A Gentle Introduction to CASL

Michel Bidoit
LSV, CNRS & ENS de Cachan, France

Peter D. Mosses
BRICS & University of Aarhus, Denmark

www.cofi.info
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.

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.
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 × Elem % pred abbreviates predicate
∀x, y, z : Elem
• ¬(x < x) % (strict)%
• x < y ⇒ ¬(y < x) % (asymmetric)%
• x < y ∧ y < z ⇒ 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.

```casl
spec TOTAL_Order =
    STRICT_Partial_Order
then \forall x, y : Elem \land x < y \lor y < x \lor x = y
end
```

© Michel Bidoit, Peter D. Mosses, CoFI
In simple cases, an operation (or a predicate) symbol may be declared and its intended interpretation defined at the same time.

\[
\text{spec } \text{TOTAL\_ORDER\_WITH\_MINMAX} = \\
\text{TOTAL\_ORDER} \\
\text{then ops } \min(x, y : \text{Elem}) : \text{Elem} = x \text{ when } x < y \text{ else } y; \\
\quad \max(x, y : \text{Elem}) : \text{Elem} = y \text{ when } \min(x, y) = x \text{ else } x \\
\text{end}
\]
spec **Variant_Of_Total_Order_With_MinMax** = **Total_Order**

then vars \( x, y : Elem \)

\[ \begin{align*}
\text{op} & \quad \text{min} : Elem \times Elem \rightarrow Elem \\
\text{•} & \quad x < y \Rightarrow \text{min}(x, y) = x \\
\text{•} & \quad \neg(x < y) \Rightarrow \text{min}(x, y) = y \\
\text{op} & \quad \text{max} : Elem \times Elem \rightarrow Elem \\
\text{•} & \quad x < y \Rightarrow \text{max}(x, y) = y \\
\text{•} & \quad \neg(x < y) \Rightarrow \text{max}(x, y) = x
\end{align*} \]
Symbols may be conveniently displayed as usual mathematical symbols by means of %display annotations.

\[
\text{spec } \text{PARTIAL\_ORDER} = \\
\text{strict PARTIAL\_ORDER}
\]

\[
\text{then pred } \_\leq\_ (x, y : Elem) \iff (x < y \lor x = y)
\]

\[
\text{end}
\]
The %implies annotation is used to indicate that some axioms are supposedly redundant, being consequences of others.

```casl
spec Partial_Order_1 =

  Partial_Order

then %implies

  \forall x, y, z : Elem \cdot x \leq y \land y \leq z \Rightarrow x \leq z \quad \%(transitive)%

end


  Partial_Order

then %implies

  \forall x, y : Elem \cdot x < y \lor y < x \lor x = y \quad \%(total)%

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

```plaintext
spec Monoid =
    sort Monoid
    ops 1 : Monoid;
        ___*___ : Monoid × Monoid → Monoid, assoc, unit 1
end
```
Genericity of specifications can be made explicit using parameters.

\[
\text{spec \, \textsc{Generic-Monoid} [ sort \, Elem ] = }
\]

\[
\begin{align*}
\text{sort} & \quad \text{Monoid} \\
\text{ops} & \quad \text{inj} : Elem \rightarrow \text{Monoid}; \\
& \quad 1 : \text{Monoid}; \\
& \quad \_ * \_ : \text{Monoid} \times \text{Monoid} \rightarrow \text{Monoid}, \text{assoc}, \text{unit} \, 1 \\
\forall x, y : Elem \quad \bullet \quad \text{inj}(x) = \text{inj}(y) \Rightarrow x = y \\
\end{align*}
\]

\textbf{end}
spec Non_Generic_Monoid =
  sort Elem
then sort Monoid
  ops inj : Elem \rightarrow Monoid;
  1 : Monoid;
  __*__ : Monoid \times Monoid \rightarrow Monoid, assoc, unit 1
  \forall x, y : Elem \cdot inj(x) = inj(y) \Rightarrow x = y
end
References to generic specifications always instantiate the parameters.

\[
\text{spec } \text{Generic\_Commutative\_Monoid} [\text{sort} \text{ Elem}] = \\
\text{Generic\_Monoid} [\text{sort} \text{ Elem}]
\]
\[
\text{then } \forall x, y : \text{Monoid} \cdot x \ast y = y \ast x
\]
\[
\text{end}
\]

\[
\text{spec } \text{Generic\_Commutative\_Monoid\_1} [\text{sort} \text{ Elem}] = \\
\text{Generic\_Monoid} [\text{sort} \text{ Elem}]
\]
\[
\text{then } \text{op} \quad \text{--} \ast \text{--} : \text{Monoid} \times \text{Monoid} \rightarrow \text{Monoid}, \text{ comm}
\]
\[
\text{end}
\]
Datatype declarations may be used to abbreviate declarations of sorts and constructors.

\[
\text{spec } \text{Container } [\text{sort } \text{Elem}] = \\
\text{type } \text{Container ::= empty } \mid \text{insert(Elem; Container)} \\
\text{pred } \text{is_in : Elem } \times \text{Container} \\
\forall e, e' : \text{Elem; } C : \text{Container} \\
\bullet \lnot (e \text{ is_in empty}) \\
\bullet e \text{ is_in insert(e', C)} \iff (e = e' \lor e \text{ is_in C})
\]

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 × Container

∀e, e′ : Elem; C : Container
    • e is_in mark_insert(e′, C) ⇔ (e = e′ ∨ e is_in C)
    • ¬(e is_marked_in empty)
    • e is_marked_in insert(e′, C) ⇔ e is_marked_in C
    • e is_marked_in mark_insert(e′, C) ⇔ (e = e′ ∨ e is_marked_in C)

end
```
Generated Specifications
➤ Sorts may be specified as generated by their constructors.

\[
\text{spec } \text{GENERATED\_CONTAINER} \ [\text{sort } \text{Elem}] = \\
\text{generated type } \text{Container ::= empty} \mid \text{insert(} \text{Elem; Container} \text{)} \\
\text{pred } \_\text{is\_in\_} : \text{Elem} \times \text{Container} \\
\forall e, e' : \text{Elem; } C : \text{Container} \\
\quad \bullet \ (e \text{ is\_in empty}) \\
\quad \bullet \ e \text{ is\_in insert}(e', C) \iff (e = e' \lor e \text{ is\_in } C') \\
\text{end}
\]
Generated specifications are in general loose.

\[
\text{spec} \quad \text{GENERATED\_CONTAINER\_MERGE} \ [\text{sort} \ Elem] = \text{GENERATED\_CONTAINER} \ [\text{sort} \ Elem]
\]

\textbf{then op} \quad \text{merge} : \text{Container} \times \text{Container} \rightarrow \text{Container}

\forall e : \text{Elem}; \ C, C' : \text{Container}

- \text{e is\_in } (C \text{ merge } C') \iff (\text{e is\_in } C \lor \text{e is\_in } C')

\text{end}
Generated specifications need not be loose.

```casl
spec GENERATED_SET [ sort Elem ] =
generated type Set ::= empty | insert(Elem; Set)
pred _is_in_: Elem × Set
ops {_.}(e : Elem) : Set = insert(e, empty);
    _∪_ : Set × Set → Set;
    remove : Elem × Set → Set

∀e, e' : Elem; S, S' : Set
• ¬(e is_in empty)
• e is_in insert(e', S) ⇔ (e = e' ∨ e is_in S)
• S = S' ⇔ (∀x : Elem • x is_in S ⇔ x is_in S')  %(equal_sets)%
• e is_in (S ∪ S') ⇔ (e is_in S ∨ e is_in S')
• e is_in remove(e', S) ⇔ (¬(e = e') ∧ e is_in S)

then %implies ∀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 | {_.}(Elem) | _∪_ (Set; Set)
op _∪_ : Set × Set → Set, assoc, comm, idem, unit empty
end
```
Generated types may need to be declared together.

\[
\begin{align*}
\text{sort} & \quad \text{Node} \\
\text{generated type} & \quad \text{Tree} ::= \text{mktree}(\text{Node}; \text{Forest}) \\
\text{generated type} & \quad \text{Forest} ::= \text{empty} \mid \text{add}(\text{Tree}; \text{Forest})
\end{align*}
\]

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

\[
\begin{align*}
\text{sort} & \quad \text{Node} \\
\text{generated types} & \quad \text{Tree} ::= \text{mktree}(\text{Node}; \text{Forest}) \\
& \quad \text{Forest} ::= \text{empty} \mid \text{add}(\text{Tree}; \text{Forest})
\end{align*}
\]
Free Specifications
Free specifications provide initial semantics and avoid the need for explicit negation.

\[
\text{spec \ \textsc{Natural} = free type } Nat ::= 0 \mid suc(Nat)
\]
Free datatype declarations are particularly convenient for defining enumerated datatypes.

```
spec Color =
  free type RGB ::= Red | Green | Blue
  free type CMYK ::= Cyan | Magenta | Yellow | Black
end
```
Free specifications can also be used when the constructors are related by some axioms.

```plaintext
spec INTEGER =
  free { type Int ::= 0 | suc(Int) | pre(Int)
     ∀x : Int • suc(pre(x)) = x
     • pre(suc(x)) = x }
end
```
Predicates hold minimally in models of free specifications.

\[
\text{spec \ NATURAL ORDER = NATURAL}
\]

\textbf{then free \{ pred \_ \_ < \_ \_ : Nat \times Nat }

\[
\forall x, y : Nat \\
\quad \bullet \ 0 < suc(x) \\
\quad \bullet \ x < y \Rightarrow suc(x) < suc(y) \}
\]

\textbf{end}
Operations and predicates may be safely defined by induction on the constructors of a free datatype declaration.

\[\text{spec } \text{Natural\_Arithmetic } = \text{Natural\_Order}\]

\[\text{then } \text{ops } 1 : \text{Nat} = \text{suc}(0); \]
\[\quad \_ + \_ : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}, \text{assoc, comm, unit } 0; \]
\[\quad \_ \ast \_ : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}, \text{assoc, comm, unit } 1\]

\[\forall x, y : \text{Nat}\]
\[\quad \bullet x + \text{suc}(y) = \text{suc}(x + y)\]
\[\quad \bullet x \ast 0 = 0\]
\[\quad \bullet x \ast \text{suc}(y) = (x \ast y) + x\]
More care may be needed when defining operations or predicates on free datatypes when there are axioms relating the constructors.

```plaintext
spec INTEGER_ARITHMETIC =
  INTEGER
then ops 1 : Int = suc(0);
  _ + _ : Int × Int → Int, assoc, comm, unit 0;
  _ − _ : Int × Int → Int;
  _ * _ : Int × Int → Int, assoc, comm, unit 1
∀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
```

© Michel Bidoit, Peter D. Mosses, CoFI
spec \texttt{INTEGER\_ARITHMETIC\_ORDER} = \texttt{INTEGER\_ARITHMETIC}

then preds $\_\leq \_, \_\geq \_, \_\ < \_\_, \_\ > \_\_ : \texttt{Int} \times \texttt{Int}$

\[
\forall x, y : \texttt{Int} \\
\begin{itemize}
  \item $0 \leq 0$
  \item $\neg(0 \leq \text{pre}(0))$
  \item $0 \leq x \Rightarrow 0 \leq \text{suc}(x)$
  \item $\neg(0 \leq x) \Rightarrow \neg(0 \leq \text{pre}(x))$
  \item $\text{suc}(x) \leq y \iff x \leq \text{pre}(y)$
  \item $\text{pre}(x) \leq y \iff x \leq \text{suc}(y)$
  \item $x \geq y \iff y \leq x$
  \item $x < y \iff (x \leq y \land \neg(x = y))$
  \item $x > y \iff y < x$
\end{itemize}

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

\[
\text{spec } \text{List} \ [ \text{sort } \text{Elem} ] = \text{free type } \text{List} ::= \text{empty} \mid \text{cons} (\text{Elem}; \text{List})
\]

\[
\text{spec } \text{Set} \ [ \text{sort } \text{Elem} ] = \\
\quad \text{free } \{ \text{type } \text{Set} ::= \text{empty} \mid \text{insert} (\text{Elem}; \text{Set}) \\
\quad \quad \text{pred } \_\text{is_in}_\_ : \text{Elem} \times \text{Set} \\
\quad \forall e, e' : \text{Elem}; S : \text{Set} \\
\quad \quad \bullet \ \text{insert} (e, \text{insert} (e, S)) = \text{insert} (e, S) \\
\quad \quad \bullet \ \text{insert} (e, \text{insert} (e', S)) = \text{insert} (e', \text{insert} (e, S)) \\
\quad \quad \bullet \ \neg (e \text{ is_in empty}) \\
\quad \quad \bullet \ e \text{ is_in insert} (e, S) \\
\quad \quad \bullet \ e \text{ is_in insert} (e', S) \text{ if } e \text{ is_in } S \}
\]

end
spec Transitive_Closure [sort Elem pred _R_ : Elem × Elem] =
free { pred _R+_ : Elem × Elem

∀x, y, z : Elem

• x R y ⇒ x R+y

• x R+y ∧ y R+z ⇒ x R+z }
Loose extensions of free specifications can avoid overspecification.

```casl
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 → Elem
  ∀S : Set • ¬(S = empty) ⇒ choose(S) is_in S
end
```
Datatypes with observer operations or predicates can be specified as generated instead of free.

\[
\text{spec } \text{Set}_{\text{Generated}} [\text{sort } \text{Elem}] =
\]

\[
\text{generated type } \text{Set} ::= \text{empty} | \text{insert}(\text{Elem}; \text{Set})
\]

\[
\text{pred } \text{is} \text{in} : \text{Elem} \times \text{Set}
\]

\[
\forall e, e' : \text{Elem}; S, S' : \text{Set}
\]

\begin{itemize}
    \item \neg (\text{e is in empty})
    \item \text{e is in insert}(e', S) \iff (e = e' \lor \text{e is in } S)
    \item S = S' \iff (\forall x : \text{Elem} \bullet x \text{ is in } S \iff x \text{ is in } S')
\end{itemize}

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

\[
\text{spec } \text{SET\_UNION} [\text{sort } \text{Elem}] = \\
\text{SET} [\text{sort } \text{Elem}]
\]

\text{then } %\text{def}

\text{ops } _\cup_ _ : \text{Set} \times \text{Set} \to \text{Set}, \text{ assoc, comm, idem, unit empty;}

remove : \text{Elem} \times \text{Set} \to \text{Set}

\forall e, e' : \text{Elem}; S, S' : \text{Set}

\begin{itemize}
  \item \( S \cup \text{insert}(e', S') = \text{insert}(e', S \cup S') \)
  \item remove\((e, \text{empty}) = \text{empty} \)
  \item remove\((e, \text{insert}(e, S)) = \text{remove}(e, S) \)
  \item remove\((e, \text{insert}(e', S)) = \text{insert}(e', \text{remove}(e, S)) \text{ if } -(e = e') \)
\end{itemize}

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

\[
\text{spec } \text{SET\_UNION\_1 [sort } \text{Elem} ] = \\
\text{SET\_GENERATED [sort } \text{Elem} ]
\]

\text{then } \%	ext{def}

\text{ops } \_ \_ \text{ } \cup \_ \_ : \text{Set} \times \text{Set} \rightarrow \text{Set}, \text{ assoc, comm, idem, unit empty;}

\text{ remove : Elem} \times \text{Set} \rightarrow \text{Set}

\forall e, e' : \text{Elem}; S, S' : \text{Set}

\begin{itemize}
  \item \text{e is}_{-\text{in}} \text{ } (S \cup S') \iff (e \text{ is}_{-\text{in}} S \lor e \text{ is}_{-\text{in}} S')
  \item \text{e is}_{-\text{in}} \text{ remove}(e', S) \iff (\neg(e = e') \land e \text{ is}_{-\text{in}} S)
\end{itemize}

\text{end}
Sorts declared in free specifications are not necessarily generated by their constructors.

```plaintext
spec UnNatural =
   free { type UnNat ::= 0 | suc(UnNat)
               op   _ + _ : UnNat \times UnNat \to 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 ->? 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 \( \text{choose}(\text{empty}) \) may be undefined.
Functions, even total ones, propagate undefinedness.

If the term \(\text{choose}(S)\) is undefined for some value of \(S\), then the term \(\text{insert}(\text{choose}(S), S')\) is undefined as well for this value of \(S\), although \(\text{insert}\) 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:

\[
\text{insert}(\text{choose}(S), \text{insert}(\text{choose}(S), \text{empty})) = \text{insert}(\text{choose}(S), \text{empty})
\]

holds also when the term \text{choose}(S) is undefined.
Special care is needed in specifications involving partial functions.

- Asserting \( \text{choose}(S) \text{ is in } S \) as an axiom implies that \( \text{choose}(S) \) is defined, for any \( S \).

- Asserting \( \text{remove}(\text{choose}(S), \text{insert}(\text{choose}(S), \text{empty})) = \text{empty} \) as an axiom implies that \( \text{choose}(S) \) is defined for any \( S \), since the term \( \text{empty} \) is always defined.

- Asserting \( \text{insert}(\text{choose}(S), S) = S \) as an axiom implies that \( \text{choose}(S) \) is defined for any \( S \), since a variable always denotes a defined value.
The definedness of a term can be checked or asserted.

\[
\text{spec} \quad \text{Set\_Partial\_Choose\_1} [\text{sort} \ Elem] = \\
\quad \text{Set\_Partial\_Choose} [\text{sort} \ Elem]
\]

\[
\text{then} \quad \bullet \quad \neg \text{def choose}(\text{empty})
\]

\[
\forall S : \text{Set} \quad \bullet \quad \text{def choose}(S) \Rightarrow \text{choose}(S) \text{ is in } S
\]

\]

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

If we would have specified \text{choose} by:

\[
\forall S : \text{Set} \quad \bullet \quad \neg(S = \text{empty}) \Rightarrow \text{choose}(S) \text{ is in } S
\]

then we could conclude that \text{choose}(S) is defined when \text{S} is not equal to \text{empty}, but nothing about the undefinedness of \text{choose}(\text{empty}).
The domains of definition of partial functions can be specified exactly.

\[
\text{spec } \texttt{Set\_Partial\_Choose\_2 [sort Elem]} = \\
\quad \texttt{Set\_Partial\_Choose [sort Elem]}
\]

\[
\text{then } \forall S : \text{Set} \bullet \text{def choose}(S) \iff \neg (S = \text{empty})
\]

\[
\forall S : \text{Set} \bullet \text{def choose}(S) \Rightarrow \text{choose}(S) \text{ is\_in } S
\]

end
Loosely specified domains of definition may be useful.

```plaintext
spec Natural_With_Bound_And_Addition = Natural_With_Bound

then op __+?__ : Nat × Nat →? Nat

∀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) ⇒ x+?y = x + y

end
```
Domains of definition can be specified more or less explicitly.

\[
\text{spec SET\_PARTIAL\_CHOOSE}_3 [\text{sort Elem}] = \\
\text{SET\_PARTIAL\_CHOOSE} [\text{sort Elem}]
\]

then \( \bullet \neg \text{def choose}(\text{empty}) \)

\[\forall S : \text{Set} \bullet \neg (S = \text{empty}) \Rightarrow \text{choose}(S) \text{ is in } S\]

end

We can conclude after some reasoning that:

\[\text{def choose}(S) \Leftrightarrow \neg (S = \text{empty})\]

but this is not so prominent.
spec Natural_Partial_Pre =
    Natural_Arithmetic
then op pre : Nat \rightarrow ? Nat
    \bullet \leftarrow \text{def } pre(0)
    \forall x : Nat \bullet pre(suc(x)) = x
end

is explicit enough.
spec \textbf{Natural}._\textbf{Partial}._\textbf{Subtraction} \_1 = 
\textbf{Natural}._\textbf{Partial}._\textbf{Pre}

\textbf{then \ op} \ _\_ \ − \ _\_: \textit{Nat} \times \textit{Nat} \rightarrow? \textit{Nat}

\forall \ x, \ y : \textit{Nat}

\bullet \ x - 0 = x

\bullet \ x - \textit{suc}(y) = \textit{pre}(x - y)

\textbf{end}

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

spec \textbf{Natural}._\textbf{Partial}._\textbf{Subtraction} = 
\textbf{Natural}._\textbf{Partial}._\textbf{Pre}

\textbf{then \ op} \ _\_ \ − \ _\_: \textit{Nat} \times \textit{Nat} \rightarrow? \textit{Nat}

\forall \ x, \ y : \textit{Nat}

\bullet \ \textit{def}(x - y) \Leftrightarrow (y < x \lor y = x)

\bullet \ x - 0 = x

\bullet \ x - \textit{suc}(y) = \textit{pre}(x - y)

\textbf{end}
Partial functions are minimally defined by default in free specifications.

```casl
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 ListSelectors_2 [sort Elem] =
  List [sort Elem]
then ops head : List →? Elem;
     tail : List →? List

∀ e : Elem; L : List
  • ¬ def head(empty)
  • ¬ 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.

\[
\text{spec } \text{List.Selectors} \text{[ sort } Elem \text{]} = \\
\text{free type } List ::= empty \mid \text{cons}(\text{head } : ? Elem; \text{tail } : ? List)
\]

\[
\text{spec } \text{Natural.Suc.Pre} = \text{free type } Nat ::= 0 \mid \text{suc}(\text{pre } : ? Nat)
\]
Selectors are usually total when there is only one constructor.

\[
\text{spec } \textsc{Pair}_1 \ [ \text{sorts } \textit{Elem1}, \textit{Elem2} ] = \\
\text{free type } \textsc{Pair} ::= \textit{pair}(\textit{first} : \textit{Elem1}; \textit{second} : \textit{Elem2})
\]
Constructors may be partial.

\[\text{spec} \ \text{Part\_Container} [\text{sort} \ Elem] =\]

\[
\begin{align*}
\text{generated type} & \\
P\_\text{Container} & ::= \text{empty} \mid \text{insert}(\text{Elem}; \ P\_\text{Container})?\end{align*}
\]

\[
\text{pred} \ \text{addable} : \text{Elem} \times P\_\text{Container}
\]

\[
\text{vars} \ e, e' : \text{Elem}; \ C : P\_\text{Container}
\]

\[
\begin{align*}
\bullet & \ \text{def} \ \text{insert}(e, C) \iff \text{addable}(e, C)\\
\text{pred} \ \text{is\_in} : \text{Elem} \times P\_\text{Container} & \\
\bullet & \ -(e \ \text{is\_in} \ \text{empty})\\
\bullet & \ (e \ \text{is\_in} \ \text{insert}(e', C) \iff (e = e' \vee e \ \text{is\_in} \ C)) \ \text{if} \ \text{addable}(e', C)
\end{align*}
\]

\text{end}
Existential equality requires the definedness of both terms as well as their equality.

\begin{verbatim}
spec  NATURAL_PARTIAL_SUBTRACTION_2 =
    NATURAL_PARTIAL_SUBTRACTION_1
then  \forall x, y, z : Nat  \bullet  y - x \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
\end{verbatim}
Subsorting

➤ Subsorts and supersorts are often useful in CASL specifications.
Subsort declarations directly express relationships between carrier sets.

```casl
spec GENERIC_MONOID_1 [sort Elem] =
    sorts Elem < Monoid
    ops 1 : Monoid;
        __ * __ : Monoid \times Monoid \rightarrow Monoid, assoc, unit 1
end
```

© Michel Bidoit, Peter D. Mosses, CoFI
Operations declared on a sort are automatically inherited by its subsorts.

\[
\text{spec } \text{VEHICLE} = \\
\text{NATURAL}
\]

then sorts \( Car, \ Bicycle < \text{Vehicle} \)

ops \( \text{max\_speed} : \text{Vehicle} \rightarrow \text{Nat} ; \)

\( \text{weight} : \text{Vehicle} \rightarrow \text{Nat} ; \)

\( \text{engine\_capacity} : \text{Car} \rightarrow \text{Nat} \)

end
Inheritance applies also for subsorts that are declared afterwards.

\[
\text{spec } \texttt{MORE\_VEHICLE} = \texttt{VEHICLE} \text{ then sorts } \texttt{Boat} < \texttt{Vehicle}
\]
Subsort membership can be checked or asserted.

```plaintext
spec SPEED_REGULATION =
  VEHICLE
then ops speed_limit : Vehicle → Nat;
    car_speed_limit, bike_speed_limit : Nat

∀v : Vehicle
  • v ∈ Car ⇒ speed_limit(v) = car_speed_limit
  • v ∈ Bicycle ⇒ speed_limit(v) = bike_speed_limit
end
```
Datatype declarations can involve subtset declarations.

\begin{itemize}
  \item \textbf{sorts} Car, Bicycle, Boat
  \item \textbf{type} \texttt{Vehicle ::= sort Car | sort Bicycle | sort Boat}
\end{itemize}

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

\begin{itemize}
  \item \textbf{sorts} Car, Bicycle, Boat
  \item \textbf{generated type} \texttt{Vehicle ::= sort Car | sort Bicycle | sort Boat}
\end{itemize}

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

\begin{itemize}
  \item \textbf{sorts} Car, Bicycle, Boat
  \item \textbf{free type} \texttt{Vehicle ::= sort Car | sort Bicycle | sort Boat}
\end{itemize}

prevents the definition of further subsorts, and moreover the definition of a common
subsort of both Car and Boat (e.g., \textbf{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)
    • ¬ even(1)
∀n : Nat • even(suc(suc(n))) ⇔ even(n)
sort Even = {x : Nat • even(x)}
sort Prime = {x : Nat • 1 < x ∧
    ∀y, z : Nat • x = y * z ⇒ y = 1 ∨ z = 1}
end

spec Positive =
    Natural_Partial_Pre
then sort Pos = {x : Nat • ¬(x = 0)}
```
It may be useful to redeclare previously defined operations, using the new subsorts introduced.

```casl
spec Positive_Arithmetic =
    Positive
then ops 1 : Pos;
    suc : Nat → Pos;
    _ + _, _ * _ : Pos × Pos → Pos;
    _ + _ : Pos × Nat → Pos;
    _ + _ : Nat × Pos → Pos
end
```
A subsort may correspond to the definition domain of a partial function.

```plaintext
spec Positive_Pre = Positive_Arithmetic
then op pre : Pos → Nat
```
Using subsorts may avoid the need for partial functions.

```
spec Natural_Positive_Arithmetic =

  free types Nat ::= 0 | sort Pos;
  Pos ::= suc(pre : Nat)

  ops 1 : Pos = suc(0);
  __ + __ : Nat × Nat → Nat, assoc, comm, unit 0;
  __ * __ : Nat × Nat → Nat, assoc, comm, unit 1;
  __ + __, __ * __ : Pos × Pos → Pos;
  __ + __ : Pos × Nat → Pos;
  __ + __ : Nat × Pos → Pos

  ∀x, y : Nat
  • x + suc(y) = suc(x + y)
  • x * 0 = 0
  • x * suc(y) = x + (x * y)
```

© Michel Bidoit, Peter D. Mosses, CoFI
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 $\text{def } (t \text{ as } s)$ is equivalent to $t \in s$.

- $\text{pre}(\text{pre} (\text{suc}(1)) \text{ as Pos})$
- $\text{def } \text{pre}(\text{pre} (\text{suc}(1)) \text{ as Pos})$
- $\neg \text{def } (\text{pre}(\text{pre} (\text{suc}(1)) \text{ as Pos}) \text{ as Pos})$
Supersorts may be useful when generalizing previously specified sorts.

```
spec INTEGER_ARITHMETIC_1 =
Natural_Positive_Arithmetic

then free type Int ::= sort Nat | -(Pos)

ops __ + __ : Int × Int → Int, assoc, comm, unit 0;
   __ − __ : Int × Int → Int;
   __ * __ : Int × Int → Int, assoc, comm, unit 1

∀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.

\[
\text{spec } \text{SET\_ERROR\_CHOOSE } [\text{sort } \text{Elem}] = \\
\text{GENERATED\_SET } [\text{sort } \text{Elem}]
\]

\[
\text{then sorts } \text{Elem} < \text{ElemError} \\
\text{op } \text{choose} : \text{Set} \to \text{ElemError} \\
\text{pred } \text{is\_in\_} : \text{ElemError} \times \text{Set} \\
\forall S : \text{Set} \implies \neg (S = \text{empty}) \implies \text{choose}(S) \in \text{Elem} \land \text{choose}(S) \text{ is\_in } S
\]

\[
\text{end}
\]

\[
\text{spec } \text{SET\_ERROR\_CHOOSE\_1 } [\text{sort } \text{Elem}] = \\
\text{GENERATED\_SET } [\text{sort } \text{Elem}]
\]

\[
\text{then sorts } \text{Elem} < \text{ElemError} \\
\text{op } \text{choose} : \text{Set} \to \text{ElemError} \\
\forall S : \text{Set} \implies \neg (S = \text{empty}) \implies (\text{choose}(S) \text{ as Elem}) \text{ is\_in } S
\]

\[
\text{end}
\]
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 \textsc{List\_Set} \ [ \text{sort} \ Elem \] =
\textsc{List\_Selectors} \ [ \text{sort} \ Elem \]
and \textsc{Generated\_Set} \ [ \text{sort} \ Elem \]
then op \texttt{elements\_of\_} : \texttt{List} → \texttt{Set}
\begin{align*}
∀ e : \texttt{Elem}; \ L : \texttt{List} \\
\bullet & \texttt{elements\_of\_} \text{ empty} = \texttt{empty} \\
\bullet & \texttt{elements\_of\_} \text{ cons}(e, L) = \{ e \} \cup \texttt{elements\_of\_} \text{ L}
\end{align*}
end
Specifications may combine parts with loose, generated, and free interpretations.

```plaintext
spec List_Choose [sort Elem] =
  ListSelectors [sort Elem]
and Set_Partial_Choose_2 [sort Elem]
then ops elements_of _: List → Set;
  choose: List →? Elem

∀e : Elem; L : List
  • elements_of empty = empty
  • elements_of cons(e, L) = {e} ∪ elements_of L
  • def choose(L) ⇔ ¬(L = empty)
  • choose(L) = choose(elements_of L)
end
```
spec  \textbf{Set\_to\_List} [\textbf{sort} \textit{Elem}] = \\
\textbf{List\_Set} [\textbf{sort} \textit{Elem}]

\textbf{then op} \textit{list\_of} \_\_ : \textit{Set} \rightarrow \textit{List}

\forall S : \textit{Set} \bullet \textit{elements\_of} (\textit{list\_of} S) = S

\textbf{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 \rightarrow Stack,

  ops cons \rightarrow \text{push}

  head \rightarrow \text{top},

  tail \rightarrow \text{pop}

end
```
When combining specifications, origins of symbols can be indicated.

```plaintext
spec List_Set_1 [sort Elem] =
   List_Selectors [sort Elem] with empty, cons
and Generated_Set [sort Elem] with empty, {__}, __ ∪ __
then op elements_of__: List → Set

∀ e : Elem; L : List

  • elements_of empty = empty

  • elements_of cons(e, L) = {e} ∪ elements_of L

end
```
Auxiliary symbols used in structured specifications can be hidden.

```plaintext
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 __ ≤ __
```
Auxiliary symbols can be made local when they do not need to be exported.

spec \textbf{List\_Order} [\textbf{Total\_Order} with sort \textit{Elem}, pred \_\_ < \_] = \\
\textbf{List\_Selectors} [sort \textit{Elem}]
then local \textbf{op} \textit{insert} : \textit{Elem} \times \textit{List} \rightarrow \textit{List}
\begin{align*}
\forall e, e' : \textit{Elem}; L : \textit{List} \\
\begin{itemize}
\item \textit{insert}(e, \text{empty}) = \text{cons}(e, \text{empty})
\item \textit{insert}(e, \text{cons}(e', L)) = \text{cons}(e', \text{insert}(e, L)) \text{ when } e' < e \\
\text{else } \text{cons}(e, \text{cons}(e', L))
\end{itemize}
\end{align*}
within \textbf{op} \textit{order} : \textit{List} \rightarrow \textit{List}
\begin{align*}
\forall e : \textit{Elem}; L : \textit{List} \\
\begin{itemize}
\item \textit{order}(\text{empty}) = \text{empty}
\item \textit{order}(\text{cons}(e, L)) = \text{insert}(e, \textit{order}(L))
\end{itemize}
\end{align*}
end
spec **List_Order_Sorted**

\[
[\text{TOTAL\_ORDER} \text{ with sort } \text{Elem}, \text{pred } \_\_ < \_\_] = \\
\text{List\_Selectors} [\text{sort } \text{Elem}]
\]

then local \text{pred } \_\_\text{is\_sorted} : \text{List}

\[\forall e, e' : \text{Elem}; L : \text{List}
\]

\[\begin{array}{l}
\bullet \text{empty is\_sorted} \\
\bullet \text{cons}(e, \text{empty}) \text{ is\_sorted} \\
\bullet \text{cons}(e, \text{cons}(e', L)) \text{ is\_sorted} \iff \\
\qquad \text{cons}(e', L) \text{ is\_sorted} \land \neg(e' < e)
\end{array}\]

**within op** \text{order} : \text{List} \rightarrow \text{List}

\[\forall L : \text{List} \bullet \text{order}(L) \text{ is\_sorted}\]

end
Care is needed with local sort declarations.

spec WRONG_LIST_ORDER_SORTED

\[ \text{[Total Order with sort Elem, pred \( \_ \_ < \_ \_ \)] = List_Selectors[sort Elem]} \]

then local pred \_\_is_sorted : List

sort SortedList \( = \{L : \text{List} \quad \bullet \quad L \text{ is_sorted}\} \)

\( \forall e, e' : \text{Elem}; \quad L : \text{List} \)

- empty is_sorted
- cons(e, empty) is_sorted
- cons(e, cons(e', L)) is_sorted \( \Leftrightarrow \)
  \( \text{cons}(e', L) \text{ is_sorted} \wedge \neg(e' < e) \)

within op order : List \( \rightarrow \) SortedList

end
spec **List_Order_Sorted_2**

\[[\text{Total_Order} \text{ with sort } Elem, \ pred \ _\_ < \_\_] = \text{List_Selectors} [\text{sort } Elem]\]

then local \ pred \ _\_\_is_sorted : List

\[\forall e, e' : Elem; \ L : List\]

\[\begin{align*}
& \bullet \ \text{empty is\_sorted} \\
& \bullet \ \text{cons}(e, \text{empty}) \text{ is\_sorted} \\
& \bullet \ \text{cons}(e, \text{cons}(e', L)) \text{ is\_sorted} \iff \text{cons}(e', L) \text{ is\_sorted} \land \neg(e' < e) \\
\end{align*}\]

within \ sort \ SortedList = \{L : List \bullet L \text{ is\_sorted}\}

op \ order : List \rightarrow SortedList

end
spec List_Order_Sorted_3

[Total_Order with sort Elem, \text{pred } _-_ < _-_] = List_Selectors [sort Elem]

then \{ \text{pred } _-_\text{is_sorted} : List \\
\forall e, e' : Elem; L : List \\
\quad \bullet \text{ empty is_sorted} \\
\quad \bullet \text{ cons}(e, \text{empty}) \text{ is_sorted} \\
\quad \bullet \text{ cons}(e, \text{cons}(e', L)) \text{ is_sorted} \iff \\
\quad \quad \text{cons}(e', L) \text{ is_sorted} \land \neg(e' < e) \\
\text{then sort SortedList} = \{ L : List \bullet L \text{ is_sorted}\}
\text{op order : List} \rightarrow \text{SortedList} \\
\text{hide } _-_\text{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.

\[
\text{spec } \text{LIST\_ORDER\_NAT} = \text{LIST\_ORDER} [\text{NATURAL\_ORDER}]
\]
The argument specification of an instantiation must ensure that the properties required by the parameter hold.

\[
\text{spec \ Nat\_Word} = \text{Generic\_Monoid \ [Natural]} \\
\text{spec \ List\_Order\_Nat} = \text{List\_Order \ [Natural\_Order]}
\]

The definition of \text{Nat\_Word} abbreviates:
\text{Natural \ and \ \{} \text{Non\_Generic\_Monoid \ with \ Elem \mapsto \Nat} \ \}. 

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

\[
\text{spec \ Nat\_Word\_1} = \\
\text{Generic\_Monoid \ [Natural]} \text{ \ with \ Monoid \mapsto \Nat\_Word} \\
\text{end}
\]
There must be no shared symbols between the argument specification and the body of the instantiated generic specification.

\[
\text{spec} \quad \text{THIS\_IS\_WRONG} = [\text{GENERIC\_MONOID}] [\text{MONOID}]
\]

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.

\[
\text{spec } \text{List\_Order\_Positive} = \text{List\_Order} [\text{Positive}]
\]
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 -> Nat ]
```
A generic specification may have more than one parameter.

\[
\text{spec \ Pair} \ [\text{sort} \ Elem1 \ ] \ [\text{sort} \ Elem2 \ ] = \\
\text{free type} \ Pair ::= \text{pair}(\text{first : Elem1}; \text{second : Elem2})
\]

\[
\text{spec \ Table} \ [\text{sort} \ Key \ ] \ [\text{sort} \ Val \ ] = \ldots
\]

Note that writing:

\[
\text{spec \ Homogeneous\_Pair\_1} \ [\text{sort} \ Elem \ ] \ [\text{sort} \ Elem \ ] = \\
\text{free type} \ Pair ::= \text{pair}(\text{first : Elem}; \text{second : Elem})
\]

merely defines pairs of values of the same sort, and \text{Homogeneous\_Pair\_1} is (equivalent to and) better defined as follows:

\[
\text{spec \ Homogeneous\_Pair} \ [\text{sort} \ Elem \ ] = \\
\text{free type} \ Pair ::= \text{pair}(\text{first : Elem}; \text{second : Elem})
\]
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 ↦ 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 ↦ Nat, Elem2 ↦ RGB
```
When parameters are trivial, one can always avoid explicit fitting maps.

\[
\text{spec } \text{Pair\_Natural\_Color\_2} = \\
\text{Pair } [\text{sort } \text{Nat}] [\text{sort } \text{RGB}]
\]

and \nat \text{Natural\_Arithmetic} \text{ and } \text{Color}

Compare for instance:

\[
\text{spec } \text{Pair\_Pos} = \\
\text{Homogeneous\_Pair } [\text{sort } \text{Pos}] \text{ and } \text{Integer\_Arithmetic\_1}
\]

with:

\[
\text{spec } \text{Pair\_Pos\_1} = \\
\text{Homogeneous\_Pair } [[\text{Integer\_Arithmetic\_1} \text{ fit } \text{Elem} \mapsto \text{Pos}]]
\]

Note that the instantiation:

\[
\text{Homogeneous\_Pair\_1} [\text{Natural}] [\text{Color} \text{ fit } \text{Elem} \mapsto \text{RGB}]
\]

is ill-formed, since it entails mapping the sort \text{Elem} to both \text{Nat} and \text{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 [ListSelectors [sort Elem] fit Elem ↦ List ]
```

Note especially that the following specification:

```
spec Mistake [sort Elem] =
  GENERATED_SET [ListSelectors [sort Elem ]]
```

does not provide sets of lists of elements.

```
spec Set_and_List [sort Elem] =
  GENERATED_SET [sort Elem ] and ListSelectors [sort Elem ]
```
It may be worth mentioning that the following composition of generic specifications is ill-formed:

```casl
spec This_Is_Still_Wrong =
    Generic_Monoid [ Generic_Monoid [ sort Elem ]
        fit Elem ↦ Monoid ]
```

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

```casl
spec LIST_REV [sort Elem] =
  free type List[Elem] ::= empty |
  cons(head :? Elem; tail :? List[Elem])
  ops  _ ++ _ : List[Elem] × List[Elem] → List[Elem],
       assoc, unit empty;
  reverse : List[Elem] → List[Elem]

∀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 ↦ RGB]  %% Provides the sort List[RGB]

spec Two_Lists_1 =
  List_Rev [Integer_Arithmetic_1 fit Elem ↦ Nat]
and List_Rev [Integer_Arithmetic_1 fit Elem ↦ Int]

Remember that Nat < Int does not entail List[Nat] < List[Int].
spec Monoid_C [sort Elem] =

  sort Monoid[Elem]

  ops inj : Elem → Monoid[Elem];

  1 : Monoid[Elem];


  assoc, unit 1

  ∀x, y : Elem • inj(x) = inj(y) ⇒ x = y

end

spec Monoid_of_Monoid [sort Elem] =

  Monoid_C [Monoid_C [sort Elem] fit Elem ↦ 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 × List[Elem] → List[Elem]
  ∀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] → List[Elem]
  ∀e : Elem; L : List[Elem]
    • order[<](empty) = empty
    • order[<](cons(e, L)) = insert(e, order[<](L))
end
```
spec \textbf{List\_Rev\_with\_Two\_Orders} =

\textbf{List\_Rev\_Order}

\begin{itemize}
\item \texttt{[Integer\_Arithmetic\_Order]} fit \texttt{Elem \mapsto Int, \_ < \_ \mapsto \_ < \_} \\
\hfill \text{%% Provides the sort \textit{List[Int]} and the operation \textit{order[\_ < \_]}}
\end{itemize}

and \textbf{List\_Rev\_Order}

\begin{itemize}
\item \texttt{[Integer\_Arithmetic\_Order]} fit \texttt{Elem \mapsto Int, \_ < \_ \mapsto \_ > \_} \\
\hfill \text{%% Provides the sort \textit{List[Int]} and the operation \textit{order[\_ > \_]}}
\end{itemize}

then \texttt{\%implies}

\forall L : \textit{List[Int]} \bullet \textit{order[\_ < \_]}(L) = \text{reverse}(\textit{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 → Nat]
  given Natural_Arithmetic = List_Rev [sort Elem]
then op weight : List[Elem] → Nat
  \forall e : Elem; L : List[Elem]
    • weight(empty) = 0
    • weight(cons(e, L)) = weight(e) + weight(L)
end
```
One could have written instead:

```plaintext
spec List_Weighted_Elem
    [Natural_Arithmetic \textbf{then sort} Elem \textbf{op} weight : Elem \rightarrow Nat] = \ldots
```

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 ‘\textbf{given Natural_Arithmetic}’ clause would make the
declaration:

```plaintext
spec List_Weighted_Elem [\textbf{sort} Elem \textbf{op} weight : Elem \rightarrow Nat] = \ldots
```

ill-formed, since the sort \textit{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 ↦ Pair,
                        weight ↦ first ]

spec List_Weighted_Instantiated =
    List_Weighted_Elem [sort Value op weight : Value → 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] → Nat
   ∀ e : Elem; L : List[Elem]
   • length(empty) = 0
   • length(cons(e, L)) = length(L) + 1
then %implies
   ∀ L : List[Elem] • length(reverse(L)) = length(L)
end

spec List_Length_Natural = List_Length [Natural_Arithmetic]
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 \text{ then } \ldots \]

are better written

\[ F \[ X \] \text{ 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 ↦ Int, __ < __ ↦ __ < __
```

**view** `INTEGER_AS_REVERSE_TOTAL_ORDER` :

```
TOTAL_ORDER to INTEGER_ARITHMETIC_ORDER =
Elem ↦ Int, __ < __ ↦ __ > __
```

**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`

```
∀L : List[Int] • order[__ < __](L) = reverse(order[__ > __](L))
```

end
Views can also be generic.

```casl
view List_as_Monoid [sort Elem] :
  Monoid to List_Rev [sort Elem] =
  Monoid ↦ List[Elem], 1 ↦ empty, _ * _ ↦ _ + _
```
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.

\[
\begin{align*}
\text{spec} \quad \text{Color} &= \ldots \\
\text{spec} \quad \text{Natural\_Order} &= \ldots \\
\text{spec} \quad \text{Natural\_Arithmetic} &= \ldots \\
\text{spec} \quad \text{Elem} &= \text{sort} \quad \text{Elem} \\
\text{spec} \quad \text{Cont}[\text{Elem}] &= \\
\text{generated type} \quad \text{Cont}[\text{Elem}] &::= \text{empty} \mid \text{insert}(\text{Elem}; \ \text{Cont}[\text{Elem}]) \\
\text{preds} \quad \text{is\_empty} : \ \text{Cont}[\text{Elem}]; \\
\text{is\_in} : \ \text{Elem} \times \ \text{Cont}[\text{Elem}] \\
\text{ops} \quad \text{choose} : \ \text{Cont}[\text{Elem}] \rightarrow \text{?} \ \text{Elem}; \\
\quad \text{delete} : \ \text{Elem} \times \ \text{Cont}[\text{Elem}] \rightarrow \ \text{Cont}[\text{Elem}] \\
\forall e, e' : \ \text{Elem}; \ \ C : \ \text{Cont}[\text{Elem}] \\
\quad \bullet \ \text{empty is\_empty} \\
\quad \bullet \ \neg \ \text{insert}(e, C) \ \text{is\_empty} \\
\quad \bullet \ \neg \ e \ \text{is\_in} \ \text{empty} \\
\quad \bullet \ e \ \text{is\_in} \ \text{insert}(e', C) \ \iff \ (e = e' \ \lor \ e \ \text{is\_in} \ C) \\
\quad \bullet \ \text{def} \ \text{choose}(C) \ \iff \ \neg \ C \ \text{is\_empty} \\
\quad \bullet \ \text{def} \ \text{choose}(C) \ \Rightarrow \ \text{choose}(C) \ \text{is\_in} \ C \\
\quad \bullet \ e \ \text{is\_in} \ \text{delete}(e', C) \ \iff \ (e \ \text{is\_in} \ C \ \land \ \neg (e = e')) \\
\end{align*}
\]
spec \texttt{Cont-Diff [Elem]} =
\texttt{Cont [Elem]}

then op \texttt{diff} : \texttt{Cont[Elem]} \times \texttt{Cont[Elem]} \rightarrow \texttt{Cont[Elem]}
\forall e : \texttt{Elem}; \ C, C' : \texttt{Cont[Elem]}
\begin{itemize}
\item \texttt{e is_in} \texttt{diff (C, C')} \Leftrightarrow (\texttt{e is_in} C \land \neg(\texttt{e is_in} C'))
\end{itemize}
end

spec \texttt{REQ} = \texttt{Cont-Diff [Natural_Order]}
spec  FLAT_REQ =
   free type  Nat ::= 0 | suc(Nat)
   pred    _ < _ : Nat × Nat

generated type  Cont[Nat] ::= empty | insert(Nat; Cont[Nat])

peds  _is_empty : Cont[Nat];
      _is_in_ : Nat × Cont[Nat]

ops  choose : Cont[Nat] →? Nat;
     delete : Nat × Cont[Nat] → Cont[Nat];

∀ e, e' : Nat; C, C' : Cont[Nat]
  • 0 < suc(e)
  • ¬(e < 0)
  • suc(e) < suc(e') ⇔ e < e'
  • empty is_empty
  • ¬ insert(e, C) is_empty
  • ¬ e is_in empty
  • e is_in insert(e', C) ⇔ (e = e' ∨ e is_in C)
  • def choose(C) ⇔ ¬ C is_empty
  • def choose(C) ⇒ choose(C) is_in C
  • e is_in delete(e', C) ⇔ (e is_in C ∧ ¬(e = e'))
  • e is_in diff(C, C') ⇔ (e is_in C ∧ ¬(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.

\begin{verbatim}
arch spec SYSTEM =

units  N :  NATURAL_ORDER;
        C :  CONT [NATURAL_ORDER]   given  N;
        D :  CONT_DIFF [NATURAL_ORDER]  given  C

result  D
\end{verbatim}
There can be several distinct architectural choices for the same requirements specification.

\[
\text{arch spec } \text{SYSTEM}_1 = \\
\text{units } N : \text{NATURAL\_ORDER}; \\
\text{CD} : \text{CONT\_DIFF}\{\text{NATURAL\_ORDER}\} \text{ given } N \\
\text{result } CD
\]
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 $\text{Cont-Diff} [\text{Natural-Order}]$ using different implementations of $\text{Cont} [\text{Natural-Order}]$. 
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 $\text{CONT\_DIFF} [\text{NATURAL\_ORDER}]$ is a conservative extension of $\text{CONT} [\text{NATURAL\_ORDER}]$. 
**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 \text{SYSTEM\_G} =

\text{units} \quad N : \text{NATURAL\_ORDER};

\quad F : \text{NATURAL\_ORDER} \rightarrow \text{CONT} \left[ \text{NATURAL\_ORDER} \right];

\quad G : \text{CONT} \left[ \text{NATURAL\_ORDER} \right] \rightarrow \text{CONT\_DIFF} \left[ \text{NATURAL\_ORDER} \right]

\text{result} \quad G \left[ F \left[ N \right] \right]
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.

\[
\text{unit spec } \text{CONT\_COMP} = \text{Elem} \rightarrow \text{Cont} \ [\text{Elem}]
\]

\[
\text{unit spec } \text{DIFF\_COMP} = \text{Cont} \ [\text{Elem}] \rightarrow \text{Cont\_Diff} \ [\text{Elem}]
\]

\[
\text{arch spec } \text{SYSTEM\_G1} = \\
\text{units } N : \text{Natural\_Order}; \\
F : \text{CONT\_COMP}; \\
G : \text{DIFF\_COMP}
\]

\[
\text{result } G [F[N]]
\]
Both named and un-named specifications can be used to specify components.

unit spec Diff_Comp_1 =

```
              ∀e : Elem; C, C' : Cont[Elem]
              ● e is_in diff(C, C') ⇔
                      (e is_in C ∧ ¬(e is_in C')) }
```

© Michel Bidoit, Peter D. Mosses, CoFI

132

CASL Tutorial
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 $\text{Cont\_Comp}$.

- 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 $\text{generic component}$ which gets $\text{applied}$ to suitable components.
A generic component may be applied more than once in the same architectural specification.

```plaintext
arch spec Other_System =

units N : Natural_Order;
C : Color;
F : Cont_Comp

result F[N] and F[C fit Elem ↦ RGB]
```
Several applications of the same generic component is different from applications of several generic components with similar specifications.

\[
\text{arch spec } \text{OTHER\_SYSTEM\_1 } = \\
\text{units } N : \text{NATURAL\_ORDER}; \\
C : \text{COLOR}; \\
FN : \text{NATURAL\_ORDER} \rightarrow \text{CONT} [\text{NATURAL\_ORDER}]; \\
FC : \text{COLOR} \rightarrow \text{CONT} [\text{COLOR} \text{ fit} \text{ Elem} \mapsto \text{RGB} ]
\]

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] → Set
  ∀e : Elem; C : Cont[Elem]
  • elements_of empty = empty
  • elements_of insert(e, C) = {e} ∪ elements_of C
end

arch spec Arch.Cont2Set.Nat =
units N : Natural_Order;
  C : Cont_Comp;
  S : Set_Comp;
result F [C [N]] [S [N]]
```
Open systems can be described by architectural specifications using generic unit expressions in the result part.

\[
\text{arch spec } \text{Arch}\_\text{Cont2Set} = \\
\text{units } C : \text{Cont\_Comp}; \\
S : \text{Set\_Comp}; \\
F : \text{Cont} [\text{Elem}] \times \text{Generated\_Set} [\text{Elem}] \rightarrow \text{Cont2Set} [\text{Elem}] \\
\text{result } \lambda X : \text{Elem} \cdot F [C [X]] [S [X]]
\]

\[
\text{arch spec } \text{Arch}\_\text{Cont2Set\_Used} = \\
\text{units } N : \text{Natural\_Order}; \\
CSF : \text{arch spec } \text{Arch}\_\text{Cont2Set} \\
\text{result } CSF [N]
\]
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] ↦ Cont[Nat]]
```
arch spec \texttt{Wrong_ARCH_SPEC} =

\textbf{units} \hspace{1em} \texttt{CN} : \texttt{CONT} \{\texttt{NATURAL_ORDER}\};
\texttt{SN} : \texttt{GENERATED_SET} \{\texttt{NATURAL_ORDER}\};
\texttt{F} : \texttt{CONT} \{\texttt{Elem}\} \times \texttt{GENERATED_SET} \{\texttt{Elem}\} \rightarrow \texttt{CONT2SET} \{\texttt{Elem}\};
\textbf{result} \hspace{1em} \texttt{F} \{\texttt{CN}\} \{\texttt{SN}\};

arch spec \texttt{Badly_Structured_ARCH_SPEC} =

\textbf{units} \hspace{1em} \texttt{N} : \texttt{NATURAL_ORDER};
\texttt{A} : \texttt{NATURAL_ORDER} \rightarrow \texttt{NATURAL_ARITHMETIC};
\texttt{C} : \texttt{CONT\_COMP};
\texttt{S} : \texttt{SET\_COMP};
\texttt{F} : \texttt{CONT} \{\texttt{Elem}\} \times \texttt{GENERATED_SET} \{\texttt{Elem}\} \rightarrow \texttt{CONT2SET} \{\texttt{Elem}\};
\textbf{result} \hspace{1em} \texttt{F} \{\texttt{C} \{\texttt{A} \{\texttt{N}\}\}\} \{\texttt{S} \{\texttt{A} \{\texttt{N}\}\}\};
Auxiliary unit definitions or local unit definitions may be used to avoid repetition of generic unit applications.

\[\text{arch spec } \textbf{Well-Structured_Arch_Spec} = \]

\[
\text{units} \quad N : \textbf{Natural_Order}; \\
A : \textbf{Natural_Order} \rightarrow \textbf{Natural_Arithmetic}; \\
AN = A[N]; \\
C : \textbf{Cont_Comp}; \\
S : \textbf{Set_Comp}; \\
F : \textbf{Cont}[\text{Elem}] \times \textbf{Generated_Set}[\text{Elem}] \rightarrow \textbf{Cont2Set}[\text{Elem}] \\
\text{result } F[C[AN]][S[AN]]\]
arch spec \texttt{ANOTHER\_WELL\_STRUCTURED\_ARCH\_SPEC} =

units \hspace{1em} N : \texttt{NATURAL\_ORDER};
\hspace{1em} A : \texttt{NATURAL\_ORDER} \to \texttt{NATURAL\_ARITHMETIC};
\hspace{1em} C : \texttt{CONT\_COMP};
\hspace{1em} S : \texttt{SET\_COMP};
\hspace{1em} F : \texttt{CONT \{Elem\}} \times \texttt{GENERATED\_SET \{Elem\}} \to \texttt{CONT2SET \{Elem\}}

result local \hspace{1em} AN = A [N] \texttt{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.

```plaintext
library UserManual/Examples

... spec NATURAL = ...

... spec NATURAL_Order = NATURAL then ...
```

© Michel Bidoit, Peter D. Mosses, CoFI
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 = ...

...
spec Partial_Order = Strict_Partial_Order then ...\leq...

...
spec Generated_Set [sort Elem] = ... \cup ...

...
spec Integer_Arithmetic_Order = ... \leq ... \geq ...

...
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:

\[
\text{%prec}\ \{\text{--+--, ----}\} < \{\text{--*--}\}
\]

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

\[
\text{%left_assoc}\ \{\text{--+--}, \text{--*--}\}
\]

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 \textit{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 = ...

... spec INTEGER_ARITHMETIC_ORDER = ...

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

```plaintext
library Basic/Numbers

...%
%left_assoc __@@__
%number __@@__
%floating __:::, __E__
%prec {__E__} < {__:::__}

...

spec NAT =
  free type Nat ::= 0 | suc(Nat)
  ...

ops 1 : Nat = suc(0); ...; 9 : Nat = suc(8);
    __@@__(m, n : Nat) : Nat = (m * suc(9)) + n
  ...
```

© Michel Bidoit, Peter D. Mosses, CoFI 151
CASL Tutorial
spec \textbf{Int} = \textbf{Nat} \textbf{then} \ldots

spec \textbf{Rat} = \textbf{Int} \textbf{then} \ldots

spec \textbf{DecimalFraction} = \textbf{Rat} \textbf{then} \ldots

\ldots

\begin{verbatim}
ops _:::_ : Nat \times Nat \rightarrow Rat;
\_E\_ : Rat \times Int \rightarrow Rat
\end{verbatim}

\ldots

\begin{itemize}
\item Libraries should include appropriate annotations.
\end{itemize}
Libraries can include items downloaded from other libraries.

```plaintext
library Basic/StructuredDatatypes
...
from Basic/Numbers get Nat, Int
...
spec List [sort Elem] given Nat = ...
...
spec Array ... given Int = ...
...
```

```plaintext
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.

```plaintext
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 = ...

...
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.