A Gentle Introduction to CASL

Michel Bidoit
LSV, CNRS & ENS de Cachan, France

Peter D. Mosses
BRICS & University of Aarhus, Denmark

www.cofi.info

June 4, 2004

This CASL Tutorial is a companion document to the CASL User Manual,
by Michel Bidoit and Peter D. Mosses, published in 2004 as Springer LNCS 2900.

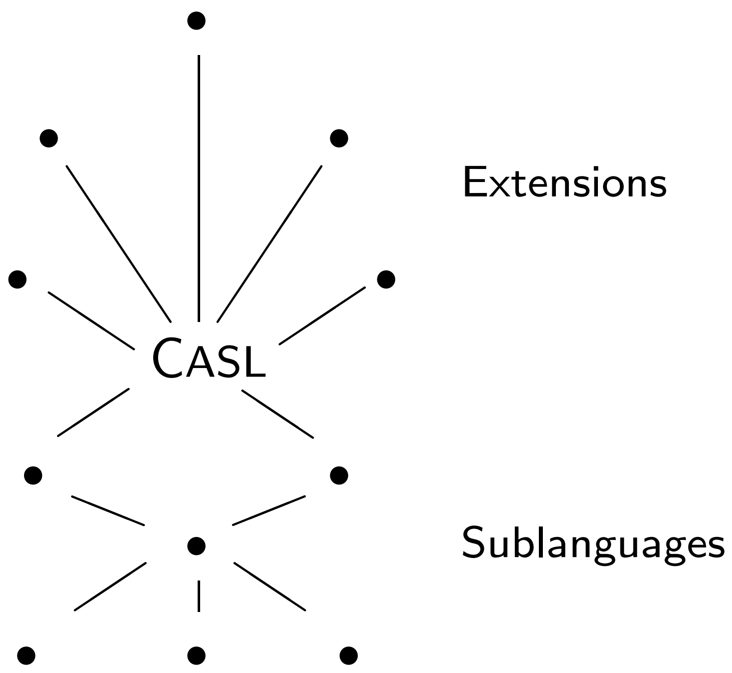
Contents

Introduction

- *There was an urgent need for a common framework.*
- *CoFI aims at establishing a wide consensus.*
- *The focus of CoFI is on* algebraic *techniques.*
- *CoFI has already achieved its main aims.*
- *CoFI is an open, voluntary initiative.*
- *CoFI has received funding as an ESPRIT Working Group, and is sponsored by IFIP WG 1.3.*
- *New participants are welcome!*

➤ *CASL has been designed as a general-purpose algebraic specification language, subsuming many existing languages.*

➤ *CASL is at the center of a* family *of languages.*



The CASL Family of Languages

➤ *CASL itself has several major* parts*.*

Underlying Concepts

➤ *CASL is based on standard concepts of algebraic specification.*

➤ *Basic specifications.*

- A basic specification declares symbols, and gives axioms and constraints.
- The semantics of a basic specification is a signature and a class of models.
- CASL specifications may declare sorts, subsorts, operations, and predicates.
- Subsorts declarations are interpreted as embeddings.
- Operations may be declared as total or partial.
- Predicates are different from boolean-valued operations.
- Operation symbols and predicate symbols may be overloaded.
- Axioms are formulas of first-order logic.
- Sort generation constraints eliminate ‘junk’ from specific carrier sets.

➤ *Structured specifications.*

- The semantics of a structured specification is simply a signature and a class of models.
- A translation merely renames symbols.
- Hiding symbols removes parts of models.
- Union of specifications identifies common symbols.
- Extension of specifications identifies common symbols too.
- Free specifications restrict models to being free, with initiality as a special case.
- Generic specifications have parameters, and have to be instantiated when referenced.

➤ *Architectural specifications and Libraries.*

- The semantics of an architectural specification reflects its modular structure.

- Architectural specifications involve the notions of persistent function and conservative extension.

- The semantics of a library of specifications is a mapping from the names of the specifications to their semantics.

Foundations

➤ *A complete presentation of CASL is in the* Reference Manual

- CASL has a definitive summary.
- CASL has a complete formal definition.
- Abstract and concrete syntax of CASL are defined formally.
- CASL has a complete formal semantics.
- CASL specifications denote classes of models.
- The semantics is largely institution-independent.
- The semantics is the ultimate reference for the meanings of all CASL constructs.
- Proof systems for various layers of CASL are provided.
- A formal refinement concept for CASL specifications is introduced.
- The foundations of our CASL are rock-solid!

Getting Started

- *Simple specifications may be written in* CASL *essentially as in many other algebraic specification languages.*
- CASL *provides also useful abbreviations.*
- CASL *allows loose, generated and free specifications.*

Loose Specifications

➤ *CASL syntax for declarations and axioms involves familiar notation, and is mostly self-explanatory.*

---

spec STRICT_PARTIAL_ORDER =

  %% Let's start with a simple example !

  sort *Elem*

  pred $\_\_ < \_\_ : Elem \times Elem$ **%% pred** abbreviates predicate

  $\forall x, y, z : Elem$

  - $\neg(x < x)$ %(strict)%
  - $x < y \Rightarrow \neg(y < x)$ %(asymmetric)%
  - $x < y \wedge y < z \Rightarrow x < z$ %(transitive)%

  %{ Note that there may exist x, y such that
    neither $x < y$ nor $y < x$. }**%**

end

➤ *Specifications can easily be extended by new declarations and axioms.*

---

spec TOTAL_ORDER =
    STRICT_PARTIAL_ORDER
then $\forall x, y : Elem \bullet x < y \vee y < x \vee x = y$ %(total)%
end

➤ *In simple cases, an operation (or a predicate) symbol may be declared and its intended interpretation defined at the same time.*

---

spec TOTAL_ORDER_WITH_MINMAX =
  TOTAL_ORDER
then **ops** $min(x, y : Elem) : Elem = x \ when \ x < y \ else \ y;$
  $max(x, y : Elem) : Elem = y \ when \ min(x, y) = x \ else \ x$
end

spec VARIANT_OF_TOTAL_ORDER_WITH_MINMAX =
TOTAL_ORDER
then **vars** $x, y : Elem$
op $min : Elem \times Elem \rightarrow Elem$
- $x < y \Rightarrow min(x, y) = x$
- $\neg(x < y) \Rightarrow min(x, y) = y$

op $max : Elem \times Elem \rightarrow Elem$
- $x < y \Rightarrow max(x, y) = y$
- $\neg(x < y) \Rightarrow max(x, y) = x$

end

➤ *Symbols may be conveniently displayed as usual mathematical symbols by means of* **%display** *annotations.*

%display __<=__ %LATEX __ $\leq$ __

spec PARTIAL_ORDER =
    STRICT_PARTIAL_ORDER
then **pred** $\_\_ \leq \_\_(x, y : Elem) \Leftrightarrow (x < y \vee x = y)$
end

➤ *The* **%implies** *annotation is used to indicate that some axioms are supposedly redundant, being consequences of others.*

---

spec PARTIAL_ORDER_1 =
    PARTIAL_ORDER
then **%implies**
    $\forall x, y, z : Elem \bullet\ x \leq y \wedge y \leq z \Rightarrow x \leq z$ %(transitive)%
end

spec IMPLIES_DOES_NOT_HOLD =
    PARTIAL_ORDER
then **%implies**
    $\forall x, y : Elem \bullet\ x < y \vee y < x \vee x = y$ %(total)%
end

➤ *Attributes may be used to abbreviate axioms for associativity, commutativity, idempotence, and unit properties.*

---

spec MONOID =
  sort *Monoid*
  ops *1* : *Monoid*;
    $\_\_ * \_\_ : Monoid \times Monoid \rightarrow Monoid, \; assoc, \; unit\ 1$
end

➤ *Genericity of specifications can be made explicit using parameters.*

---

spec GENERIC_MONOID [**sort** $Elem$] =

sort $Monoid$

ops $inj : Elem \rightarrow Monoid;$

$1 : Monoid;$

$\_\_ * \_\_ : Monoid \times Monoid \rightarrow Monoid, \; assoc, \; unit \; 1$

$\forall x, y : Elem \bullet inj(x) = inj(y) \Rightarrow x = y$

end

spec NON_GENERIC_MONOID =

  sort *Elem*

then **sort** *Monoid*

  ops $inj \quad : Elem \to Monoid;$

  $1 \quad : Monoid;$

  $\_\_ * \_\_ : Monoid \times Monoid \to Monoid, \; assoc, \; unit\ 1$

  $\forall x, y : Elem \bullet \; inj(x) = inj(y) \Rightarrow x = y$

end

➤ *References to generic specifications always instantiate the parameters.*

---

spec GENERIC_COMMUTATIVE_MONOID [**sort** *Elem*] =
GENERIC_MONOID [**sort** *Elem*]
then $\forall x, y : Monoid \bullet x * y = y * x$
end

spec GENERIC_COMMUTATIVE_MONOID_1 [**sort** *Elem*] =
GENERIC_MONOID [**sort** *Elem*]
then **op** $\_\_ * \_\_ : Monoid \times Monoid \rightarrow Monoid,\ comm$
end

➤ *Datatype declarations may be used to abbreviate declarations of sorts and constructors.*

---

spec CONTAINER [**sort** $Elem$] =

type $Container ::= empty \mid insert(Elem; Container)$

pred $\_\_is\_in\_\_ : Elem \times Container$

$\forall e, e' : Elem; \; C : Container$

- $\neg(e \; is\_in \; empty)$
- $e \; is\_in \; insert(e', C) \Leftrightarrow (e = e' \vee e \; is\_in \; C)$

end

➤ *Loose datatype declarations are appropriate when further constructors may be added in extensions.*

---

spec MARKING_CONTAINER [**sort** $Elem$] =
CONTAINER [**sort** $Elem$]
then **type** $Container ::= mark\_insert(Elem; \ Container)$
pred $\_\_is\_marked\_in\_\_ : Elem \times Container$
$\forall e, e' : Elem; \ C : Container$

- $e \ is\_in \ mark\_insert(e', C) \Leftrightarrow (e = e' \vee e \ is\_in \ C)$
- $\neg(e \ is\_marked\_in \ empty)$
- $e \ is\_marked\_in \ insert(e', C) \Leftrightarrow e \ is\_marked\_in \ C$
- $e \ is\_marked\_in \ mark\_insert(e', C) \Leftrightarrow (e = e' \vee e \ is\_marked\_in \ C)$

end

Generated Specifications

➤ *Sorts may be specified as generated by their constructors.*

---

spec GENERATED_CONTAINER [**sort** $Elem$] =

  generated type $Container ::= empty \mid insert(Elem; Container)$

  pred $\_\_is\_in\_\_ : Elem \times Container$

  $\forall e, e' : Elem; \; C : Container$

  - $\neg(e \; is\_in \; empty)$
  - $e \; is\_in \; insert(e', C) \Leftrightarrow (e = e' \vee e \; is\_in \; C)$

end

➤ *Generated specifications are in general loose.*

---

spec GENERATED_CONTAINER_MERGE [**sort** $Elem$] =
GENERATED_CONTAINER [**sort** $Elem$]
then **op** $\_\_merge\_\_ : Container \times Container \rightarrow Container$
$\forall e : Elem;\ C, C' : Container$
• $e\ is\_in\ (C\ merge\ C') \Leftrightarrow (e\ is\_in\ C \vee e\ is\_in\ C')$
end

➤ *Generated specifications need not be loose.*

---

spec GENERATED_SET [**sort** $Elem$] =

generated type $Set ::= empty \mid insert(Elem; Set)$

pred $\_\_is\_in\_\_ : Elem \times Set$

ops $\{\_\_\}(e : Elem) : Set = insert(e, empty);$

$\_\_ \cup \_\_ \quad : Set \times Set \to Set;$

$remove \quad : Elem \times Set \to Set$

$\forall e, e' : Elem; \; S, S' : Set$

- $\neg(e \; is\_in \; empty)$
- $e \; is\_in \; insert(e', S) \Leftrightarrow (e = e' \vee e \; is\_in \; S)$
- $S = S' \Leftrightarrow (\forall x : Elem \bullet x \; is\_in \; S \Leftrightarrow x \; is\_in \; S')$ %(equal_sets)%
- $e \; is\_in \; (S \cup S') \Leftrightarrow (e \; is\_in \; S \vee e \; is\_in \; S')$
- $e \; is\_in \; remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e \; is\_in \; S)$

then %implies $\forall e, e' : Elem; \; S : Set$

- $insert(e, insert(e, S)) = insert(e, S)$
- $insert(e, insert(e', S)) = insert(e', insert(e, S))$

generated type $Set ::= empty \mid \{\_\_\}(Elem) \mid \_\_ \cup \_\_(Set; Set)$

op $\_\_ \cup \_\_ : Set \times Set \to Set, \; assoc, \; comm, \; idem, \; unit \; empty$

end

➤ *Generated types may need to be declared together.*

---

sort *Node*
generated type $Tree ::= mktree(Node;\ Forest)$
generated type $Forest ::= empty \mid add(Tree;\ Forest)$

is both *incorrect* (linear visibility) and *wrong* (the corresponding semantics is not the "expected" one). One must write instead:

sort *Node*
generated types $Tree \quad ::= mktree(Node;\ Forest);$
$\qquad\qquad\qquad\quad Forest ::= empty \mid add(Tree;\ Forest)$

Free Specifications

➤ *Free specifications provide initial semantics and avoid the need for explicit negation.*

---

spec NATURAL = **free type** $Nat ::= 0 \mid suc(Nat)$

➤ *Free datatype declarations are particularly convenient for defining enumerated datatypes.*

---

spec COLOR =

  free type $RGB ::= Red \mid Green \mid Blue$

  free type $CMYK ::= Cyan \mid Magenta \mid Yellow \mid Black$

end

➤ *Free specifications can also be used when the constructors are related by some axioms.*

---

spec INTEGER =

  free { **type** $Int ::= 0 \mid suc(Int) \mid pre(Int)$

    $\forall x : Int \bullet \; suc(pre(x)) = x$

      $\bullet \; pre(suc(x)) = x$ }

end

➤ *Predicates hold minimally in models of free specifications.*

---

spec NATURAL_ORDER =
    NATURAL
then **free** { **pred** $\_\_ < \_\_ : Nat \times Nat$
    $\forall x, y : Nat$
    • $0 < suc(x)$
    • $x < y \Rightarrow suc(x) < suc(y)$ }
end

➤ *Operations and predicates may be safely defined by induction on the constructors of a free datatype declaration.*

---

spec NATURAL_ARITHMETIC =
NATURAL_ORDER
then **ops** $1 : Nat = suc(0);$
$\_\_+\_\_ : Nat \times Nat \to Nat, \ assoc, \ comm, \ unit\ 0;$
$\_\_*\_\_ : Nat \times Nat \to Nat, \ assoc, \ comm, \ unit\ 1$

$\forall x, y : Nat$

- $x + suc(y) = suc(x + y)$
- $x * 0 = 0$
- $x * suc(y) = (x * y) + x$

end

➤ *More care may be needed when defining operations or predicates on free datatypes when there are axioms relating the constructors.*

---

spec INTEGER_ARITHMETIC =
INTEGER
then **ops** $1 : Int = suc(0);$
$\_\_ + \_\_ : Int \times Int \to Int, \ assoc, \ comm, \ unit\ 0;$
$\_\_ - \_\_ : Int \times Int \to Int;$
$\_\_ * \_\_ : Int \times Int \to Int, \ assoc, \ comm, \ unit\ 1$

$\forall x, y : Int$

- $x + suc(y) = suc(x + y)$
- $x + pre(y) = pre(x + y)$
- $x - 0 = x$
- $x - suc(y) = pre(x - y)$
- $x - pre(y) = suc(x - y)$
- $x * 0 = 0$
- $x * suc(y) = (x * y) + x$
- $x * pre(y) = (x * y) - x$

end

spec INTEGER_ARITHMETIC_ORDER =

INTEGER_ARITHMETIC

then **preds** $\_\_ \leq \_\_,\ \_\_ \geq \_\_,\ \_\_ < \_\_,\ \_\_ > \_\_ : Int \times Int$

$\forall x, y : Int$

- $0 \leq 0$
- $\neg(0 \leq pre(0))$
- $0 \leq x \Rightarrow 0 \leq suc(x)$
- $\neg(0 \leq x) \Rightarrow \neg(0 \leq pre(x))$
- $suc(x) \leq y \Leftrightarrow x \leq pre(y)$
- $pre(x) \leq y \Leftrightarrow x \leq suc(y)$
- $x \geq y \Leftrightarrow y \leq x$
- $x < y \Leftrightarrow (x \leq y \wedge \neg(x = y))$
- $x > y \Leftrightarrow y < x$

end

➤ *Generic specifications often involve free extensions of (loose) parameters.*

---

spec LIST [**sort** $Elem$] = **free type** $List ::= empty \mid cons(Elem; \ List)$

spec SET [**sort** $Elem$] =

free { **type** $Set ::= empty \mid insert(Elem; \ Set)$

pred $\_\_is\_in\_\_ : Elem \times Set$

$\forall e, e' : Elem; \ S : Set$

- $insert(e, insert(e, S)) = insert(e, S)$
- $insert(e, insert(e', S)) = insert(e', insert(e, S))$
- $\neg(e \ is\_in \ empty)$
- $e \ is\_in \ insert(e, S)$
- $e \ is\_in \ insert(e', S) \ if \ e \ is\_in \ S$ }

end

spec TRANSITIVE_CLOSURE [**sort** *Elem* **pred** $\_\_R\_\_ : Elem \times Elem$] =

  free { **pred** $\_\_R^+\_\_ : Elem \times Elem$

  $\forall x, y, z : Elem$

  - $x \; R \; y \Rightarrow x \; R^+ y$
  - $x \; R^+ y \wedge y \; R^+ z \Rightarrow x \; R^+ z$ }

➤ *Loose extensions of free specifications can avoid overspecification.*

---

spec NATURAL_WITH_BOUND =
    NATURAL_ARITHMETIC
then **op** $max\_size : Nat$
    • $0 < max\_size$
end

spec SET_CHOOSE [**sort** $Elem$] =
    SET [**sort** $Elem$]
then **op** $choose : Set \rightarrow Elem$
    $\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$
end

➤ *Datatypes with observer operations or predicates can be specified as generated instead of free.*

---

spec SET_GENERATED [**sort** $Elem$] =

  generated type $Set ::= empty \mid insert(Elem; Set)$

  pred $\_\_is\_in\_\_ : Elem \times Set$

  $\forall e, e' : Elem; \ S, S' : Set$

  - $\neg(e \ is\_in \ empty)$
  - $e \ is\_in \ insert(e', S) \Leftrightarrow (e = e' \vee e \ is\_in \ S)$
  - $S = S' \Leftrightarrow (\forall x : Elem \bullet x \ is\_in \ S \Leftrightarrow x \ is\_in \ S')$

end

➤ *The* **%def** *annotation is useful to indicate that some operations or predicates are uniquely defined.*

---

spec SET_UNION [**sort** *Elem*] =
    SET [**sort** *Elem*]
then **%def**
    ops $\_\_ \cup \_\_ : Set \times Set \to Set,\ assoc,\ comm,\ idem,\ unit\ empty;$
        $remove : Elem \times Set \to Set$

    $\forall e, e' : Elem;\ S, S' : Set$

- $S \cup insert(e', S') = insert(e', S \cup S')$
- $remove(e, empty) = empty$
- $remove(e, insert(e, S)) = remove(e, S)$
- $remove(e, insert(e', S)) = insert(e', remove(e, S))\ if\ \neg(e = e')$

end

➤ *Operations can be defined by axioms involving observer operations, instead of inductively on constructors.*

---

spec SET_UNION_1 [**sort** $Elem$] =
  SET_GENERATED [**sort** $Elem$]
then %def
  ops $\_\_ \cup \_\_ : Set \times Set \to Set$, $assoc$, $comm$, $idem$, $unit\ empty$;
    $remove : Elem \times Set \to Set$

  $\forall e, e' : Elem; \ S, S' : Set$

  - $e\ is\_in\ (S \cup S') \Leftrightarrow (e\ is\_in\ S \vee e\ is\_in\ S')$
  - $e\ is\_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is\_in\ S)$

end

➤ *Sorts declared in free specifications are not necessarily generated by their constructors.*

---

spec UNNATURAL =

  free { **type** $UnNat ::= 0 \mid suc(UnNat)$

    op $\_\_ + \_\_ : UnNat \times UnNat \rightarrow UnNat,$

      $assoc, \; comm, \; unit \; 0$

    $\forall x, y : UnNat \bullet x + suc(y) = suc(x + y)$

    $\forall x : UnNat \bullet \exists y : UnNat \bullet x + y = 0$ }

end

Partial Functions

- *Partial functions arise naturally.*

➤ *Partial functions are declared differently from total functions.*

---

spec SET_PARTIAL_CHOOSE [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **op** $choose : Set \rightarrow ?\ Elem$
end

➤ *Terms containing partial functions may be undefined, i.e., they may fail to denote any value.*

E.g., the (value of the) term $choose(empty)$ may be undefined.

➤ *Functions, even total ones, propagate undefinedness.*

If the term $choose(S)$ is undefined for some value of S,
then the term $insert(choose(S), S')$ is undefined as well for this value of S,
although $insert$ is a total function.

➤ *Predicates do not hold on undefined arguments.*

If the term $choose(S)$ is undefined,
then the atomic formula $choose(S)\ is\_in\ S$ does not hold.

➤ *Equations hold when both terms are undefined.*

The ordinary equation:

$$insert(choose(S), insert(choose(S), empty)) = insert(choose(S), empty)$$

holds also when the term $choose(S)$ is undefined.

➤ *Special care is needed in specifications involving partial functions.*

- Asserting $choose(S)\ is\_in\ S$ as an axiom implies that $choose(S)$ is defined, for any S.
- Asserting $remove(choose(S), insert(choose(S), empty)) = empty$ as an axiom implies that $choose(S)$ is defined for any S, since the term $empty$ is always defined.
- Asserting $insert(choose(S), S) = S$ as an axiom implies that $choose(S)$ is defined for any S, since a variable always denotes a defined value.

➤ *The definedness of a term can be checked or asserted.*

---

spec SET_PARTIAL_CHOOSE_1 [**sort** $Elem$] =
    SET_PARTIAL_CHOOSE [**sort** $Elem$]
then $\bullet\ \neg\ def\ choose(empty)$
    $\forall S : Set \bullet\ def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$
end

We know that $choose$ is undefined when applied to $empty$,
but we don't know exactly when $choose(S)$ is defined.
(It may be undefined on other values than $empty$.)

If we would have specified $choose$ by:

$$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$$

then we could conclude that $choose(S)$ is defined when S is not equal to $empty$,
but nothing about the undefinedness of $choose(empty)$.

➤ *The domains of definition of partial functions can be specified exactly.*

---

spec SET_PARTIAL_CHOOSE_2 [**sort** $Elem$] =
  SET_PARTIAL_CHOOSE [**sort** $Elem$]
then $\forall S : Set \bullet def\ choose(S) \Leftrightarrow \neg(S = empty)$
  $\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$
end

➤ *Loosely specified domains of definition may be useful.*

---

spec NATURAL_WITH_BOUND_AND_ADDITION =

NATURAL_WITH_BOUND

then **op** $\_\_+?\_\_ : Nat \times Nat \rightarrow? Nat$

$\forall x, y : Nat$

- $def(x+?y)$ *if* $x + y < max\_size$

%{ $x + y < max\_size$ implies both

$x < max\_size$ and $y < max\_size$ }%

- $def(x+?y) \Rightarrow x+?y = x + y$

end

➤ *Domains of definition can be specified more or less explicitly.*

spec SET_PARTIAL_CHOOSE_3 [**sort** *Elem*] =
  SET_PARTIAL_CHOOSE [**sort** *Elem*]
then • $\neg$ *def* $choose(empty)$
  $\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$
end

We can conclude after some reasoning that:

$$def\ choose(S) \Leftrightarrow \neg(S = empty)$$

but this is not so prominent.

spec NATURAL_PARTIAL_PRE =
NATURAL_ARITHMETIC
then op $pre : Nat \rightarrow ?\ Nat$
$\bullet\ \neg\ def\ pre(0)$
$\forall x : Nat\ \bullet\ pre(suc(x)) = x$
end

is explicit enough.

spec NATURAL_PARTIAL_SUBTRACTION_1 =
NATURAL_PARTIAL_PRE
then **op** $\_\_ - \_\_ : Nat \times Nat \rightarrow ? \; Nat$
$\forall x, y : Nat$

- $x - 0 = x$
- $x - suc(y) = pre(x - y)$

end

is correct, but clearly not explicit enough, and better specified as follows:

spec NATURAL_PARTIAL_SUBTRACTION =
NATURAL_PARTIAL_PRE
then **op** $\_\_ - \_\_ : Nat \times Nat \rightarrow ? \; Nat$
$\forall x, y : Nat$

- $def(x - y) \Leftrightarrow (y < x \vee y = x)$
- $x - 0 = x$
- $x - suc(y) = pre(x - y)$

end

➤ *Partial functions are minimally defined by default in free specifications.*

---

spec LIST_SELECTORS_1 [**sort** *Elem*] =
    LIST [**sort** *Elem*]
then **free** { **ops** $head : List \rightarrow ? \; Elem;$
    $tail \;\; : List \rightarrow ? \; List$

$\forall e : Elem; \; L : List$

- $head(cons(e, L)) = e$
- $tail(cons(e, L)) = L$ }

end

spec LIST_SELECTORS_2 [**sort** *Elem*] =
LIST [**sort** *Elem*]
then **ops** $head : List \to ?\ Elem;$
$tail : List \to ?\ List$

$\forall e : Elem;\ L : List$

- $\neg\ def\ head(empty)$
- $\neg\ def\ tail(empty)$
- $head(cons(e, L)) = e$
- $tail(cons(e, L)) = L$

end

➤ *Selectors can be specified concisely in datatype declarations, and are usually partial.*

---

spec LIST_SELECTORS [**sort** $Elem$] =
    free type $List ::= empty \mid cons(head :? Elem; tail :? List)$

spec NATURAL_SUC_PRE = **free type** $Nat ::= 0 \mid suc(pre :? Nat)$

➤ *Selectors are usually total when there is only one constructor.*

---

spec PAIR_1 [**sorts** $Elem1, Elem2$] =
    free type $Pair ::= pair(first : Elem1; second : Elem2)$

➤ *Constructors may be partial.*

---

spec PART_CONTAINER [**sort** $Elem$] =

  generated type

    $P\_Container ::= empty \mid insert(Elem; P\_Container)?$

  pred $addable : Elem \times P\_Container$

  vars $e, e' : Elem;\ C : P\_Container$

  • $def\ insert(e, C) \Leftrightarrow addable(e, C)$

  pred $\_\_is\_in\_\_ : Elem \times P\_Container$

  • $\neg(e\ is\_in\ empty)$

  • $(e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C))\ if\ addable(e', C)$

end

➤ *Existential equality requires the definedness of both terms as well as their equality.*

---

spec NATURAL_PARTIAL_SUBTRACTION_2 =
NATURAL_PARTIAL_SUBTRACTION_1
then $\forall x, y, z : Nat \bullet\ y - x \stackrel{e}{=} z - x \Rightarrow y = z$
%{ $y - x = z - x \Rightarrow y = z$ would be wrong,
$def(y - x) \wedge def(z - x) \wedge y - x = z - x \Rightarrow y = z$
is correct, but better abbreviated in the above axiom **}%**
end

Subsorting

➤ *Subsorts and supersorts are often useful in CASL specifications.*

➤ *Subsort declarations directly express relationships between carrier sets.*

---

spec GENERIC_MONOID_1 [**sort** *Elem*] =

  sorts $Elem < Monoid$

  ops $1 : Monoid;$

  $\_\_ * \_\_ : Monoid \times Monoid \rightarrow Monoid,\ assoc,\ unit\ 1$

end

➤ *Operations declared on a sort are automatically inherited by its subsorts.*

---

spec VEHICLE =
  NATURAL
then **sorts** $Car, \ Bicycle < Vehicle$
  ops $max\_speed : Vehicle \to Nat;$
  $weight : Vehicle \to Nat;$
  $engine\_capacity : Car \to Nat$
end

➤ *Inheritance applies also for subsorts that are declared afterwards.*

---

spec MORE_VEHICLE = VEHICLE **then sorts** $Boat < Vehicle$

➤ *Subsort membership can be checked or asserted.*

---

spec SPEED_REGULATION =
  VEHICLE
then **ops** $speed\_limit : Vehicle \rightarrow Nat;$
    $car\_speed\_limit,\ bike\_speed\_limit : Nat$
  $\forall v : Vehicle$
  - $v \in Car \Rightarrow speed\_limit(v) = car\_speed\_limit$
  - $v \in Bicycle \Rightarrow speed\_limit(v) = bike\_speed\_limit$

end

➤ *Datatype declarations can involve subsort declarations.*

---

sorts $Car, \; Bicycle, \; Boat$
type $Vehicle ::= sort \; Car \mid sort \; Bicycle \mid sort \; Boat$

is equivalent to the declaration **sorts** $Car, \; Bicycle, \; Boat < Vehicle$, and leaves the way open to further kinds of vehicles (e.g., planes).

sorts $Car, \; Bicycle, \; Boat$
generated type $Vehicle ::= sort \; Car \mid sort \; Bicycle \mid sort \; Boat$

prevents the definition of further subsorts, e.g., for planes.

sorts $Car, \; Bicycle, \; Boat$
free type $Vehicle ::= sort \; Car \mid sort \; Bicycle \mid sort \; Boat$

prevents the definition of further subsorts, and moreover the definition of a common subsort of both Car and $Boat$ (e.g., **sorts** $Amphibious < Car, Boat$).

➤ *Subsorts may also arise as classifications of previously specified values, and their values can be explicitly defined.*

---

spec NATURAL_SUBSORTS =
  NATURAL_ARITHMETIC
then **pred** $even : Nat$
  • $even(0)$
  • $\neg\ even(1)$
  $\forall n : Nat \bullet\ even(suc(suc(n))) \Leftrightarrow even(n)$
  sort $Even = \{x : Nat \bullet\ even(x)\}$
  sort $Prime = \{x : Nat \bullet\ 1 < x\ \wedge$
    $\forall y, z : Nat \bullet\ x = y * z \Rightarrow y = 1 \vee z = 1\}$
end

spec POSITIVE =
  NATURAL_PARTIAL_PRE
then **sort** $Pos = \{x : Nat \bullet\ \neg(x = 0)\}$

➤ *It may be useful to redeclare previously defined operations, using the new subsorts introduced.*

---

spec POSITIVE_ARITHMETIC =
POSITIVE
then ops $1 : Pos;$
$suc : Nat \rightarrow Pos;$
$\_\_ + \_\_, \_\_ * \_\_ : Pos \times Pos \rightarrow Pos;$
$\_\_ + \_\_ : Pos \times Nat \rightarrow Pos;$
$\_\_ + \_\_ : Nat \times Pos \rightarrow Pos$
end

➤ *A subsort may correspond to the definition domain of a partial function.*

---

spec POSITIVE_PRE =
POSITIVE_ARITHMETIC
then **op** $pre : Pos \longrightarrow Nat$

➤ *Using subsorts may avoid the need for partial functions.*

---

spec NATURAL_POSITIVE_ARITHMETIC =

free types $Nat ::= 0 \mid sort\ Pos;$

$Pos ::= suc(pre : Nat)$

ops $1 : Pos = suc(0);$

$\_\_+\_\_ : Nat \times Nat \rightarrow Nat,\ assoc,\ comm,\ unit\ 0;$

$\_\_*\_\_ : Nat \times Nat \rightarrow Nat,\ assoc,\ comm,\ unit\ 1;$

$\_\_+\_\_,\ \_\_*\_\_ : Pos \times Pos \rightarrow Pos;$

$\_\_+\_\_ : Pos \times Nat \rightarrow Pos;$

$\_\_+\_\_ : Nat \times Pos \rightarrow Pos$

$\forall x, y : Nat$

- $x + suc(y) = suc(x + y)$
- $x * 0 = 0$
- $x * suc(y) = x + (x * y)$

end

➤ *Casting a term from a supersort to a subsort is explicit and the value of the cast may be undefined.*

Casting a term t to a sort s is written $t\ as\ s$,
and $def\ (t\ as\ s)$ is equivalent to $t \in s$.

- $pre(\ pre(suc(1))\ as\ Pos\ )$
- $def\ \ pre(\ pre(suc(1))\ as\ Pos\ )$
- $\neg\ def(pre(\ pre(suc(1))\ as\ Pos\ )\ as\ Pos)$

➤ *Supersorts may be useful when generalizing previously specified sorts.*

---

spec INTEGER_ARITHMETIC_1 =
NATURAL_POSITIVE_ARITHMETIC
then **free type** $Int ::= sort\ Nat \mid -\_\_(Pos)$
ops $\_\_ + \_\_ : Int \times Int \to Int,\ assoc,\ comm,\ unit\ 0;$
$\_\_ - \_\_ : Int \times Int \to Int;$
$\_\_ * \_\_ : Int \times Int \to Int,\ assoc,\ comm,\ unit\ 1$

$\forall x : Int;\ n : Nat;\ p, q : Pos$

- $suc(n) + (-1) = n$
- $suc(n) + (-suc(q)) = n + (-q)$
- $(-p) + (-q) = -(p + q)$
- $x - 0 = x$
- $x - p = x + (-p)$
- $x - (-q) = x + q$
- $0 * (-q) = 0$
- $p * (-q) = -(p * q)$
- $(-p) * (-q) = p * q$

end

➤ *Supersorts may also be used for extending the intended values by new values representing errors or exceptions.*

---

spec SET_ERROR_CHOOSE [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **sorts** $Elem < ElemError$
op $choose : Set \to ElemError$
pred $\_\_is\_in\_\_ : ElemError \times Set$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S) \in Elem \wedge choose(S)\ is\_in\ S$
end

spec SET_ERROR_CHOOSE_1 [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **sorts** $Elem < ElemError$
op $choose : Set \to ElemError$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow (choose(S)\ as\ Elem)\ is\_in\ S$
end

Structuring Specifications

➤ *Large and complex specifications are easily built out of simpler ones by means of (a small number of) specification-building operations.*

➤ *Union and extension can be used to structure specifications.*

---

spec LIST_SET [**sort** *Elem*] =
  LIST_SELECTORS [**sort** *Elem*]
and GENERATED_SET [**sort** *Elem*]
then **op** $elements\_of\_\_ : List \rightarrow Set$
  $\forall e : Elem;\ L : List$
  - $elements\_of\ empty = empty$
  - $elements\_of\ cons(e, L) = \{e\} \cup elements\_of\ L$

end

➤ *Specifications may combine parts with loose, generated, and free interpretations.*

---

spec LIST_CHOOSE [**sort** $Elem$] =
    LIST_SELECTORS [**sort** $Elem$]
and SET_PARTIAL_CHOOSE_2 [**sort** $Elem$]
then **ops** $elements\_of\ \_\_ : List \to Set;$
    $choose : List \to ?\ Elem$

$\forall e : Elem;\ L : List$

- $elements\_of\ empty = empty$
- $elements\_of\ cons(e, L) = \{e\} \cup elements\_of\ L$
- $def\ choose(L) \Leftrightarrow \neg(L = empty)$
- $choose(L) = choose(elements\_of\ L)$

end

spec SET_TO_LIST [**sort** *Elem*] =
LIST_SET [**sort** *Elem*]
then **op** $list\_of\_\_ : Set \rightarrow List$
$\forall S : Set \bullet elements\_of(list\_of\ S) = S$
end

➤ *Renaming may be used to avoid unintended name clashes, or to adjust names of sorts and change notations for operations and predicates.*

---

spec STACK [**sort** $Elem$] =

  LIST_SELECTORS [**sort** $Elem$] **with sort** $List \mapsto Stack$,

ops $cons \mapsto push\_\_onto\_\_$,

$head \mapsto top$,

$tail \mapsto pop$

end

➤ *When combining specifications, origins of symbols can be indicated.*

---

spec LIST_SET_1 [**sort** $Elem$] =
    LIST_SELECTORS [**sort** $Elem$] **with** $empty$, $cons$
and GENERATED_SET [**sort** $Elem$] **with** $empty$, $\{\_\_\}$, $\_\_ \cup \_\_$
then **op** $elements\_of\_\_ : List \rightarrow Set$
    $\forall e : Elem; \ L : List$
    • $elements\_of \ empty = empty$
    • $elements\_of \ cons(e, L) = \{e\} \cup elements\_of \ L$
end

➤ *Auxiliary symbols used in structured specifications can be hidden.*

---

spec NATURAL_PARTIAL_SUBTRACTION_3 =
    NATURAL_PARTIAL_SUBTRACTION_1 **hide** *suc*, *pre*
end

spec NATURAL_PARTIAL_SUBTRACTION_4 =
    NATURAL_PARTIAL_SUBTRACTION_1
    reveal *Nat*, *0*, *1*, $\_\_ + \_\_$, $\_\_ - \_\_$, $\_\_ * \_\_$, $\_\_ < \_\_$
end

spec PARTIAL_ORDER_2 = PARTIAL_ORDER **reveal pred** $\_\_ \leq \_\_$

➤ *Auxiliary symbols can be made local when they do not need to be exported.*

---

spec LIST_ORDER [TOTAL_ORDER **with sort** $Elem$, **pred** $\_\_ < \_\_$] =
  LIST_SELECTORS [**sort** $Elem$]
then local **op** $insert : Elem \times List \to List$
  $\forall e, e' : Elem;\ L : List$
  - $insert(e, empty) = cons(e, empty)$
  - $insert(e, cons(e', L)) = cons(e', insert(e, L))$ *when* $e' < e$ *else* $cons(e, cons(e', L))$

  within **op** $order : List \to List$
  $\forall e : Elem;\ L : List$
  - $order(empty) = empty$
  - $order(cons(e, L)) = insert(e, order(L))$

end

spec LIST_ORDER_SORTED
[TOTAL_ORDER **with sort** *Elem*, **pred** $\_\_ < \_\_$] =
LIST_SELECTORS [**sort** *Elem*]
then local **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
- $empty\ is\_sorted$
- $cons(e, empty)\ is\_sorted$
- $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
  $cons(e', L)\ is\_sorted \wedge \neg(e' < e)$

within op $order : List \to List$
$\forall L : List \bullet order(L)\ is\_sorted$
end

➤ *Care is needed with local sort declarations.*

---

spec WRONG_LIST_ORDER_SORTED
  [TOTAL_ORDER **with sort** $Elem$, **pred** $\_\_ < \_\_$] =
  LIST_SELECTORS [**sort** $Elem$]
then local **pred** $\_\_is\_sorted : List$
  sort $SortedList = \{L : List \bullet L\ is\_sorted\}$
  $\forall e, e' : Elem;\ L : List$
  • $empty\ is\_sorted$
  • $cons(e, empty)\ is\_sorted$
  • $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
    $cons(e', L)\ is\_sorted \wedge \neg(e' < e)$
  within op $order : List \rightarrow SortedList$
end

spec LIST_ORDER_SORTED_2
[TOTAL_ORDER **with sort** *Elem*, **pred** __ < __] =
LIST_SELECTORS [**sort** *Elem*]
then local **pred** __*is_sorted* : *List*
$\forall e, e' : Elem;\ L : List$
- $empty\ is\_sorted$
- $cons(e, empty)\ is\_sorted$
- $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
  $cons(e', L)\ is\_sorted \wedge \neg(e' < e)$

within **sort** $SortedList = \{L : List \bullet L\ is\_sorted\}$
op $order : List \rightarrow SortedList$
end

```
spec LIST_ORDER_SORTED_3
     [TOTAL_ORDER with sort Elem, pred __ < __] =
     LIST_SELECTORS [sort Elem]
then {     pred __is_sorted : List
           ∀e, e′ : Elem; L : List
           • empty is_sorted
           • cons(e, empty) is_sorted
           • cons(e, cons(e′, L)) is_sorted ⇔
                 cons(e′, L) is_sorted ∧ ¬(e′ < e)
     then  sort SortedList = {L : List • L is_sorted}
           op   order : List → SortedList
     } hide __is_sorted
end
```

➤ *Naming a specification allows its reuse.*

It is in general advisable to define as many named specifications as felt appropriate, since this improves the reusability of specifications: a named specification can easily be reused by referring to its name.

Generic Specifications

➤ *Making a specification generic (when appropriate) improves its reusability.*

➤ *Parameters are arbitrary specifications.*

---

spec GENERIC_MONOID [**sort** $Elem$] = ...

spec LIST_SELECTORS [**sort** $Elem$] = ...

spec LIST_ORDER [TOTAL_ORDER **with sort** $Elem$, **pred** $\_\_ < \_\_$] = ...

➤ *The argument specification of an instantiation must provide symbols corresponding to those required by the parameter.*

---

spec LIST_ORDER_NAT = LIST_ORDER [NATURAL_ORDER]

➤ *The argument specification of an instantiation must ensure that the properties required by the parameter hold.*

---

spec NAT_WORD = GENERIC_MONOID [NATURAL]

spec LIST_ORDER_NAT = LIST_ORDER [NATURAL_ORDER]

The definition of NAT_WORD abbreviates:
NATURAL **and** { NON_GENERIC_MONOID **with** $Elem \mapsto Nat$ }.

➤ *When convenient, an instantiation can be completed by a renaming.*

spec NAT_WORD_1 =
GENERIC_MONOID [NATURAL]
with $Monoid \mapsto Nat\_Word$
end

➤ *There must be no shared symbols between the argument specification and the body of the instantiated generic specification.*

---

spec THIS_IS_WRONG = GENERIC_MONOID [MONOID]

The above instantiation is ill-formed since the sort *Monoid* and the operation symbols '*1*' and '$*$' are shared between the body of the generic specification GENERIC_MONOID and the argument specification MONOID.

➤ *In instantiations, the fitting of parameter symbols to identical argument symbols can be left implicit.*

---

spec GENERIC_COMMUTATIVE_MONOID [**sort** *Elem*] =
    GENERIC_MONOID [**sort** *Elem*]
then ...

➤ *The fitting of parameter sorts to unique argument sorts can also be left implicit.*

➤ *Fitting of operation and predicate symbols can sometimes be left implicit too, and can imply fitting of sorts.*

---

spec LIST_ORDER_POSITIVE = LIST_ORDER [POSITIVE]

➤ *The intended fitting of the parameter symbols to the argument symbols may have to be specified explicitly.*

---

spec NAT_WORD_2 =
    GENERIC_MONOID [NATURAL_SUBSORTS **fit** $Elem \mapsto Nat$]

➤ *A generic specification may have more than one parameter.*

---

spec PAIR [**sort** *Elem1*] [**sort** *Elem2*] =
  free type $Pair ::= pair(first : Elem1; \ second : Elem2)$

spec TABLE [**sort** *Key*] [**sort** *Val*] = ...

Note that writing:

spec HOMOGENEOUS_PAIR_1 [**sort** *Elem*] [**sort** *Elem*] =
  free type $Pair ::= pair(first : Elem; \ second : Elem)$

merely defines pairs of values of the same sort, and HOMOGENEOUS_PAIR_1 is (equivalent to and) better defined as follows:

spec HOMOGENEOUS_PAIR [**sort** *Elem*] =
  free type $Pair ::= pair(first : Elem; \ second : Elem)$

➤ *Instantiation of generic specifications with several parameters is similar to the case of just one parameter.*

---

spec PAIR_NATURAL_COLOR =
  PAIR [NATURAL_ARITHMETIC] [COLOR **fit** $Elem2 \mapsto RGB$]

Using the specification PAIR_1 (similar to PAIR, but with one single parameter introducing two sorts *Elem1* and *Elem2*), would require us to write:

spec PAIR_NATURAL_COLOR_1 =
  PAIR_1 [NATURAL_ARITHMETIC **and** COLOR
      fit $Elem1 \mapsto Nat,\ Elem2 \mapsto RGB$]

➤ *When parameters are trivial, one can always avoid explicit fitting maps.*

---

spec PAIR_NATURAL_COLOR_2 =
PAIR [**sort** Nat] [**sort** RGB]
and NATURAL_ARITHMETIC **and** COLOR

Compare for instance:

spec PAIR_POS =
HOMOGENEOUS_PAIR [**sort** Pos] **and** INTEGER_ARITHMETIC_1

with:

spec PAIR_POS_1 =
HOMOGENEOUS_PAIR [INTEGER_ARITHMETIC_1 **fit** $Elem \mapsto Pos$]

Note that the instantiation:
HOMOGENEOUS_PAIR_1 [NATURAL] [COLOR **fit** $Elem \mapsto RGB$]
is ill-formed, since it entails mapping the sort $Elem$ to both Nat and RGB.

➤ *It is easy to specialize a generic specification with several parameters using a "partial instantiation".*

---

spec MY_TABLE [**sort** Val] =
    TABLE [NATURAL_ARITHMETIC] [**sort** Val]

➤ *Composition of generic specifications is expressed using instantiation.*

---

spec SET_OF_LIST [**sort** *Elem*] =
    GENERATED_SET [LIST_SELECTORS [**sort** *Elem*] **fit** *Elem* $\mapsto$ *List*]

Note especially that the following specification:

spec MISTAKE [**sort** *Elem*] =
    GENERATED_SET [LIST_SELECTORS [**sort** *Elem*]]

does *not* provide sets of lists of elements.

spec SET_AND_LIST [**sort** *Elem*] =
    GENERATED_SET [**sort** *Elem*] **and** LIST_SELECTORS [**sort** *Elem*]

It may be worth mentioning that the following composition of generic specifications is ill-formed:

spec THIS_IS_STILL_WRONG =
    GENERIC_MONOID [GENERIC_MONOID [**sort** *Elem*]
                     fit $Elem \mapsto Monoid$]

➤ *Compound sorts introduced by a generic specification get automatically renamed on instantiation, which avoids name clashes.*

---

spec LIST_REV [**sort** *Elem*] =

free type $List[Elem] ::= empty \mid$
$cons(head :? Elem; tail :? List[Elem])$

ops $\_\_ ++ \_\_ : List[Elem] \times List[Elem] \to List[Elem],$
$assoc, \; unit\ empty;$

$reverse : List[Elem] \to List[Elem]$

$\forall e : Elem; \; L, L1, L2 : List[Elem]$

- $cons(e, L1) ++ L2 = cons(e, L1 ++ L2)$
- $reverse(empty) = empty$
- $reverse(cons(e, L)) = reverse(L) ++ cons(e, empty)$

end

spec LIST_REV_NAT = LIST_REV [NATURAL]

spec TWO_LISTS =
LIST_REV [NATURAL] %% Provides the sort $List[Nat]$
and LIST_REV [COLOR **fit** $Elem \mapsto RGB$] %% Provides the sort $List[RGB]$

spec TWO_LISTS_1 =
LIST_REV [INTEGER_ARITHMETIC_1 **fit** $Elem \mapsto Nat$]
and LIST_REV [INTEGER_ARITHMETIC_1 **fit** $Elem \mapsto Int$]

Remember that $Nat < Int$ does not entail $List[Nat] < List[Int]$.

spec MONOID_C [**sort** *Elem*] =

  sort $Monoid[Elem]$

  ops $inj \quad : Elem \rightarrow Monoid[Elem];$

  $1 \quad : Monoid[Elem];$

  $\_\_ * \_\_ : Monoid[Elem] \times Monoid[Elem] \rightarrow Monoid[Elem],$

  $assoc, \quad unit\ 1$

  $\forall x, y : Elem \bullet inj(x) = inj(y) \Rightarrow x = y$

end

spec MONOID_OF_MONOID [**sort** *Elem*] =

  MONOID_C [MONOID_C [**sort** *Elem*] **fit** $Elem \mapsto Monoid[Elem]$]

➤ *Compound symbols can also be used for operations and predicates.*

---

spec LIST_REV_ORDER [TOTAL_ORDER] =
LIST_REV [**sort** *Elem*]
then local **op** $insert : Elem \times List[Elem] \to List[Elem]$
$\forall e, e' : Elem;\ L : List[Elem]$

- $insert(e, empty) = cons(e, empty)$
- $insert(e, cons(e', L)) = cons(e', insert(e, L))$ *when* $e' < e$
  else $cons(e, cons(e', L))$

within **op** $order[\_\_ < \_\_] : List[Elem] \to List[Elem]$
$\forall e : Elem;\ L : List[Elem]$

- $order[\_\_ < \_\_](empty) = empty$
- $order[\_\_ < \_\_](cons(e, L)) = insert(e, order[\_\_ < \_\_](L))$

end

spec LIST_REV_WITH_TWO_ORDERS =

LIST_REV_ORDER

[INTEGER_ARITHMETIC_ORDER **fit** $Elem \mapsto Int, \_\_ < \_\_ \mapsto \_\_ < \_\_$]

%% Provides the sort $List[Int]$ and the operation $order[\_\_ < \_\_]$

and LIST_REV_ORDER

[INTEGER_ARITHMETIC_ORDER **fit** $Elem \mapsto Int, \_\_ < \_\_ \mapsto \_\_ > \_\_$]

%% Provides the sort $List[Int]$ and the operation $order[\_\_ > \_\_]$

then %implies

$\forall L : List[Int] \bullet order[\_\_ < \_\_](L) = reverse(order[\_\_ > \_\_](L))$

end

➤ *Parameters should be distinguished from references to fixed specifications that are not intended to be instantiated.*

---

spec LIST_WEIGHTED_ELEM [**sort** $Elem$ **op** $weight : Elem \rightarrow Nat$]
    given NATURAL_ARITHMETIC =
  LIST_REV [**sort** $Elem$]
then **op** $weight : List[Elem] \rightarrow Nat$
  $\forall e : Elem;\ L : List[Elem]$
  • $weight(empty) = 0$
  • $weight(cons(e, L)) = weight(e) + weight(L)$
end

One could have written instead:

spec LIST_WEIGHTED_ELEM
  [NATURAL_ARITHMETIC **then sort** *Elem* **op** *weight* : $Elem \to Nat$] = ...

but the latter, which is correct, misses the essential distinction between the part which is intended to be specialized and the part which is 'fixed' (since, by definition, the parameter is the part which has to be specialized).

Note also that omitting the '**given** NATURAL_ARITHMETIC' clause would make the declaration:

spec LIST_WEIGHTED_ELEM [**sort** *Elem* **op** *weight* : $Elem \to Nat$] = ...

ill-formed, since the sort *Nat* is not available.

➤ *Argument specifications are always implicitly regarded as extension of the imports.*

---

spec LIST_WEIGHTED_PAIR_NATURAL_COLOR =

LIST_WEIGHTED_ELEM [PAIR_NATURAL_COLOR **fit** $Elem \mapsto Pair$, $weight \mapsto first$]

spec LIST_WEIGHTED_INSTANTIATED =

LIST_WEIGHTED_ELEM [**sort** $Value$ **op** $weight : Value \rightarrow Nat$]

➤ *Imports are also useful to prevent ill-formed instantiations.*

---

spec LIST_LENGTH [**sort** *Elem*] **given** NATURAL_ARITHMETIC =
    LIST_REV [**sort** *Elem*]
then **op** $length : List[Elem] \to Nat$
    $\forall e : Elem; \ L : List[Elem]$
    • $length(empty) = 0$
    • $length(cons(e, L)) = length(L) + 1$
then %implies
    $\forall L : List[Elem] \bullet length(reverse(L)) = length(L)$
end

spec LIST_LENGTH_NATURAL =
    LIST_LENGTH [NATURAL_ARITHMETIC]

spec WRONG_LIST_LENGTH [**sort** *Elem*] =
NATURAL_ARITHMETIC **and** LIST_REV [**sort** *Elem*]
then . . .
end

The specification WRONG_LIST_LENGTH is fine as long as one does not need to instantiate it with NATURAL_ARITHMETIC as argument specification.

The instantiation WRONG_LIST_LENGTH [NATURAL_ARITHMETIC] is ill-formed since some symbols of the argument specification are shared with some symbols of the body (and not already occurring in the parameter) of the instantiated generic specification, which is wrong. Of course the same problem will occur with any argument specification which provides, e.g., the sort *Nat*.

➤ *In generic specifications, auxiliary required specifications should be imported rather than extended.*

Since an instantiation is ill-formed as soon as there are some shared symbols between the argument specification and the body of the generic specification, when designing a generic specification, it is generally advisable to turn auxiliary required specifications into imports, and generic specifications of the form:

$F\,[\,X\,] = SP$ **then** $\ldots$

are better written

$F\,[\,X\,]$ **given** $SP = \ldots$

to allow the instantiation $F\,[\,SP\,]$.

➤ *Views are named fitting maps, and can be defined along with specifications.*

---

view INTEGER_AS_TOTAL_ORDER :
TOTAL_ORDER **to** INTEGER_ARITHMETIC_ORDER =
$Elem \mapsto Int,\ \_\_ < \_\_ \mapsto \_\_ < \_\_$

view INTEGER_AS_REVERSE_TOTAL_ORDER :
TOTAL_ORDER **to** INTEGER_ARITHMETIC_ORDER =
$Elem \mapsto Int,\ \_\_ < \_\_ \mapsto \_\_ > \_\_$

spec LIST_REV_WITH_TWO_ORDERS_1 =
LIST_REV_ORDER [**view** INTEGER_AS_TOTAL_ORDER]
and LIST_REV_ORDER [**view** INTEGER_AS_REVERSE_TOTAL_ORDER]
then %implies
$\forall L : List[Int] \bullet\ order[\_\_ < \_\_](L) = reverse(order[\_\_ > \_\_](L))$
end

➤ *Views can also be generic.*

---

view LIST_AS_MONOID [**sort** *Elem*] :

MONOID **to** LIST_REV [**sort** *Elem*] =

$Monoid \mapsto List[Elem],\ 1 \mapsto empty,\ \_\_ * \_\_ \mapsto \_\_ {+}{+} \_\_$

Specifying the Architecture of Implementations

➤ *Architectural specifications impose structure on implementations, whereas specification-building operations only structure the text of specifications.*

➤ *The examples in this chapter are artificially simple.*

---

spec COLOR = ...
spec NATURAL_ORDER = ...
spec NATURAL_ARITHMETIC = ...
spec ELEM = **sort** $Elem$
spec CONT [ELEM] =
  generated type $Cont[Elem] ::= empty \mid insert(Elem; Cont[Elem])$
  preds $\_\_is\_empty : Cont[Elem];$
    $\_\_is\_in\_\_ : Elem \times Cont[Elem]$
  ops $choose : Cont[Elem] \rightarrow ?\ Elem;$
    $delete : Elem \times Cont[Elem] \rightarrow Cont[Elem]$
  $\forall e, e' : Elem; C : Cont[Elem]$

- $empty\ is\_empty$
- $\neg\ insert(e, C)\ is\_empty$
- $\neg\ e\ is\_in\ empty$
- $e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
- $def\ choose(C) \Leftrightarrow \neg\ C\ is\_empty$
- $def\ choose(C) \Rightarrow choose(C)\ is\_in\ C$
- $e\ is\_in\ delete(e', C) \Leftrightarrow (e\ is\_in\ C \wedge \neg(e = e'))$

end

spec CONT_DIFF [ELEM] =
CONT [ELEM]
then **op** $\mathit{diff} : \mathit{Cont}[\mathit{Elem}] \times \mathit{Cont}[\mathit{Elem}] \to \mathit{Cont}[\mathit{Elem}]$
$\forall e : \mathit{Elem};\ C, C' : \mathit{Cont}[\mathit{Elem}]$
• $e\ \mathit{is\_in}\ \mathit{diff}(C, C') \Leftrightarrow (e\ \mathit{is\_in}\ C \wedge \neg(e\ \mathit{is\_in}\ C'))$
end

spec REQ = CONT_DIFF [NATURAL_ORDER]

spec FLAT_REQ =
  free type $Nat ::= 0 \mid suc(Nat)$
  pred $\_\_ < \_\_ : Nat \times Nat$
  generated type $Cont[Nat] ::= empty \mid insert(Nat; Cont[Nat])$
  preds $\_\_is\_empty : Cont[Nat];$
    $\_\_is\_in\_\_ : Nat \times Cont[Nat]$
  ops $choose : Cont[Nat] \to ? Nat;$
    $delete : Nat \times Cont[Nat] \to Cont[Nat];$
    $diff : Cont[Nat] \times Cont[Nat] \to Cont[Nat]$
  $\forall e, e' : Nat; C, C' : Cont[Nat]$
  - $0 < suc(e)$
  - $\neg(e < 0)$
  - $suc(e) < suc(e') \Leftrightarrow e < e'$
  - $empty\ is\_empty$
  - $\neg\ insert(e, C)\ is\_empty$
  - $\neg\ e\ is\_in\ empty$
  - $e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
  - $def\ choose(C) \Leftrightarrow \neg\ C\ is\_empty$
  - $def\ choose(C) \Rightarrow choose(C)\ is\_in\ C$
  - $e\ is\_in\ delete(e', C) \Leftrightarrow (e\ is\_in\ C \wedge \neg(e = e'))$
  - $e\ is\_in\ diff(C, C') \Leftrightarrow (e\ is\_in\ C \wedge \neg(e\ is\_in\ C'))$

end

➤ *An architectural specification consists of a list of unit declarations, specifying the required components, and a result part, indicating how they are to be combined.*

---

arch spec SYSTEM =
units N : NATURAL_ORDER;
C : CONT [NATURAL_ORDER] **given** N;
D : CONT_DIFF [NATURAL_ORDER] **given** C
result D

➤ *There can be several distinct architectural choices for the same requirements specification.*

---

arch spec SYSTEM_1 =

units N : NATURAL_ORDER;

CD : CONT_DIFF [NATURAL_ORDER] **given** N

result CD

➤ *Each unit declaration listed in an architectural specification corresponds to a separate implementation task.*

In the architectural specification SYSTEM, the task of providing a component D expanding C and implementing CONT_DIFF [NATURAL_ORDER] is independent from the tasks of providing implementations N of NATURAL_ORDER and C of CONT [NATURAL_ORDER] given N.

Hence, when providing the component D, one cannot make any further assumption on how the component C is (or will be) implemented, besides what is expressly ensured by its specification.

Thus the component D should expand *any* given implementation C of CONT [NATURAL_ORDER] and provide an implementation of CONT_DIFF [NATURAL_ORDER], which is tantamount to providing a *generic* implementation G of CONT_DIFF [NATURAL_ORDER] which takes the particular implementation of CONT [NATURAL_ORDER] as a parameter to be expanded. Then we obtain D by simply applying G to C.

Genericity here arises from the independence of the developments of C and D, rather than from the desire to build multiple implementations of CONT_DIFF [NATURAL_ORDER] using different implementations of CONT [NATURAL_ORDER].

➤ *A unit can be implemented only if its specification is a conservative extension of the specifications of its given units.*

For instance, the component D can exist only if the specification CONT_DIFF [NATURAL_ORDER] is a conservative extension of CONT [NATURAL_ORDER].

```
spec CONT_DIFF_1 =
     CONT [NATURAL_ORDER]
then op diff : Cont[Nat] × Cont[Nat] → Cont[Nat]
     ∀x, y : Nat; C, C′ : Cont[Nat]
      • diff(C, empty) = C
      • diff(empty, C′) = empty
      • diff(insert(x, C), insert(y, C′)) =
            insert(x, diff(C, insert(y, C′))) when x < y
            else diff(C, C′) when x = y
            else diff(insert(x, C), C′)
      • x is_in diff(C, C′) ⇔ (x is_in C ∧ ¬(x is_in C′))
end
```

```
arch spec INCONSISTENT =
units  N  :  NATURAL_ORDER;
       C  :  CONT [NATURAL_ORDER] given N;
       D  :  CONT_DIFF_1 given C
result D
```

➤ *Genericity of components can be made explicit in architectural specifications.*

---

arch spec SYSTEM_G =

units N : NATURAL_ORDER;

F : NATURAL_ORDER → CONT [NATURAL_ORDER];

G : CONT [NATURAL_ORDER] → CONT_DIFF [NATURAL_ORDER]

result $G\,[F\,[N]]$

➤ *A generic component may be applied to an argument richer than required by its specification.*

---

arch spec SYSTEM_A =

units NA : NATURAL_ARITHMETIC;

F : NATURAL_ORDER → CONT [NATURAL_ORDER];

G : CONT [NATURAL_ORDER] → CONT_DIFF [NATURAL_ORDER]

result $G\,[F\,[NA]]$

➤ *Specifications of components can be named for further reuse.*

---

unit spec CONT_COMP = ELEM → CONT [ELEM]

unit spec DIFF_COMP = CONT [ELEM] → CONT_DIFF [ELEM]

arch spec SYSTEM_G1 =

units N : NATURAL_ORDER;

F : CONT_COMP;

G : DIFF_COMP

result $G\,[F\,[N]]$

➤ *Both named and un-named specifications can be used to specify components.*

---

unit spec DIFF_COMP_1 =

$$\textsc{Cont}\,[\textsc{Elem}] \rightarrow \{\ \textbf{op}\ \mathit{diff} : \mathit{Cont}[\mathit{Elem}] \times \mathit{Cont}[\mathit{Elem}] \rightarrow \mathit{Cont}[\mathit{Elem}]$$

$$\forall e : \mathit{Elem};\ C, C' : \mathit{Cont}[\mathit{Elem}]$$

$$\bullet\ e\ \mathit{is\_in}\ \mathit{diff}(C, C') \Leftrightarrow$$

$$(e\ \mathit{is\_in}\ C \wedge \neg(e\ \mathit{is\_in}\ C'))\ \}$$

➤ *Specifications of generic components should not be confused with generic specifications.*

- Generic specifications naturally give rise to specifications of generic components, which can be named for later reuse, as illustrated above by CONT_COMP.
- A generic specification is nothing other than a piece of specification that can easily be adapted by instantiation.
- A specification of a generic component cannot be instantiated, it is the specified *generic component* which gets *applied* to suitable components.

➤ *A generic component may be applied more than once in the same architectural specification.*

---

arch spec OTHER_SYSTEM =

units N : NATURAL_ORDER;

C : COLOR;

F : CONT_COMP

result $F\,[N]$ **and** $F\,[C$ **fit** $Elem \mapsto RGB]$

➤ *Several applications of the same generic component is different from applications of several generic components with similar specifications.*

---

arch spec OTHER_SYSTEM_1 =
units N : NATURAL_ORDER;
C : COLOR;
FN : NATURAL_ORDER → CONT [NATURAL_ORDER];
FC : COLOR → CONT [COLOR **fit** $Elem \mapsto RGB$]
result $FN[N]$ **and** $FC[C]$

➤ *Generic components may have more than one argument.*

---

unit spec SET_COMP = ELEM → GENERATED_SET [ELEM]

spec CONT2SET [ELEM] =
  CONT [ELEM] **and** GENERATED_SET [ELEM]
then **op** $elements\_of\_\_ : Cont[Elem] \rightarrow Set$
  $\forall e : Elem;\ C : Cont[Elem]$
  - $elements\_of\ \ empty = empty$
  - $elements\_of\ \ insert(e, C) = \{e\} \cup elements\_of\ \ C$

end

arch spec ARCH_CONT2SET_NAT =
units N : NATURAL_ORDER;
  C : CONT_COMP;
  S : SET_COMP;
  F : CONT [ELEM] × GENERATED_SET [ELEM] → CONT2SET [ELEM]
result $F\ [C\ [N]]\ [S\ [N]]$

➤ *Open systems can be described by architectural specifications using generic unit expressions in the result part.*

---

arch spec ARCH_CONT2SET =

units C : CONT_COMP;

S : SET_COMP;

F : CONT [ELEM] × GENERATED_SET [ELEM] → CONT2SET [ELEM]

result λX : ELEM • $F\,[C\,[X]]\,[S\,[X]]$

arch spec ARCH_CONT2SET_USED =

units N : NATURAL_ORDER;

CSF : **arch spec** ARCH_CONT2SET

result $CSF\,[N]$

➤ *When components are to be combined, it is best to check that any shared symbol originates from the same non-generic component.*

---

arch spec ARCH_CONT2SET_NAT_1 =

units N : NATURAL_ORDER;

C : CONT_COMP;

S : SET_COMP;

G : { CONT [ELEM] **and** GENERATED_SET [ELEM] }

→ CONT2SET [ELEM]

result $G\,[C\,[N]$ **and** $S\,[N]$ **fit** $Cont[Elem] \mapsto Cont[Nat]\,]$

arch spec WRONG_ARCH_SPEC =

units CN : CONT [NATURAL_ORDER];

SN : GENERATED_SET [NATURAL_ORDER];

F : CONT [ELEM] × GENERATED_SET [ELEM] → CONT2SET [ELEM]

result $F\ [CN]\ [SN]$

arch spec BADLY_STRUCTURED_ARCH_SPEC =

units N : NATURAL_ORDER;

A : NATURAL_ORDER → NATURAL_ARITHMETIC;

C : CONT_COMP;

S : SET_COMP;

F : CONT [ELEM] × GENERATED_SET [ELEM] → CONT2SET [ELEM]

result $F\ [C\ [A\ [N]]]\ [S\ [A\ [N]]]$

➤ *Auxiliary unit definitions or local unit definitions may be used to avoid repetition of generic unit applications.*

---

arch spec WELL_STRUCTURED_ARCH_SPEC =

units N : NATURAL_ORDER;

A : NATURAL_ORDER → NATURAL_ARITHMETIC;

AN = $A\,[\,N\,]$;

C : CONT_COMP;

S : SET_COMP;

F : CONT [ELEM] × GENERATED_SET [ELEM] → CONT2SET [ELEM]

result $F\ [C\ [AN]]\ [S\ [AN]]$

arch spec ANOTHER_WELL_STRUCTURED_ARCH_SPEC =

units N : NATURAL_ORDER;

A : NATURAL_ORDER $\rightarrow$ NATURAL_ARITHMETIC;

C : CONT_COMP;

S : SET_COMP;

F : CONT [ELEM] $\times$ GENERATED_SET [ELEM] $\rightarrow$ CONT2SET [ELEM]

result **local** $AN = A\,[N]$ **within** $F\,[C\,[AN]]\,[S\,[AN]]$

Libraries

➤ *Libraries are named collections of named specifications.*

➤ *Local libraries are self-contained.*

A library is called *local* when it is self-contained, i.e., for each reference to a specification name in the library, the library includes a specification with that name.

➤ *Distributed libraries support reuse.*

Distributed libraries allow duplication of specifications to be avoided altogether.

Instead of making an explicit copy of a named specification from one library for use in another, the second library merely indicates that the specification concerned can be *downloaded* from the first one.

➤ *Different versions of the same library are distinguished by hierarchical version numbers.*

➤ *Local libraries are self-contained collections of specifications.*

---

library USERMANUAL/EXAMPLES

...

spec NATURAL = ...

...

spec NATURAL_ORDER = NATURAL **then** ...

...

➤ *Specifications can refer to previous items in the same library.*

---

library USERMANUAL/EXAMPLES

...

spec STRICT_PARTIAL_ORDER = ...

...

spec TOTAL_ORDER = STRICT_PARTIAL_ORDER **then** ...

...

spec PARTIAL_ORDER = STRICT_PARTIAL_ORDER **then** ...

...

➤ *All kinds of named specifications can be included in libraries.*

---

library USERMANUAL/EXAMPLES

...

spec STRICT_PARTIAL_ORDER = ...

...

spec GENERIC_MONOID [**sort** *Elem*] = ...

...

view INTEGER_AS_TOTAL_ORDER : ...

...

view LIST_AS_MONOID [**sort** *Elem*] : ...

...

arch spec SYSTEM = ...

...

unit spec CONT_COMP = ...

...

➤ *Display, parsing, and literal syntax annotations apply to entire libraries.*

---

library USERMANUAL/EXAMPLES

...

%display __<=__ %LATEX $\_\_ \leq \_\_$

%display __>=__ %LATEX $\_\_ \geq \_\_$

%display __union__ %LATEX $\_\_ \cup \_\_$

%prec $\{\_\_+\_\_, \_\_-\_\_\} < \{\_\_ * \_\_\}$

%left_assoc $\_\_+\_\_, \_\_ * \_\_$

...

spec STRICT_PARTIAL_ORDER $= \ldots$

...

spec PARTIAL_ORDER $=$ STRICT_PARTIAL_ORDER **then** $\ldots \leq \ldots$

...

spec GENERATED_SET [**sort** *Elem*] $= \ldots \cup \ldots$

...

spec INTEGER_ARITHMETIC_ORDER $= \ldots \leq \ldots \geq \ldots$

...

Parsing annotations allow omission of grouping parentheses when terms are input. A single annotation can indicate the relative precedence or the associativity (left or right) of a group of operation symbols. The precedence annotation for infix arithmetic operations given above, namely:

%prec $\{\_\_+\_\_, \_\_-\_\_\} < \{\_\_*\_\_\}$

allows a term such as $a + (b * c)$ to be input (and hence also displayed) as $a + b * c$. The left-associativity annotation for $+$ and $*$:

%left_assoc $\_\_+\_\_, \_\_*\_\_$

allows $(a + b) + c$ to be input as $a + b + c$, and similarly for $*$; but the parentheses cannot be omitted in $(a + b) - c$ (not even if '$\_\_-\_\_$' were to be included in the same left-associativity annotation).

When an operation symbol is declared with the associativity attribute $assoc$, an associativity *annotation* for that symbol is provided automatically.

➤ *Libraries and library items can have author and date annotations.*

---

library USERMANUAL/EXAMPLES

%authors(Michel Bidoit ⟨`bidoit@lsv.ens-cachan.fr`⟩,
Peter D. Mosses ⟨`pdmosses@brics.dk`⟩ **)%**

%dates 15 Oct 2003, 1 Apr 2000

. . .

spec STRICT_PARTIAL_ORDER = . . .

. . .

%authors Michel Bidoit ⟨`bidoit@lsv.ens-cachan.fr`⟩

%dates 10 July 2003

spec INTEGER_ARITHMETIC_ORDER =

. . .

➤ *Libraries can be installed on the Internet for remote access. Validated libraries can be registered for public access.*

---

library BASIC/NUMBERS

...

%left_assoc $\_\_@@\_\_$

%number $\_\_@@\_\_$

%floating $\_\_:::\_\_\,,\_\_E\_\_$

%prec $\{\_\_E\_\_\} < \{\_\_:::\_\_\}$

...

spec NAT =

free type $Nat ::= 0 \mid suc(Nat)$

...

ops $1 : Nat = suc(0); \ldots; 9 : Nat = suc(8);$

$\_\_@@\_\_(m, n : Nat) : Nat = (m * suc(9)) + n$

...

spec INT = NAT **then** ...

spec RAT = INT **then** ...

spec DECIMALFRACTION = RAT **then**

...

ops $\_\_:::\_\_ : Nat \times Nat \rightarrow Rat;$

$\_\_E\_\_ : Rat \times Int \rightarrow Rat$

...

➤ *Libraries should include appropriate annotations.*

➤ *Libraries can include items downloaded from other libraries.*

---

library BASIC/STRUCTUREDDATATYPES

...

from BASIC/NUMBERS **get** NAT, INT

...

spec LIST [**sort** *Elem*] **given** NAT = ...

...

spec ARRAY ... **given** INT = ...

...

from BASIC/NUMBERS **get** NAT $\mapsto$ NATURAL, INT $\mapsto$ INTEGER

➤ *Substantial libraries of basic datatypes are already available.*

BASIC/NUMBERS: natural numbers, integers, and rationals.

BASIC/RELATIONSANDORDERS: reflexive, symmetric, and transitive relations, equivalence relations, partial and total orders, boolean algebras.

BASIC/ALGEBRA_I: monoids, groups, rings, integral domains, and fields.

BASIC/SIMPLEDATATYPES: booleans, characters.

BASIC/STRUCTUREDDATATYPES: sets, lists, strings, maps, bags, arrays, trees.

BASIC/GRAPHS: directed graphs, paths, reachability, connectedness, colorability, and planarity.

BASIC/ALGEBRA_II: monoid and group actions on a space, euclidean and factorial rings, polynomials, free monoids, and free commutative monoids.

BASIC/LINEARALGEBRA_I: vector spaces, bases, and matrices.

BASIC/LINEARALGEBRA_II: algebras over a field.

BASIC/MACHINENUMBERS: bounded subtypes of naturals and integers.

➤ *Libraries need not be registered for public access.*

---

library `http://www.cofi.info/CASL/Test/Security`

...

from `http://casl:password@www.cofi.info/CASL/RSA` **get** KEY

...

spec DECRYPT = KEY **then** ...

...

➤ *Subsequent versions of a library are distinguished by explicit version numbers.*

---

library BASIC/NUMBERS **version** 1.0

...

spec NAT = ...

...

spec INT = NAT **then** ...

...

spec RAT = INT **then** ...

...

➤ *Libraries can refer to specific versions of other libraries.*

---

library BASIC/STRUCTUREDDATATYPES **version** 1.0

. . .

from BASIC/NUMBERS **version** 1.0 **get** NAT, INT

. . .

spec LIST [**sort** *Elem*] **given** NAT = . . .

. . .

spec ARRAY . . . **given** INT = . . .

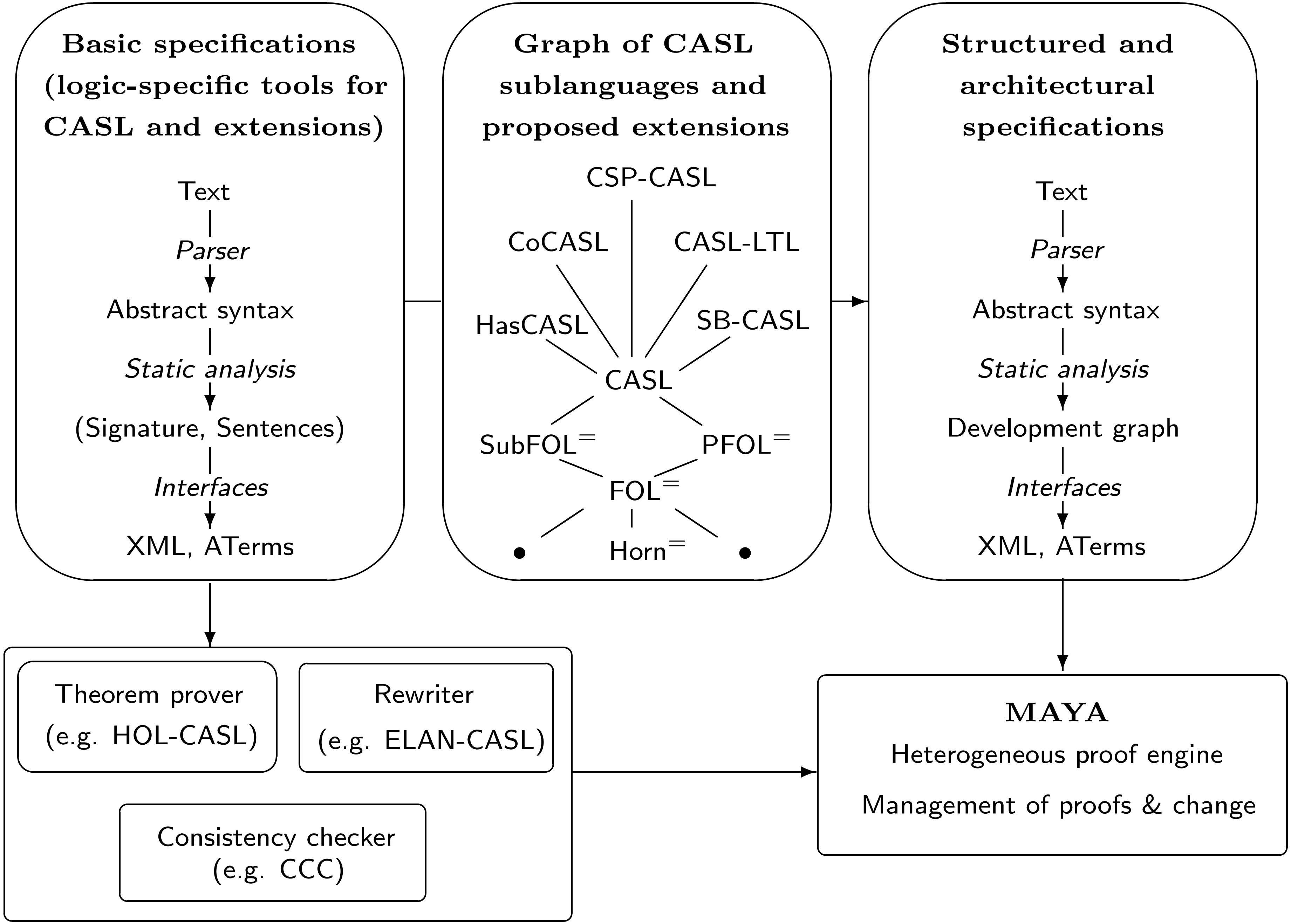
. . .

➤ *All downloadings should be collected at the beginning of a library.*

Tools

➤ *The Heterogeneous Tool Set (HETS) is the main analysis tool for CASL.*

- CASL specifications can also be checked for well-formedness using a form-based web page.
- HETS can be used for parsing and checking static well-formedness of specifications.
- HETS also displays and manages proof obligations, using development graphs.
- Nodes in a development graph correspond to CASL specifications.
  Arrows show how specifications are related by the structuring constructs.
- Internal nodes in a development graph correspond to unnamed parts of a structured specification.
- HOL-CASL is an interactive theorem prover for CASL, based on the tactical theorem prover ISABELLE.
- CASL is linked to ISABELLE/HOL by an encoding.
- ASF+SDF was used to prototype the CASL syntax.
- The ASF+SDF Meta-Environment provides syntax-directed editing of CASL specifications.



Architecture of the heterogeneous tool set.