= QUAL-VAR
                  | INST-QUAL-NAME
                  | term-appl TERM TERM
                  | tuple-term TERM*
                  | typed-term TERM TYPE
                  | conditional TERM FORMULA TERM
                  | partial-lambda PATTERN* TERM
                  | total-lambda   PATTERN* TERM
                  | let-term TERM PATTERN-EQ+
                  | case-term TERM CASE+
                  | cast TERM TYPE

INST-QUAL-NAME  ::= inst-qual-name OP-NAME TYPE* TYPESCHEME
QUAL-VAR        ::= qual-var VAR TYPE

PATTERN         ::= QUAL-VAR | INST-QUAL-NAME
                  | pattern-appl PATTERN PATTERN
                  | tuple-pattern PATTERN*
                  | typed-pattern PATTERN TYPE
                  | as-pattern VAR PATTERN

PATTERN-EQ      ::= pattern-eq PATTERN TERM
CASE            ::= case PATTERN TERM

CLASS-NAME      ::= ID
TYPE-NAME       ::= ID
OP-NAME         ::= ID
VAR             ::= ID
TYPEVAR         ::= SIMPLE-ID
```

Appendix C

Concrete Syntax

The relationship between the concrete syntax and the corresponding abstract syntax is rather straightforward—except that mapping the use of mixfix notation in a concrete `TERM` or `PATTERN` to an abstract `TERM` or `PATTERN` depends on the static analysis. The mixfix resolution does not depend on the types of operations, but the type of names determines their qualification. Here, the relationship is merely suggested by the use of the same nonterminal symbols in the concrete and abstract grammars plus the non-terminal `MIXFIX` in the concrete grammar. The non-terminal `MIXFIX` also covers list notations, compound list of identifiers and type instance lists for polymorphic operations.

Further examples of specifications illustrating the concrete syntax are given in [MMS03, HM03]. A parser for HasCasl is available via the http://www.tzi.de/cofi web page.

Section C.1 below provides a context-free grammar for the HasCasl input syntax. It has been derived from the 'abbreviated' abstract syntax grammar in Appendix B, except for the productions for mixfix terms and patterns as well as for the abbreviated declaration of equally typed variables (or equally kinded type variables) using commas. The context-free grammar is ambiguous; Section C.2 explains various precedence rules for disambiguation and minor differences to Casl.

The lexical syntax, comments and annotations, the literal syntax, and the display format is identical that of Casl.

C.1 Context-Free Syntax

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

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

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

C.1.1 Basic Specifications

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

BASIC-ITEMS     ::= class/classes CLASS-ITEM ;...; CLASS-ITEM ;/
                  | class instance/instances
                           CLASS-ITEM ;...; CLASS-ITEM ;/
                  | SIG-ITEMS
                  | free type/types
                           DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | generated type/types
                           DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | generated { SIG-ITEMS...SIG-ITEMS } ;/
                  | internal { BASIC-ITEMS...BASIC-ITEMS } ;/
                  | var/vars GEN-VAR-DECL ;...; GEN-VAR-DECL ;/
                  | forall GEN-VAR-DECL ;...; GEN-VAR-DECL
                             "." FORMULA "."..."." FORMULA ;/
                  | "." FORMULA "."..."." FORMULA ;/
                  | program/programs PATTERN-EQ ;...; PATTERN-EQ ;/

CLASS-ITEM      ::= CLASS-DECL
                  | CLASS-DECL { BASIC-ITEMS...BASIC-ITEMS }

CLASS-DECL      ::= CLASS-NAME ,..., CLASS-NAME
                  | CLASS-NAME ,..., CLASS-NAME : KIND
                  | CLASS-NAME ,..., CLASS-NAME < KIND

KIND            ::= Type
                  | CLASS-NAME
                  | ( KIND ,..., KIND )
                  | { VAR "." VAR < TYPE }
                  | EXT-KIND -> KIND

EXT-KIND        ::= KIND | KIND + | KIND -

SIG-ITEMS       ::= sort/sorts SORT-ITEM ;...; SORT-ITEM
                  | type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | type/types TYPE-ITEM ;...; TYPE-ITEM ;/
                  | type/types instance/instances
                                TYPE-ITEM ;...; TYPE-ITEM ;/
                  | op/ops OP-ITEM ;...; OP-ITEM ;/
                  | fun/funs OP-ITEM ;...; OP-ITEM ;/
                  | pred/preds PRED-ITEM ;...; PRED-ITEM ;/

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

TYPE-ITEM       ::= SORT-ITEM
                  | TYPE-PATTERN := SYNONYM-TYPE

SYNONYM-TYPE    ::= TYPE
```

```
                 | \ TYPE-ARGS "." TYPE

TYPE-PATTERN    ::= TYPE-NAME
                 | TYPE-NAME TYPE-ARGS

TYPE-ARGS       ::= TYPE-ARG...TYPE-ARG
TYPE-ARG        ::= EXT-TYPE-VAR
                 | EXT-TYPE-VAR : EXT-KIND
                 | EXT-TYPE-VAR < TYPE
                 | ( TYPE-ARG )
EXT-TYPE-VAR    ::= TYPE-VAR | TYPE-VAR + | TYPE-VAR -

DATATYPE-DECL   ::= TYPE-PATTERN "::=" ALTERNATIVES
                 | TYPE-PATTERN "::=" ALTERNATIVES deriving CLASSES

CLASSES         ::= CLASS-NAME ,..., CLASS-NAME

ALTERNATIVES    ::= ALTERNATIVE "|"..."|" ALTERNATIVE

ALTERNATIVE     ::= OP-NAME TUPLE-COMPONENT...TUPLE-COMPONENT
                 | OP-NAME TUPLE-COMPONENT...TUPLE-COMPONENT ?
                 | OP-NAME
                 | sort/sorts TYPE-NAME ,..., TYPE-NAME
                 | type/types TYPE ,..., TYPE

TUPLE-COMPONENT ::= ( COMPONENT ;...; COMPONENT )

COMPONENT       ::= OP-NAME ,..., OP-NAME : TYPE
                 | OP-NAME ,..., OP-NAME :? TYPE
                 | TYPE

OP-ITEM         ::= OP-NAME ,..., OP-NAME : TYPESCHEME
                 | OP-NAME ,..., OP-NAME : TYPESCHEME, OP-ATTRS
                 | OP-NAME : TYPESCHEME = TERM
                 | OP-NAME [ TYPE-VAR-DECLS ] OP-HEAD = TERM
                 | OP-NAME OP-HEAD = TERM

OP-HEAD         ::= TUPLE-ARGS : TYPE
                 | TUPLE-ARGS :? TYPE
                 | :? TYPE

TUPLE-ARGS      ::= TUPLE-ARG...TUPLE-ARG
TUPLE-ARG       ::= ( VAR-DECL ;...; VAR-DECL )
VAR_DECL        ::= VAR ,..., VAR : TYPE

OP-ATTRS        ::= OP-ATTR ,..., OP-ATTR
OP-ATTR         ::= BIN-ATTR | unit TERM
BIN-ATTR        ::= assoc | comm | idem

PRED-ITEM       ::= OP-NAME, ..., OP-NAME: TYPESCHEME
                 | OP-NAME [ TYPE-VAR-DECLS ] TUPLE-ARG <=> FORMULA
                 | OP-NAME TUPLE-ARG <=> FORMULA
                 | OP-NAME <=> FORMULA
```

```
TYPESCHEME      ::= TYPE
                  | forall TYPE-VAR-DECLS . TYPE

TYPE-VAR-DECLS  ::= TYPE-VARS ;...; TYPE-VARS

TYPE-VARS       ::= EXT-TYPE-VAR ,...,  EXT-TYPE-VAR
                  | EXT-TYPE-VAR ,...,  EXT-TYPE-VAR : EXT-KIND
                  | EXT-TYPE-VAR ,...,  EXT-TYPE-VAR < TYPE

GEN-VAR-DECL    ::= TYPE-VARS
                  | VAR-DECL

TYPE            ::= TYPE ARROW TYPE
                  | TYPE *...* TYPE
                  | ( TYPE )
                  | Pred Type
                  | ? TYPE
                  | Unit
                  | Logical
                  | TYPE : KIND
                  | TYPE TYPE
ARROW           ::= ->? | -> | -->? | -->

FORMULA         ::= QUANTIFIER GEN-VAR-DECL ;...; GEN-VAR-DECL
                               "." FORMULA
                  | FORMULA /\.../\ FORMULA
                  | FORMULA \/...\/ FORMULA
                  | FORMULA => FORMULA
                  | FORMULA if FORMULA
                  | FORMULA <=> FORMULA
                  | not FORMULA
                  | true | false
                  | def TERM
                  | TERM in TYPE
                  | TERM = TERM
                  | TERM =e= TERM
                  | TERM

TERM            ::= QUAL-VAR
                  | INST-QUAL-NAME
                  | TERM TERM
                  | (TERM ,..., TERM) | ( )
                  | TERM : TYPE
                  | TERM when FORMULA else TERM
                  | \ LAMBDA-DOT TERM
                  | \ PATTERN...PATTERN LAMBDA-DOT TERM
                  | let PATTERN-EQ ;...; PATTERN-EQ in TERM
                  | TERM where PATTERN-EQ ;...; PATTERN-EQ ;/
                  | case TERM of CASE "|"..."|" CASE
                  | TERM as TYPE
                  | LITERAL
                  | MIXFIX

LAMBDA-DOT      ::= "." | .!
```

```
QUANTIFIER      ::= forall | exists | exists!

PATTERN-EQ      ::= PATTERN = TERM
CASE            ::= PATTERN -> TERM

PATTERN         ::= QUAL-VAR
                  | INST-QUAL-NAME
                  | PATTERN PATTERN
                  | (PATTERN ,..., PATTERN) | ()
                  | PATTERN : TYPE
                  | VAR @ PATTERN
                  | MIXFIX

MIXFIX          ::= PLACE
                  | NO-BRACKET-TOKEN
                  | [ MIXFIX ,..., MIXFIX ]  | [ ]
                  | { MIXFIX ,..., MIXFIX }  | { }
                  | ( MIXFIX ,..., MIXFIX )  | ( )
                  | MIXFIX...MIXFIX

QUAL-VAR        ::= (var VAR : TYPE)
INST-QUAL-NAME  ::= (op INST-OP-NAME : TYPESCHEME)
                  | (fun INST-OP-NAME : TYPESCHEME)
                  | (pred INST-OP-NAME : TYPESCHEME)

INST-OP-NAME    ::= OP-NAME
                  | OP-NAME [TYPE ,..., TYPE]

OP-NAME         ::= ID
TYPE-NAME       ::= ID
CLASS-NAME      ::= ID
VAR             ::= ID
TYPEVAR         ::= SIMPLE-ID
```

C.2 Disambiguation

The context-free grammar given in Section C.1 for input syntax is quite ambiguous. This section explains various precedence rules for disambiguation, and the intended grouping of mixfix terms and patterns (which is to be recognized in a separate phase, dependent on the declared symbols and parsing annotations).

C.2.1 Precedence

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

$$\begin{array}{l}
\%prec\ \{\_\_ \Leftrightarrow \_\_\} < \{\_\_ \Rightarrow \_\_, \_\_\ \mathit{if}\ \_\_\} \\
\%prec\ \{\_\_ \Rightarrow \_\_, \_\_\ \mathit{if}\ \_\_\} < \{\_\_ \wedge \_\_, \_\_ \vee \_\_\} \\
\%prec\ \{\_\_ \wedge \_\_, \_\_ \vee \_\_\} < \{\_\_ = \_\_, \_\_ \overset{e}{=} \_\_\} \\
\%prec\ \{\_\_ = \_\_, \_\_ \overset{e}{=} \_\_\} < \{\_\_\ \mathit{when}\ \_\_\ \mathit{else}\ \_\_\} \\
\%left\_assoc\ \_\_\ \mathit{if}\ \_\_, \_\_ \wedge \_\_, \_\_ \vee \_\_ \\
\%right\_assoc\ \_\_ \Rightarrow \_\_, \_\_\ \mathit{when}\ \_\_\ \mathit{else}\ \_\_
\end{array}$$

'`QUANTIFIER VAR-DECL;... . FORMULA`' has the lowest precedence of the term constructs, with the last `FORMULA` extending as far to the right as possible, e.g., '`forall x:S . F => G`' is disambiguated as '`forall x:S . (F => G)`', not as '`(forall x:S . F) => G`'.

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

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

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

- Applications of all postfix symbols have the highest precedence. This extends to all mixfix operation symbols of the form '`__ ... __ TOKEN`'.
- Applications of all prefix symbols have the next-highest precedence within terms after postfixes. This extends to all mixfix operation symbols of the form '`TOKEN __ ... __`'.

- Applications of infix symbols have lower precedence within terms after prefixes. This extends to all mixfix symbols of the form '`__ ... __ ... __`'. Mixtures of different infix symbols may be disambiguated using *precedence annotations*. *Associativity annotations* allow iterated applications of an infix symbol symbol to be written without grouping.
- The term constructs involving types like `MEMBERSHIP`, `TYPED-TERM`, and `CAST` have the lowest precedence.

User defined infix symbols without explicit precedence annotations are given higher precedence than equality.

The precedence of the constructs `MEMBERSHIP`, `TYPED-TERM`, and `CAST` has changed in comparison with CASL, but corresponds to the precedence of Haskell, because a type is no longer a simple sort name but extends to the right as far as possible. As in CASL, the type annotation '$f(x) : nat$' refers to the whole application, but unlike in CASL, so does '$f\ x : nat$'. In order to annotate just the term x, parentheses need to be used, i.e. '$f(x : nat)$'.

C.3 Lexical Syntax

The lexical syntax of HASCASL is almost identical to that of CASL. Additional keywords are

```
case class classes deriving fun funs instance
instances internal let of program programs where
```

Moreover, there are additional reserved symbols:

```
:= .! ·! \ ->
```

Within a type context, also the following symbols have a special meaning and must not occur in a `TYPE-NAME`. They are, however, legal as an `OP-NAME`.

```
<  ?  * ×  ->?  -->  -->?
```

Furthermore, there are the predefined classes and types:

```
Type Unit Pred Logical
```

C.4 Display of Mathematical Symbols

The input symbols in the following table are to be displayed as the mathematical symbols shown below them.

`*`	`->`	`-->`	`forall`	`exists`	`/\`	`\/`	`=>`	`<=>`	`not`	`=e=`	`in`	`.`	`\`
$\times$	$\rightarrow$	$\xrightarrow{\text{cont}}$	$\forall$	$\exists$	$\wedge$	$\vee$	$\Rightarrow$	$\Leftrightarrow$	$\neg$	$\stackrel{e}{=}$	$\in$	$\bullet$	λ

When a mathematical symbol is not available (e.g., when browsing HTML on WWW) the input syntax for it may be displayed instead. Moreover, characters whose display format is in ISO Latin-1 may be used for input. This allows the direct input of the symbols displayed as '$\neg$' and '$\times$' (also ' $\bullet$ ' may be input as a raised dot), and ensures that the text of a specification as shown by a WWW browser is valid input syntax (at least in the absence of display annotations).

Note the following differences:

- '$\in$' and '*in*' in '*let* ... *in* ...'
- the structured **then** and '*then*' in '*if* ... *then* ... *else* ...'
- the structured **lambda** and '\' that are both displayed as λ

All other printing conventions and display format annotations are those of CASL, despite that identifiers within annotations are less restricted and may also be the connectives given in Section C.2.1.