A Gentle Introduction to CASL

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

Peter D. Mosses
BRICS & University of Aarhus, Denmark
CoFI External Relations Coordinator

➤ *Copyright ©2000, 2001 Michel Bidoit and Peter D. Mosses, CoFI, The Common Framework Initiative for Algebraic Specification and Development.*

Permission is granted to anyone to make or distribute verbatim copies of this document, in any medium, provided that the copyright notice and permission notice are preserved, and that the distributor grants the recipient permission for further redistribution as permitted by this notice. Modified versions may not be made.

March 25, 2001

Contents

Introduction 4

Underlying Concepts 6

Total, Many-Sorted Specifications 7

- Loose Specifications 8
- Generated Specifications 17
- Free Specifications 22

Partial Functions 38

Subsorting 57

Structuring Specifications 70

Generic Specifications 78

Specifying Architectural Structure 90

Libraries 91

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 =

%% 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)%
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

%display __leq__ %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 supposed to be 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

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

---

spec MONOID =

sort $Monoid$

ops $1 : Monoid;$

$__ * __ : Monoid \times Monoid \to Monoid, assoc, unit\ 1$

end

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

---

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

sort $Monoid$

ops $inj : Elem \to Monoid;$

$1 : Monoid;$

$__ * __ : Monoid \times Monoid \to Monoid, assoc, unit\ 1$

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

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** $__merge__ : Container \times Container \to Container$
$\forall e : Elem;\ C, C' : Container$
• $e\ is_in\ (C\ merge\ C') \Leftrightarrow (e\ is_in\ C \vee e\ is_in\ C')$
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 $\{__\}(e : Elem) : Set = insert(e, empty);$

$__ \cup __ : Set \times Set \to Set;$

$remove : Elem \times Set \to Set$

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

- $\neg(e\ is_in\ empty)$
- $e\ is_in\ insert(e', S) \Leftrightarrow (e = e' \vee e\ is_in\ S)$
- $S = S' \Leftrightarrow (\forall x : Elem \bullet x\ is_in\ S \Leftrightarrow x\ is_in\ S')$
- $e\ is_in\ (S \cup S') \Leftrightarrow (e\ is_in\ S \vee e\ is_in\ S')$
- $e\ is_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is_in\ S)$

then %implies

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

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

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

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

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 INTEGER =

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

$\forall x, y : Int$

- $suc(pre(x)) = x$
- $pre(suc(x)) = x$ }

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

---

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

$\forall x, y : Nat$

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

end

➤ *More care may be needed when defining operations on free datatypes with 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** $__ \leq __, __ \geq __ : Int \times Int$

$\forall x, y : Int$

- $0 \leq 0$
- $\neg(0 \leq pre(0))$
- $0 \leq x \Rightarrow 0 \leq suc(x)$
- $\neg(0 \leq x) \Rightarrow \neg(0 \leq pre(x))$
- $suc(x) \leq y \Leftrightarrow x \leq pre(y)$
- $pre(x) \leq y \Leftrightarrow x \leq suc(y)$
- $x \geq y \Leftrightarrow y \leq x$

end

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

---

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

free type $List ::= empty \mid cons(Elem;\ List)$

end

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** $__R^+__ : Elem \times Elem$

$\forall x, y, z : Elem$

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

end

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

---

spec NATURAL_BOUNDED =
NATURAL_ARITHMETIC
then **op** $max_size : Nat$ **axiom** $0 < max_size$ **end**

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

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

---

spec SET_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 NUMBER =

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

op $__ + __ : Number \times Number \to Number,$
$assoc, comm, unit\ 0$

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

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

end

Partial Functions

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

---

spec SET_PARTIAL_CHOOSE [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **op** *choose* : *Set* →? *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

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

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 • $\neg(def\ choose(empty))$
$\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is_in\ S$
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 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_BOUNDED_ADDITION =

NATURAL_BOUNDED

then **op** $__+?__ : Nat \times Nat \to ? \; Nat$

$\forall x, y : Nat$

- $def(x+?y)$ *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** $pre : Nat \rightarrow ? \ Nat$

$\forall x, y : Nat$

- $\neg \ def \ pre(0)$
- $pre(suc(x)) = x$

end

is explicit enough.

spec NATURAL_PARTIAL_SUBTRACTION =

NATURAL_PARTIAL_PRE

then **op** $__ - __ : Nat \times Nat \to ? Nat$

$\forall x, y : Nat$

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

end

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

spec NATURAL_PARTIAL_SUBTRACTION_1 =

NATURAL_PARTIAL_PRE

then **op** $__ - __ : Nat \times Nat \to ? Nat$

$\forall x, y : Nat$

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

end

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

---

spec LIST_SELECTORS [**sort** $Elem$] =
LIST [**sort** $Elem$]
then **free** { **ops** $head : List \to ? \ Elem;$
$tail : List \to ? \ List$
$\forall e : Elem; \ L : List$
- $head(cons(e, L)) = e$
- $tail(cons(e, L)) = L$ }

end

spec LIST_SELECTORS_1 [**sort** $Elem$] =

LIST [**sort** $Elem$]

then **ops** $head : List \to ? \ Elem;$

$tail : List \to ? \ List$

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

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

end

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

---

spec LIST_SELECTORS_2 [**sort** $Elem$] =
  free type $List ::= nil \mid cons(head\ :?\ Elem;\ tail\ :?\ List)$
end

spec NATURAL_SUC_PRE =
  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 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 involves the definedness of both terms as well as their equality.*

---

spec NATURAL_PARTIAL_SUBTRACTION_2 =

NATURAL_PARTIAL_SUBTRACTION_1

then

$\forall x, y : Nat$

%{ $y - x = z - x \Rightarrow y = z$ 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_1 [**sort** $Elem$] =
  sorts $Elem < Monoid$
  ops $1 : Monoid$;
    $__ * __ : Monoid \times Monoid \to Monoid, assoc, unit\ 1$
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** $speed_limit : Vehicle \to Nat$

$\forall v : Vehicle$

- $v \in Car \Rightarrow speed_limit(v) = 130$
- $v \in Bicycle \Rightarrow speed_limit(v) = 30$

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 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 $Prime = \{x : Nat \bullet$
$\forall y, z : Nat \bullet x = y * z \Rightarrow y = 1 \vee z = 1\}$
end

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

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

---

spec PERSITIVE_ARITHMETIC =
PERSITIVE
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 POSITIVE_PRE = POSITIVE_ARITHMETIC **then**

op $pre : Pos \to Nat$

end

➤ *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 $1 : Pos = suc(0);$

$__ + __ : Nat \times Nat \to Nat, assoc, comm, unit\ 0;$

$__ * __ : Nat \times Nat \to Nat, assoc, comm, unit\ 1;$

$__ + __, __ * __ : Pos \times Pos \to Pos;$

$__ + __ : Pos \times Nat \to Pos;$

$__ + __ : Nat \times Pos \to Pos$

$\forall x, y : Nat$

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

end

- *Casting a term from a supersort to a subsort is explicit and 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_ARITHMETIC_1 =

NATURAL_POSITIVE_ARITHMETIC

then **free type** $Int ::= sort\ Nat \mid -__(Pos)$

ops $__ + __ : Int \times Int \to Int, assoc, comm, unit\ 0;$

$__ - __ : Int \times Int \to Int;$

$__ * __ : Int \times Int \to Int, assoc, comm, unit\ 1;$

$|__| : Int \to Nat$

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

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

end

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

---

spec SET_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 $choose : Set \to ElemError$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S) \in Elem \wedge$
$(choose(S)\ as\ Elem)\ is_in\ S$
end

Structuring Specifications

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

---

spec LIST_SET [**sort** *Elem*] =
LIST_SELECTORS_2 [**sort** *Elem*] **and**
GENERATED_SET [**sort** *Elem*]
then **op** $elements_of__ : List \to Set$
$\forall e : Elem;\ L : List$
- $elements_of\ nil = empty$
- $elements_of\ cons(e, L) = \{e\} \cup elements_of\ L$

end

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

---

spec LIST_CHOOSE [**sort** $Elem$] =
LIST_SELECTORS_2 [**sort** $Elem$] **and** SET_PARTIAL_CHOOSE_2 [**sort** $Elem$]
then **ops** $elements_of__ : List \to Set$;
$choose : List \to ? Elem$
$\forall e : Elem;\ L : List$
- $elements_of\ nil = empty$
- $elements_of\ cons(e, L) = \{e\} \cup elements_of\ L$
- $def\ choose(L) \Leftrightarrow \neg(L = nil)$
- $choose(L) = choose(elements_of\ L)$

end

spec SET_TO_LIST [**sort** $Elem$] =
LIST_SET [**sort** $Elem$]
then **op** $list_of__ : Set \to List$
$\forall S : Set \bullet elements_of(list_of\ S) = S$
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_2 [**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_2 [**sort** *Elem*] **with** *nil*, *cons*

and GENERATED_SET [**sort** *Elem*] **with** *empty*, {_}, _ ∪ _

then **op** *elements_of* _ : *List* → *Set*

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

- $elements_of\ nil = empty$
- $elements_of\ cons(e, L) = \{e\} \cup elements_of\ L$

end

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

---

spec 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

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

---

spec LIST_ORDER [PARTIAL_ORDER **with sort** $Elem$, **pred** $__ \leq __$] =
LIST_SELECTORS_2 [**sort** $Elem$]
then local **op** $insert : Elem \times List \to List$
$\forall e, e' : Elem;\ L : List$
- $insert(e, nil) = cons(e, nil)$
- $insert(e, cons(e', L)) = cons(e, cons(e', L))\ when\ e \leq e'$
$else\ cons(e', insert(e, L))$

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

end

spec LIST_ORDER_SORTED [PARTIAL_ORDER **with sort** $Elem$, **pred** $__ \leq __$] =
LIST_SELECTORS_2 [**sort** $Elem$]

then **local** **preds** $__is_sorted$: $List$;
$__is_in__$: $Elem \times List$;
$same_elements$: $List \times List$

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

- $nil\ is_sorted$
- $cons(e, nil)\ is_sorted$
- $cons(e, cons(e', L))\ is_sorted \Leftrightarrow$
  $(e \leq e' \wedge cons(e', L)\ is_sorted)$
- $\neg(e\ is_in\ nil)$
- $e\ is_in\ cons(e', L) \Leftrightarrow (e = e' \vee e\ is_in\ L)$
- $same_elements(L, L') \Leftrightarrow$
  $(\forall e : Elem \bullet e\ is_in\ L \Leftrightarrow e\ is_in\ L')$

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

$\forall L : List \bullet order(L)\ is_sorted \wedge$
$same_elements(L, order(L))$

end

Generic Specifications

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

- *Parameters are arbitrary specifications.*

spec LIST_SELECTORS_2 [**sort** *Elem*] = ...

spec LIST_ORDER [PARTIAL_ORDER **with sort** *Elem*, **pred** __ ≤ __] = ...

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

spec PAIR [**sort** *Elem1*] [**sort** *Elem2*] = ...

spec INDEX =

TOTAL_ORDER **with** $Elem \mapsto Index$ **hide** min, max

then **ops** $first, last : Index$

- $\neg(last < first)$

end

spec ARRAY [INDEX] [**sort** $Elem$] =

type $Array ::= Init \mid __[__]:=__(Array;\ Index;\ Elem)?$

op $__[__] : Array \times Index \rightarrow ?\ Elem$

then **local** **pred** $__is_valid : Index$

$\forall i : Index$

- $i\ is_valid \Leftrightarrow \neg(i < first) \wedge \neg(last < i)$

within $\forall T : Array;\ i, j : Index;\ e, e' : Elem$

- $def\ T[i]:=e \Leftrightarrow i\ is_valid$
- $def\ T[i] \Leftrightarrow (i\ is_valid \wedge \exists T' : Array;\ e : Elem \bullet T = T'[i]:=e)$
- $def\ T[i]:=e \Rightarrow (T[i]:=e)[i] = e$
- $\neg(i = j) \wedge def\ T[i]:=e \wedge def\ T[j] \Rightarrow (T[i]:=e)[j] = T[j]$
- $def\ T[i]:=e \Rightarrow (T[i]:=e)[i]:=e' = T[i]:=e'$
- $\neg(i = j) \wedge def\ T[i]:=e \wedge def\ T[j]:=e' \Rightarrow (T[i]:=e)[j]:=e' = (T[j]:=e')[i]:=e$

end

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

---

spec LIST_INTEGER =
LIST_SELECTORS_2 [INTEGER_ARITHMETIC_ORDER **fit** *Elem* ↦ *Int*]
end

spec PAIR_NATURAL =
PAIR [NATURAL_BOUNDED **fit** *Elem1* ↦ *Nat*]
[NATURAL_BOUNDED **fit** *Elem2* ↦ *Nat*]
end

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

Note that an alternative to PAIR_NATURAL is to write:

spec PAIR_NATURAL_1 = PAIR [**sort** *Nat*] [**sort** *Nat*] **and** NATURAL_BOUNDED **end**

➤ *Implied and identity mappings can be omitted.*

---

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

ARRAY [NATURAL_BOUNDED

fit $Index \mapsto Nat, first \mapsto 0, last \mapsto max_size$]

[**sort** $Elem$]

end

Note that there is no need to explicitly map __ < __ ...

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

---

spec SET_LIST [**sort** *Elem*] =
GENERATED_SET
[LIST_SELECTORS_2 [**sort** *Elem*] **fit** *Elem* ↦ *List*]
end

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

---

spec LIST_1 **[sort** $Elem$**]** =

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

ops $__ ++ __ : List[Elem] \times List[Elem] \to List[Elem]$, $assoc, unit\ nil$;

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

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

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

end

```
spec  TWO_LISTS =
      LIST_1 [NATURAL_ARITHMETIC fit Elem ↦ Nat]
          %% Provides sort List[Nat]
and   LIST_1 [INTEGER_ARITHMETIC_ORDER fit Elem ↦ Int]
          %% Provides sort List[Int]
end

spec  TWO_LISTS_1 =
      LIST_1 [INTEGER_ARITHMETIC_1 fit Elem ↦ Nat]
and   LIST_1 [INTEGER_ARITHMETIC_1 fit Elem ↦ Int]
end
```

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

---

spec LIST_ORDER_SORTED_1 [PARTIAL_ORDER **with sort** $Elem$, **pred** $__ \leq __$] =
LIST_1 [**sort** $Elem$]
then **local** **op** $insert : Elem \times List[Elem] \to List[Elem]$
$\forall e, e' : Elem;\ L : List[Elem]$
- $insert(e, nil) = cons(e, nil)$
- $insert(e, cons(e', L)) = cons(e, cons(e', L))\ when\ e \leq e'$
$else\ cons(e', insert(e, L))$

within **op** $order[__ \leq __] : List[Elem] \to List[Elem]$
$\forall e : Elem;\ L : List[Elem]$
- $order[__ \leq __](nil) = nil$
- $order[__ \leq __](cons(e, L)) = insert(e, order[__ \leq __](L))$

end
spec LIST_REVERSE_ORDERS =
LIST_ORDER_SORTED_1
[INTEGER_ARITHMETIC_ORDER **fit** $Elem \mapsto Int, __ \leq __ \mapsto __ \leq __$]
and LIST_ORDER_SORTED_1
[INTEGER_ARITHMETIC_ORDER **fit** $Elem \mapsto Int, __ \leq __ \mapsto __ \geq __$]
then %implies
$\forall L : List[Int] \bullet order[__ \leq __](L) = reverse(order[__ \geq __](L))$
end

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

---

spec LIST_LENGTH [**sort** $Elem$] **given** NATURAL_ARITHMETIC =
LIST_1 [**sort** $Elem$]
then **op** $length : List[Elem] \to Nat$
$\forall e : Elem;\ L : List[Elem]$
- $length(nil) = 0$
- $length(cons(e, L)) = length(L) + 1$

then **%implies**
$\forall L : List[Elem] \bullet length(reverse(L)) = length(L)$
end

spec LIST_NATURAL = LIST_LENGTH [NATURAL_ARITHMETIC **fit** $Elem \mapsto Nat$]

spec WRONG_LIST_LENGTH [**sort** $Elem$] =
NATURAL_ARITHMETIC **and** LIST_1 [**sort** $Elem$]
then . . . **end**

would be inadequate since instantiation by NATURAL_ARITHMETIC is ill-formed.

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

---

view IntegerAsPartialOrder1 :
PartialOrder **to** IntegerArithmeticOrder =
$Elem \mapsto Int, __ \leq __ \mapsto __ \leq __$

view IntegerAsPartialOrder2 :
PartialOrder **to** IntegerArithmeticOrder =
$Elem \mapsto Int, __ \leq __ \mapsto __ \geq __$

spec ListReverseOrders1 =
ListOrderSorted1 **[view** IntegerAsPartialOrder1**]**
and ListOrderSorted1 **[view** IntegerAsPartialOrder2**]**
then %implies $\forall L : List[Int] \bullet order[__ \leq __](L) = reverse(order[__ \geq __](L))$
end

- *Views can also be generic.*

Specifying Architectural Structure

Libraries