A Gentle Introduction to CASL

Michel Bidoit
LSV, CNRS & ENS de Cachan, France
CoFI Methodology Coordinator

Peter D. Mosses
BRICS & Dept. of Computer Science, Univ. of Aarhus, Denmark
CoFI External Relations Coordinator

April 11, 2000

Contents

Introduction

CoFI

➤ CoFI *is an initiative to provide a common framework for algebraic specification and development.*

Casl

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

About This Tutorial

➤ *This tutorial illustrates and discusses how to write* Casl *specifications.*

Underlying Concepts

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

Total, Many-Sorted Specifications

- *Total, many-sorted specifications in* CASL *may be written 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 =
  sort $Elem$;
  pred $\_\_ < \_\_ : Elem \times Elem$;
  vars $x, y, z : Elem$;
  axioms
    $\neg(x < x)$;
    $x < y \Rightarrow \neg(y < x)$;
    $x < y \wedge y < z \Rightarrow x < z$;
  %% 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 **vars** $x, y : Elem$;
  axiom $x < y \vee y < x \vee x = y$;
  ops $min(x, y : Elem) = x \text{ when } x < y \text{ else } y$;
    $max(x, y : Elem) = y \text{ when } min(x, y) = x \text{ else } x$;
end

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 supposed to be consequences of others.*

---

spec $\textsc{Partial\_Order\_2} =$
$\textsc{Partial\_Order}$
then **%implies**
axiom $\forall x, y, z : \mathit{Elem} \bullet x \leq y \wedge y \leq z \Rightarrow x \leq z;$
end

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

---

spec $\textsc{Monoid} =$
sort $\mathit{Monoid};$
ops $\mathit{1} : \mathit{Monoid};$
$\_\_ * \_\_ : \mathit{Monoid} \times \mathit{Monoid} \rightarrow \mathit{Monoid}, \mathit{assoc}, \mathit{unit}\ \mathit{1};$
end

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

---

spec GENERIC_MONOID [**sort** $Elem$] =
  sort $Monoid$;
  ops $inj : Elem \to Monoid$;
  $1 : Monoid$;
  $\_\_ * \_\_ : Monoid \times Monoid \to Monoid, assoc, unit\ 1$;
  axiom $\forall x, y : Elem \bullet inj(x) = inj(y) \Rightarrow x = y$;
end

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

---

spec COM_GENERIC_MONOID_1 [**sort** $Elem$] =
  GENERIC_MONOID [**sort** $Elem$]
then **axiom** $\forall x, y : Monoid \bullet x * y = y * x$;
end

spec COM_GENERIC_MONOID_2 [**sort** $Elem$] =
  GENERIC_MONOID [**sort** $Elem$]
then **op** $\_\_ * \_\_ : Monoid \times Monoid \to Monoid, comm$;
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 $\_\_is\_in\_\_ : Elem \times Container$;
  vars $e, e' : Elem; C : Container$;
  axioms
    $\neg(e\ is\_in\ empty)$;
    $e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$;
end

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

---

spec CONTAINER_2 [**sort** $Elem$] =
  CONTAINER [**sort** $Elem$]
then **type** $Container ::= mark\_insert(Elem; Container)$;
  pred $\_\_is\_marked\_in\_\_ : Elem \times Container$;
  vars $e, e' : Elem; C : Container$;
  axioms
    $e\ is\_in\ mark\_insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$;
    $\neg(e\ is\_marked\_in\ empty)$;
    $e\ is\_marked\_in\ insert(e', C) \Leftrightarrow e\ is\_marked\_in\ C$;
    $e\ is\_marked\_in\ mark\_insert(e', C) \Leftrightarrow$
      $(e = e' \vee e\ is\_marked\_in\ C)$;
end

Generated Specifications

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

---

spec $\textsc{Generated\_Container}$ [**sort** $Elem$] $=$

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

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

vars $e, e' : Elem; C : Container$;

axioms

$\neg(e\ is\_in\ empty)$;

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

end

➤ *Generated specifications are in general loose.*

---

spec $\textsc{Generated\_Container\_2}$ [**sort** $Elem$] =
$\textsc{Generated\_Container}$ [**sort** $Elem$]
then **op** $\_\_merge\_\_ : Container \times Container \to Container;$
vars $e : Elem; C, C' : Container;$
axiom $e\ is\_in\ (C\ merge\ C') \Leftrightarrow (e\ is\_in\ C \vee e\ is\_in\ C');$
end

➤ *Generated specifications need not be loose.*

---

spec $\textsc{Generated\_Set}$ [**sort** $Elem$] =
generated type $Set ::= empty \mid insert(Elem; Set);$
pred $\_\_is\_in\_\_ : Elem \times Set;$
ops $\{\_\_\}(e : Elem) : Set = insert(e, empty);$
$\_\_ \cup \_\_ : Set \times Set \to Set;$
$remove : Elem \times Set \to Set;$
vars $e, e' : Elem; S, S' : Set$
• $\neg(e\ is\_in\ empty)$
• $e\ is\_in\ insert(e', S) \Leftrightarrow (e = e' \vee e\ is\_in\ S)$
• $S = S' \Leftrightarrow (\forall x : Elem \bullet x\ is\_in\ S \Leftrightarrow x\ is\_in\ S')$
• $e\ is\_in\ (S \cup S') \Leftrightarrow (e\ is\_in\ S \vee e\ is\_in\ S')$
• $e\ is\_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is\_in\ S)$
then %implies
generated type $Set ::= empty \mid \{\_\_\}(Elem) \mid \_\_ \cup \_\_(Set; Set);$
op $\_\_ \cup \_\_ : Set \times Set \to Set, assoc, comm, idem, unit\ empty;$
vars $e, e' : Elem; S : Set$
• $insert(e, insert(e, S)) = insert(e, S)$
• $insert(e, insert(e', S)) = insert(e', insert(e, S))$
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 $Tree ::= mktree(Node; Forest)$;
$Forest ::= empty \mid add(Tree; Forest)$;

Free Specifications

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

---

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

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

---

spec COLOR =
 free type $RGB ::= Red \mid Green \mid Blue$;
 free type $Printer\_Color ::= Cyan \mid Magenta \mid Yellow \mid Black$;
end

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

---

spec $\textsc{Integer} =$
    free $\{$ **type** $Int ::= 0 \mid suc(Int) \mid pre(Int);$
        vars $x, y : Int$
        • $suc(pre(x)) = x$
        • $pre(suc(x)) = x$ $\}$
end

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

---

spec $\textsc{Natural\_2} =$
    $\textsc{Natural}$
then **free** $\{$ **pred** $\_\_ < \_\_ : Nat \times Nat;$
        vars $x, y : Nat$
        • $0 < suc(x)$
        • $x < y \Rightarrow suc(x) < suc(y)$ $\}$
end

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

---

spec $\text{NATURAL\_3} =$
NATURAL_2
then ops $1 : Nat = suc(0);$
$\_\_ + \_\_ : Nat \times Nat \to Nat, assoc, comm, unit\ 0;$
$\_\_ * \_\_ : Nat \times Nat \to Nat, assoc, comm, unit\ 1;$
vars $x, y : Nat;$
axioms
$x + suc(y) = suc(x + y);$
$x * 0 = 0;$
$x * suc(y) = x + (x * y);$
end

➤ *More care may be needed when defining operations on free datatypes with axioms relating the constructors.*

---

spec $\text{INTEGER\_2} =$
INTEGER
then ops $1 : Int = suc(0);$
$\_\_ + \_\_ : Int \times Int \to Int, assoc, comm, unit\ 0;$
$\_\_ - \_\_ : Int \times Int \to Int;$
$\_\_ * \_\_ : Int \times Int \to Int, assoc, comm, unit\ 1;$
vars $x, y : Int;$
axioms
$x + suc(y) = suc(x + y);$
$x + pre(y) = pre(x + y);$
$x - 0 = x;$
$x - suc(y) = pre(x - y);$
$x - pre(y) = suc(x - y);$
$x * 0 = 0;$
$x * suc(y) = (x * y) + x;$
$x * pre(y) = (x * y) - x;$
end

spec $\textsc{Integer\_3}$ =
$\textsc{Integer\_2}$
then **preds** $\_\_ \leq \_\_, \_\_ \geq \_\_ : \mathit{Int} \times \mathit{Int};$
vars $x, y : \mathit{Int};$
axioms
$0 \leq 0;$
$\neg(0 \leq \mathit{pre}(0));$
$0 \leq x \Rightarrow 0 \leq \mathit{suc}(x);$
$\neg(0 \leq x) \Rightarrow \neg(0 \leq \mathit{pre}(x));$
$\mathit{suc}(x) \leq y \Leftrightarrow x \leq \mathit{pre}(y);$
$\mathit{pre}(x) \leq y \Leftrightarrow x \leq \mathit{suc}(y);$
$x \geq y \Leftrightarrow y \leq x;$
end

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

---

spec $\textsc{List}$ [**sort** Elem] =
free type $\mathit{List} ::= \mathit{empty} \mid \mathit{cons}(\mathit{Elem}; \mathit{List});$
end

spec $\textsc{Set}$ [**sort** $Elem$] =

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

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

vars $e, e' : Elem; S : Set$;

axioms

$insert(e, insert(e, S)) = insert(e, S)$;

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

$\neg(e\ is\_in\ empty)$;

$e\ is\_in\ insert(e, S)$;

$e\ is\_in\ insert(e', S)\ if\ e\ is\_in\ S$; }

end

spec $\textsc{Transitive\_Closure}$ [**sort** $Elem$; **pred** $\_\_R\_\_ : Elem \times Elem$] =

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

vars $x, y, z : Elem$

• $x\ R\ y \Rightarrow x\ R^{+} y$

• $x\ R^{+} y \wedge y\ R^{+} z \Rightarrow x\ R^{+} z$ }

end

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

---

spec NATURAL_4 =
NATURAL_3
then **op** $max\_size : Nat$; **axiom** $0 < max\_size$ **end**

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

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

---

spec SET_G [**sort** $Elem$] =
generated type $Set ::= empty \mid insert(Elem; Set)$;
pred $\_\_is\_in\_\_ : Elem \times Set$;
vars $e, e' : Elem; S, S' : Set$;
axioms
$\neg(e\ is\_in\ empty)$;
$e\ is\_in\ insert(e', S) \Leftrightarrow (e = e' \vee e\ is\_in\ S)$;
$S = S' \Leftrightarrow (\forall x : Elem \bullet x\ is\_in\ S \Leftrightarrow x\ is\_in\ S')$;
end

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

---

spec $\textsc{Set\_Ind}$ [**sort** $Elem$] =
$\textsc{Set}$ [**sort** $Elem$]
then **%define**
ops $\_\_ \cup \_\_ : Set \times Set \to Set, assoc, comm, idem, unit\ empty;$
$remove : Elem \times Set \to Set;$
vars $e, e' : Elem; S, S' : Set;$
axioms
$S \cup insert(e', S') = insert(e', S \cup S');$
$remove(e, empty) = empty;$
$remove(e, insert(e, S)) = remove(e, S);$
$remove(e, insert(e', S)) =$
$\quad insert(e', remove(e, S))\ if\ \neg(e = e');$
end

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

---

spec $\textsc{Set\_Obs}$ [**sort** $Elem$] =
$\textsc{Set\_G}$ [**sort** $Elem$]
then **%define**
ops $\_\_ \cup \_\_ : Set \times Set \to Set, assoc, comm, idem, unit\ empty;$
$remove : Elem \times Set \to Set;$
vars $e, e' : Elem; S, S' : Set;$
axioms
$e\ is\_in\ (S \cup S') \Leftrightarrow (e\ is\_in\ S \vee e\ is\_in\ S');$
$e\ is\_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is\_in\ S);$
end

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

---

spec NUMBER =
  free { **type** $Number ::= 0 \mid suc(Number)$;
    op $\_\_ + \_\_ : Number \times Number \to Number$,
      $assoc, comm, unit\ 0$;
    vars $x, y : Number$;
    axioms
      $x + suc(y) = suc(x + y)$;
      $\forall x : Number \bullet \exists y : Number \bullet x + y = 0$; }
end

Partial Functions

➤ *Partial functions arise naturally. They are declared differently from total functions.*

---

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

➤ *Terms containing partial functions may be undefined, i.e., they may not denote any value.*

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

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

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

➤ *Predicates applied to an undefined term yield false.*

If the term $choose(S)$ is undefined, then $choose(S)\ is\_in\ S$ yields false.

➤ *Equations hold also when both terms are undefined.*

The ordinary equation

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

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 $\textsc{Set\_with\_P\_Choose\_1}$ [**sort** $Elem$] $=$
$\textsc{Set\_with\_P\_Choose}$ [**sort** $Elem$]
then **axioms** $\neg(def\ choose(empty))$;
$\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$;
end

We know that $choose$ is undefined on $empty$, 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 $\textsc{Set\_with\_P\_Choose\_2}$ [**sort** $Elem$] $=$
$\textsc{Set\_with\_P\_Choose}$ [**sort** $Elem$]
then **axioms** $\forall S : Set \bullet def\ choose(S) \Leftrightarrow \neg(S = empty)$;
$\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$;
end

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

---

spec NATURAL_WITH_BOUNDS =
NATURAL_4
then **op** $\_\_+?\_\_ : Nat \times Nat \to ? Nat$;
vars $x, y : Nat$
• $def(x +? y)$ *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_WITH_P_CHOOSE_3 [**sort** *Elem*] =
SET_WITH_P_CHOOSE [**sort** *Elem*]
then **axioms** $\neg(def\ choose(empty))$;
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$;
end

We can conclude after some reasoning that
$def\ choose(S) \Leftrightarrow \neg(S = empty)$, but this is not so prominent.

spec NATURAL_5 =
NATURAL_3
then **op** $pre : Nat \to? Nat;$
vars $x, y : Nat$
• $\neg\, def\ pre(0)$
• $pre(suc(x)) = x$
end

is explicit enough.

spec NATURAL_6 =
NATURAL_5
then **op** $\_\_ - \_\_ : Nat \times Nat \to? Nat;$
vars $x, y : Nat$
• $x - 0 = x$
• $x - suc(y) = pre(x - y)$
end

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

spec NATURAL_7 =
NATURAL_5
then **op** $\_\_ - \_\_ : Nat \times Nat \to? Nat;$
vars $x, y : Nat$
• $def(x - y) \Leftrightarrow (y < x \vee y = x)$
• $x - 0 = x$
• $x - suc(y) = pre(x - y)$
end

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

---

spec $\textsc{List\_2}$ [**sort** $Elem$] $=$
$\textsc{List}$ [**sort** $Elem$]
then free { **ops** $hd : List \to? Elem;$
$tail : List \to? List;$
vars $e : Elem; L : List$
• $hd(cons(e, L)) = e$
• $tail(cons(e, L)) = L$ }
end

spec $\textsc{List\_3}$ [**sort** $Elem$] $=$
$\textsc{List}$ [**sort** $Elem$]
then ops $hd : List \to? Elem;$
$tail : List \to? List;$
vars $e : Elem; L : List$
• $\neg\, def\ hd(nil)$
• $\neg\, def\ tail(nil)$
• $hd(cons(e, L)) = e$
• $tail(cons(e, L)) = L$
end

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

---

spec LIST_4 [**sort** $Elem$] =
  free type $List ::= nil \mid cons(hd\ :?\ Elem; tail\ :?\ List)$;
end

spec NATURAL_8 =
  free type $Nat ::= 0 \mid suc(pre\ :?\ Nat)$;
  ...
end

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

---

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

➤ *Constructors also may be partial.*

---

spec $\textsc{Part\_Container}$ [**sort** $Elem$] =
generated type
$P\_Container ::= empty \mid insert(Elem; P\_Container)?;$
pred $addable : Elem \times P\_Container;$
vars $e, e' : Elem; C : P\_Container;$
axiom $def\ insert(e, C) \Leftrightarrow addable(e, C);$
pred $\_\_is\_in\_\_ : Elem \times P\_Container;$
axioms
$\neg(e\ is\_in\ empty);$
$(e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C))\ if\ addable(e', C);$
end

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

---

spec $\textsc{Natural\_9}$ =
$\textsc{Natural\_7}$
then **vars** $x, y : Nat$
%% $y - x = z - x \Rightarrow y = z$ would be wrong,
%% $def(y - x) \wedge def(z - x) \wedge y - x = z - x \Rightarrow y = z$
%% is correct, but can be better abbreviated into:
• $y - x \stackrel{e}{=} z - x \Rightarrow y = z$
end

Subsorting

➤ *Sorts may denote related domains, which can be expressed using subsorting.*

---

spec GENERIC_MONOID_2 [**sort** $Elem$] =
  sorts $Elem < Monoid$;
  ops $1 : Monoid$;
    $\_\_ * \_\_ : Monoid \times Monoid \rightarrow Monoid, assoc, unit\ 1$;
end

➤ *A sort may be essentially the union of 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

➤ *Operations declared on some sort are automatically inherited by its subsorts, even those declared later on.*

---

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

➤ *Subsort membership can be checked or asserted.*

---

spec SPEED_REGULATION =
VEHICLE
then **op** $speed\_limit : Vehicle \to Nat;$
var $v : Vehicle$
• $v \in Car \Rightarrow speed\_limit(v) = 130$
• $v \in Bicycle \Rightarrow speed\_limit(v) = 30$
end

➤ *Datatype declarations can involve subsort declarations.*

---

type $Vehicle ::= sort\ Car \mid sort\ Bicycle \mid sort\ Boat;$
leaves the way open to further kinds of vehicles (e.g., planes).

generated type $Vehicle ::= sort\ Car \mid sort\ Bicycle \mid sort\ Boat;$
prevents the definition of further subsorts, e.g., for planes.

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 NAT_WITH_EVEN_AND_PRIME =
NATURAL_3
then **pred** $even : Nat;$
var $n : Nat;$
axioms
$even(0);$
$\neg\, even(1);$
$even(suc(suc(n))) \Leftrightarrow even(n);$
sort $Even = \{x : Nat \bullet even(x)\};$
sort $Prime = \{x : Nat \bullet \forall\, y, z : Nat \bullet x = y * z \Rightarrow y = 1 \vee z = 1\};$
end

spec POS = NATURAL_5 **then sort** $Pos = \{x : Nat \bullet \neg(x = 0)\}$ **end**

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

---

spec POS_2 =
POS
then **ops** $1 : Pos;$
$suc : Nat \to Pos;$
$\_\_ + \_\_, \_\_ * \_\_ : Pos \times Pos \to Pos;$
$\_\_ + \_\_ : Pos \times Nat \to Pos;$
$\_\_ + \_\_ : Nat \times Pos \to Pos;$
end

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

---

spec $\textsc{Pos\_3} = \textsc{Pos\_2}$ **then op** $pre : Pos \rightarrow Nat$ **end**

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

---

spec $\textsc{Pos\_And\_Nat} =$

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

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

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

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

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

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

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

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

vars $x, y : Nat;$

axioms

$x + suc(y) = suc(x + y);$

$x * 0 = 0;$

$x * suc(y) = x + (x * y);$

end

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

$pre(\ pre(suc(1))\ as\ Pos\ )$

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

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

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

---

spec INTEGER_4 =
POS_AND_NAT
then **free type** $Int ::= sort\ Nat \mid -\_\_(Pos)$;
ops $\_\_ + \_\_ : Int \times Int \to Int, assoc, comm, unit\ 0$;
$\_\_ - \_\_ : Int \times Int \to Int$;
$\_\_ * \_\_ : Int \times Int \to Int, assoc, comm, unit\ 1$;
$|\_\_| : Int \to Nat$;
vars $x, y : Int; n : Nat; p, q : Pos$
• $suc(n) + (-1) = n$
• $suc(n) + (-suc(q)) = n + (-q)$
• $(-p) + (-q) = -(p + q)$
• $x - 0 = x$
• $x - p = x + (-p)$
• $x - (-q) = x + q$
• $n * (-q) = -(n * q)$
• $(-p) * (-q) = p * q$
• $|n| = n$
• $| - p| = p$
end

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

---

spec SET_WITH_ERROR_CHOOSE_1 [**sort** $Elem$] =
GENERATED_SET [**sort** $Elem$]
then **sorts** $Elem < ElemError$;
op $choose : Set \to ElemError$;
pred $\_\_is\_in\_\_ : ElemError \times Set$;
axiom $\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S) \in Elem \wedge$
$choose(S)\ is\_in\ S$;
end

spec SET_WITH_ERROR_CHOOSE_2 [**sort** $Elem$] =
GENERATED_SET [**sort** $Elem$]
then **sorts** $Elem < ElemError$;
op $choose : Set \to ElemError$;
axiom $\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S) \in Elem \wedge$
$(choose(S)\ as\ Elem)\ is\_in\ S$;
end

Structuring Specifications

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

---

spec LIST_TO_SET [**sort** *Elem*] =
LIST_4 [**sort** *Elem*] **and** GENERATED_SET [**sort** *Elem*]
then **op** $elements\_of\_\_ : List \to Set$;
vars $e : Elem; L : List$;
axioms
$elements\_of\ nil = empty$;
$elements\_of\ cons(e, L) = \{e\} \cup elements\_of\ L$;
end

➤ *Arbitrary parts of specifications can have initial semantics.*

---

spec CHOOSE_IN_LIST [**sort** *Elem*] =
LIST_4 [**sort** *Elem*] **and** SET_WITH_P_CHOOSE_2 [**sort** *Elem*]
then **ops** $elements\_of\_\_ : List \to Set$;
$choose : List \to ?\ Elem$;
vars $e : Elem; L : List$
• $elements\_of\ nil = empty$
• $elements\_of\ cons(e, L) = \{e\} \cup elements\_of\ L$
• $def\ choose(L) \Leftrightarrow \neg(L = nil)$
• $choose(L) = choose(elements\_of\ L)$
end

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

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

---

spec $\textsc{Stack}$ [**sort** $Elem$] $=$
$\textsc{List\_4}$ [**sort** $Elem$] **with sort** $List \mapsto Stack,$
ops $cons \mapsto push\_\_onto\_\_,$
$hd \mapsto top,$
$tail \mapsto pop$
end

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

---

spec $\textsc{List\_to\_Set\_2}$ [**sort** $Elem$] $=$
$\textsc{List\_4}$ [**sort** $Elem$] **with** $nil, cons$
and $\textsc{Generated\_Set}$ [**sort** $Elem$] **with** $empty, \{\_\_\}, \_\_ \cup \_\_$
then **op** $elements\_of\_\_ : List \to Set;$
vars $e : Elem; L : List$
• $elements\_of\ \ nil = empty$
• $elements\_of\ \ cons(e, L) = \{e\} \cup elements\_of\ \ L$
end

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

---

spec $\textsc{Natural\_10} = \textsc{Natural\_7}$ **hide** suc, pre **end**

spec $\textsc{Natural\_11} =$
$\textsc{Natural\_7}$ **reveal** $Nat, 0, 1, \_\_+\_\_, \_\_-\_\_, \_\_*\_\_, \_\_<\_\_$
end

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

---

spec $\textsc{List\_with\_Order}$ [$\textsc{Partial\_Order}$ **with sort** $Elem$, **pred** $\_\_\leq\_\_$] =
$\textsc{List\_4}$ [**sort** $Elem$]
then **local** **op** $insert : Elem \times List \to List;$
vars $e, e' : Elem; L : List$
• $insert(e, nil) = cons(e, nil)$
• $insert(e, cons(e', L)) = cons(e, cons(e', L))$ *when* $e \leq e'$
else $cons(e', insert(e, L))$
within **op** $order : List \to List;$
vars $e : Elem; L : List$
• $order(nil) = nil$
• $order(cons(e, L)) = insert(e, order(L))$
end

spec $\textsc{List\_with\_Order\_2}$ [$\textsc{Partial\_Order}$ **with sort** *Elem*, **pred** $\_\_ \leq \_\_$] =
$\textsc{List\_4}$ [**sort** *Elem*]

then **local** **preds** $\_\_is\_sorted : List;$
$\_\_is\_in\_\_ : Elem \times List;$
$same\_elements : List \times List;$

vars $e, e' : Elem; L, L' : List$

- $nil\ is\_sorted$
- $cons(e, nil)\ is\_sorted$
- $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow (e \leq e' \wedge cons(e', L)\ is\_sorted)$
- $\neg(e\ is\_in\ nil)$
- $e\ is\_in\ cons(e', L) \Leftrightarrow (e = e' \vee e\ is\_in\ L)$
- $same\_elements(L, L') \Leftrightarrow (\forall e : Elem \bullet e\ is\_in\ L \Leftrightarrow e\ is\_in\ L')$

within **op** $order : List \rightarrow List;$

axiom $\forall L : List \bullet order(L)\ is\_sorted \wedge same\_elements(L, order(L));$

end

Generic Specifications

➤ *Explicit parameters show the intended genericity of a specification.*

➤ *Parameters are arbitrary specifications.*

spec $\textsc{List\_4}$ [**sort** $Elem$] = ...

spec $\textsc{List\_with\_Order}$ [$\textsc{Partial\_Order}$ **with sort** $Elem$, **pred** $\_\_ \leq \_\_$] = ...

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

spec $\textsc{Pair}$ [**sort** $Elem1$] [**sort** $Elem2$] = ...

spec $\textsc{Index}$ =
$\textsc{Total\_Order}$ **with** $Elem \mapsto Index$ **hide** min, max
then **ops** $first, last : Index$; **axiom** $\neg(last < first)$; **end**
spec $\textsc{Array}$ [$\textsc{Index}$] [**sort** $Elem$] =
type $Array ::= Init \mid \_\_[\_\_]{:=}\_\_(Array; Index; Elem)?$;
op $\_\_[\_\_] : Array \times Index \to ? \; Elem$;
then
local **pred** $\_\_is\_valid : Index$;
var $i : Index$
• $i \; is\_valid \Leftrightarrow \neg(i < first) \wedge \neg(last < i)$
within **vars** $T : Array; i, j : Index; e, e' : Elem$
• $def \; T[i]{:=}e \Leftrightarrow i \; is\_valid$
• $def \; T[i] \Leftrightarrow (i \; is\_valid \wedge$
$\exists \, T' : Array; e : Elem \bullet T = T'[i]{:=}e)$
• $def \; T[i]{:=}e \Rightarrow (T[i]{:=}e)[i] = e$
• $\neg(i = j) \wedge def \; T[i]{:=}e \wedge def \; T[j] \Rightarrow$
$(T[i]{:=}e)[j] = T[j]$
• $def \; T[i]{:=}e \Rightarrow (T[i]{:=}e)[i]{:=}e' = T[i]{:=}e'$
• $\neg(i = j) \wedge def \; T[i]{:=}e \wedge def \; T[j]{:=}e' \Rightarrow$
$(T[i]{:=}e)[j]{:=}e' = (T[j]{:=}e')[i]{:=}e$
end

➤ *In instantiations, the intended fitting of the parameter symbols to the argument symbols has to be uniquely determined.*

---

spec $\textsc{List\_of\_Int} = \textsc{List\_4}\ [\textsc{Integer\_3}$ **fit** $Elem \mapsto Int]$ **end**

spec $\textsc{Pair\_of\_Nat} =$
$\textsc{Pair}\ [\textsc{Natural\_4}$ **fit** $Elem1 \mapsto Nat]\ [\textsc{Natural\_4}$ **fit** $Elem2 \mapsto Nat]$
end

Trivial instantiations can be concisely described by omitting the fitting symbol map.

Note that an alternative to $\textsc{Pair\_of\_Nat}$ is to write:

spec $\textsc{Pair\_of\_Nat\_2} = \textsc{Pair}$ [**sort** Nat] [**sort** Nat] **and** $\textsc{Natural\_4}$ **end**

➤ *Implied and identity mappings can be omitted.*

---

spec $\textsc{Array\_Indexed\_by\_Nat}$ [**sort** $Elem$] $=$
$\textsc{Array}\ [\textsc{Natural\_4}$ **fit** $Index \mapsto Nat, first \mapsto 0, last \mapsto max\_size]$
[**sort** $Elem$]
end

Note that there is no need to explicitly map $\_\_ < \_\_ \ldots$

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

---

spec SET_OF_LIST [**sort** $Elem$] =
    GENERATED_SET [LIST_4 [**sort** $Elem$] **fit** $Elem \mapsto List$]
end

➤ *Compound sorts introduced by a generic specification get automatically renamed when it is instantiated, which avoids name clashes.*

---

spec LIST_5 [**sort** $Elem$] =
    free type $List[Elem] ::= nil \mid cons(hd :? Elem; tail :? List[Elem])$;
    ops $\_\_ ++ \_\_ : List[Elem] \times List[Elem] \to List[Elem], assoc, unit\ nil$;
        $reverse : List[Elem] \to List[Elem]$;
    vars $e : Elem; L, L1, L2 : List[Elem]$;
    axioms
        $cons(e, L1) ++ L2 = cons(e, L1 ++ L2)$;
        $reverse(nil) = nil$;
        $reverse(cons(e, L)) = reverse(L) ++ cons(e, nil)$;
end

spec TWO_LISTS =
LIST_5 [NATURAL_3 **fit** $Elem \mapsto Nat$] **%%** Provides sort $List[Nat]$
and LIST_5 [INTEGER_3 **fit** $Elem \mapsto Int$] **%%** Provides sort $List[Int]$
end

spec TWO_LISTS_2 =
LIST_5 [INTEGER_4 **fit** $Elem \mapsto Nat$]
and LIST_5 [INTEGER_4 **fit** $Elem \mapsto Int$]
end

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

---

spec LIST_WITH_ORDER_3 [PARTIAL_ORDER **with sort** $Elem$, **pred** $\_\_ \leq \_\_$] =
LIST_5 [**sort** $Elem$]
then **local** **op** $insert : Elem \times List[Elem] \to List[Elem]$;
vars $e, e' : Elem; L : List[Elem]$
• $insert(e, nil) = cons(e, nil)$
• $insert(e, cons(e', L)) = cons(e, cons(e', L))$ *when* $e \leq e'$ *else* $cons(e', insert(e, L))$
within **op** $order[\_\_ \leq \_\_] : List[Elem] \to List[Elem]$;
vars $e : Elem; L : List[Elem]$
• $order[\_\_ \leq \_\_](nil) = nil$
• $order[\_\_ \leq \_\_](cons(e, L)) = insert(e, order[\_\_ \leq \_\_](L))$
end
spec REVERSE_ORDERS =
LIST_WITH_ORDER_3 [INTEGER_3 **fit** $Elem \mapsto Int, \_\_ \leq \_\_ \mapsto \_\_ \leq \_\_$]
and LIST_WITH_ORDER_3 [INTEGER_3 **fit** $Elem \mapsto Int, \_\_ \leq \_\_ \mapsto \_\_ \geq \_\_$]
then %implies
axiom $\forall L : List[Int] \bullet order[\_\_ \leq \_\_](L) = reverse(order[\_\_ \geq \_\_](L))$;
end

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

---

spec List_with_Length [**sort** $Elem$] **given** Natural_3 =
List_5 [**sort** $Elem$]
then **op** $length : List[Elem] \to Nat$;
vars $e : Elem; L : List[Elem]$
• $length(nil) = 0$
• $length(cons(e, L)) = length(L) + 1$
then %implies
axiom $\forall\, L : List[Elem] \bullet length(reverse(L)) = length(L)$;
end

spec List_of_Nat = List_with_Length [Natural_3 **fit** $Elem \mapsto Nat$]

spec Wrong_List_with_Length [**sort** $Elem$] =
Natural_3 **and** List_5 [**sort** $Elem$]
then ... **end**

would be inadequate since instantiation by Natural_3 would be ill-formed.

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

---

view Integer_as_Partial_Order_1 : Partial_Order **to** Integer_3 =
$Elem \mapsto Int, \_\_ \leq \_\_ \mapsto \_\_ \leq \_\_$

view Integer_as_Partial_Order_2 : Partial_Order **to** Integer_3 =
$Elem \mapsto Int, \_\_ \leq \_\_ \mapsto \_\_ \geq \_\_$

spec Reverse_Orders_2 =
List_with_Order_3 [**view** Integer_as_Partial_Order_1]
and List_with_Order_3 [**view** Integer_as_Partial_Order_2]
then %implies
axiom $\forall\, L : List[Int] \bullet order[\_\_ \leq \_\_](L) = reverse(order[\_\_ \geq \_\_](L))$;
end

➤ *Views can also be generic.*

Specifying Architectural Structure

Libraries