12

Basic Libraries

Till Mossakowski

The CASL Basic Libraries contain the standard datatypes.

The CASL Basic Libraries consist of specifications of often-needed datatypes and views between them, freeing the specifier from re-inventing well-known things. This can be compared to standard libraries in programming languages. While this book often discusses several styles of specification with CASL, the basic datatypes consistently follow a specific style described in [20].

Here we show a cut-down version without axioms.

Here, we describe two of the libraries (see the overview in Fig. 12.1): the libraries of numbers and of structured datatypes. We also provide stripped-down versions of the libraries themselves, with some of the specifications and all axioms and annotations removed. These stripped-down versions can serve for getting a first overview of the signatures of the specified datatypes.

The full CASL Basic Libraries with complete specifications is presented in the CASL Reference Manual [20], and is also included in the CD-ROM coming with this volume. The latest version is available at:

`http://www.cofi.info/Libraries`

HETS can be used to get an overview of the Basic Libraries.

The HETS tool described in Chap. 11 allows the structure of the specifications in the libraries to be displayed as a graph and their signatures to be inspected. This is recommended as a way of obtaining a better overview, and also for answering specific questions that arise when using the basic datatypes.

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 143–154, 2004.

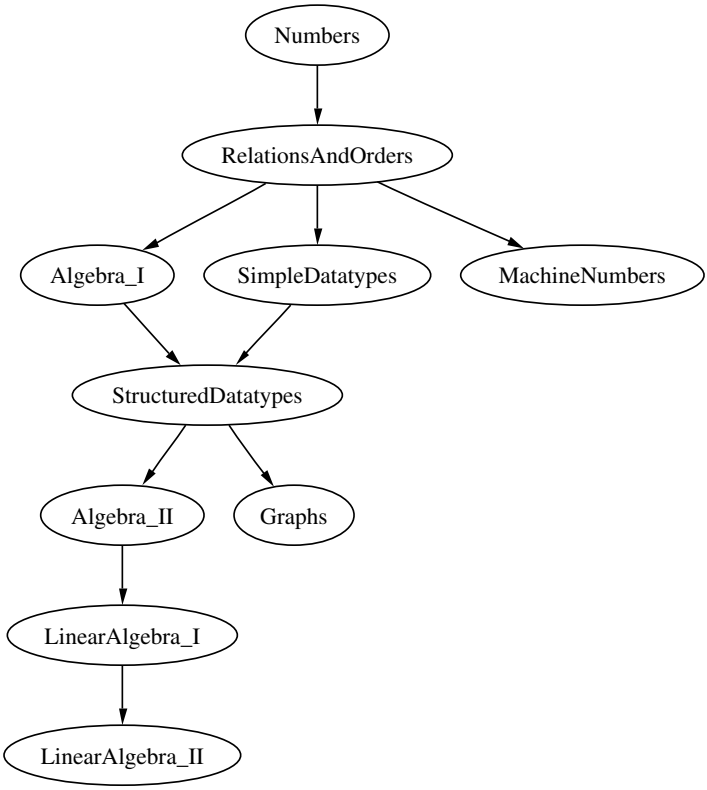

Fig. 12.1. Dependency graph of the libraries of basic datatypes.

12.1 Library Basic/Numbers

This library provides specifications of natural numbers, integers, and rational numbers.

> *The natural numbers are specified as a free datatype.*

In the specification Nat, the natural numbers are specified as a free datatype, together with a collection of predicates and operations over the sort *Nat* of natural numbers.

Note that the names for the *partial operations* subtraction _−?_ and division _/?_ include a question mark. This is to avoid overloading with the *total operations* _ − _ on integers and _/_ on rationals, which would lead to inconsistencies as both these specifications import the specification Nat. The *total operation* for subtraction differs from subtraction on the integers as

well. It is written $\_\_-!\_\_$, and it is *0* whenever the partial subtraction $-?$ is undefined, while it otherwise coincides with the latter.

The digits are introduced as constants, together with an operation $\_\_@@\_$ for concatenation of digits. Together with an annotation (see Chap. 9):

%number $\_@@\_$

this allows one to write the usual literals (like e.g. 8364) for natural numbers.

The introduction of the subsort *Pos*, consisting of the positive naturals, gives rise to certain new operations, e.g.:

$\_\_ \times \_ : Pos \times Pos \to Pos$,

whose semantics is completely determined by overloading.

library Basic/Numbers

spec Nat =

free type $Nat ::= 0 \mid suc(pre{:}?Nat)$

preds $\_\_\leq\_\_,\ \_\geq\_,\ \_\_<\_\_,\ \_>\_ : Nat \times Nat;$
$even,\ odd : Nat$

ops $\_\_! : Nat \to Nat;$
$\_\_+\_\_,\ \_\times\_,\ \_\_\hat{}\_,\ min,\ max,\ \_\_-!\_\_ : Nat \times Nat \to Nat;$
$\_\_-?\_,\ \_\_/?\_,\ \_div\_,\ \_mod\_\_,\ gcd : Nat \times Nat \to ?\ Nat$
%% Operations to represent natural numbers with digits:

ops
$1{:}\ Nat = suc(0);$ %(1_def_Nat)%
$2{:}\ Nat = suc(1);$ %(2_def_Nat)%
$3{:}\ Nat = suc(2);$ %(3_def_Nat)%
$4{:}\ Nat = suc(3);$ %(4_def_Nat)%
$5{:}\ Nat = suc(4);$ %(5_def_Nat)%
$6{:}\ Nat = suc(5);$ %(6_def_Nat)%
$7{:}\ Nat = suc(6);$ %(7_def_Nat)%
$8{:}\ Nat = suc(7);$ %(8_def_Nat)%
$9{:}\ Nat = suc(8);$ %(9_def_Nat)%
$\_\_@@\_(m{:}\ Nat;\ n{:}\ Nat){:}\ Nat = (m \times suc(9)) + n$
%(decimal_def)%

sort $Pos = \{p{:}\ Nat \bullet p > 0\}$

ops $1{:}\ Pos = suc(0);$ %(1_as_Pos_def)%
$\_\_\times\_\_ : Pos \times Pos \to Pos;$
$\_\_+\_\_ : Pos \times Nat \to Pos;$
$\_\_+\_\_ : Nat \times Pos \to Pos;$
$suc : Nat \to Pos$

end

The integers are specified as difference pairs of naturals.

The specification INT of integers is built on top of the specification NAT: integers are defined as equivalence classes of pairs of naturals written as differences — the axioms (which are omitted in the specification below) specify that two pairs are equivalent if their differences are equal:

$\forall\ a,\ b,\ c,\ d$: *Nat*
• $a - b = c - d \Leftrightarrow a + d = c + b$ %(equality_Int)%

The sort *Nat* is then declared to be a subsort of *Int*. Besides the division operator $\_\_/?\_\_$, the specification INT also provides the function pairs *div*/*mod* and *quot*/*rem*, respectively, as constructs for division — behaving differently on negative numbers, see [20] for a discussion. The operation *sign* gives the sign of an integer (which is either -1, 0, or 1).

spec INT =
NAT
then generated type $Int ::= \_-\_(Nat; Nat)$
sort $Nat < Int$
%% a system of representatives for sort Int is
%% a - 0 and 0 - p, where a: Nat and p: Pos
preds $\_\_\leq\_\_,\ \_\geq\_,\ \_\_<\_\_,\ \_>\_ : Int \times Int$;
$even,\ odd : Int$
ops $-\_\_,\ sign : Int \to Int$;
$abs : Int \to Nat$;
$\_\_+\_\_,\ \_\times\_,\ \_\_-\_\_,\ min,\ max : Int \times Int \to Int$;
$\_\_\hat{}\_\_ : Int \times Nat \to Int$;
$\_\_/?\_\_,\ \_\_div\_,\ \_\_quot\_\_,\ \_rem\_ : Int \times Int \to? Int$;
$\_\_mod\_ : Int \times Int \to? Nat$
end

The rationals are specified as fractions of integers.

The specification RAT of rational numbers follows the same scheme as the specification of integers discussed above. This time, the specification INT is imported. The rationals are then defined as equivalence classes of pairs consisting of an integer and a positive number written as fractions, using the axiom:

$\forall\ i,\ j$: *Int*; $p,\ q$: *Pos*
• $i\ /\ p = j\ /\ q \Leftrightarrow i \times q = j \times p$ %(equality_Rat)%

The sort *Int* is then declared to be a subsort of *Rat*. Note that thanks to the behavior of subsorted overloading in CASL, the declaration of the operation:

$\_\_/\_ : Rat \times Rat \to ? \; Rat;$

allows rationals to be written also as x/y, for arbitrary integers x and $y \neq 0$.

spec RAT =
INT
then generated type $Rat ::= \_\_/\_(Int; Pos)$
sort $Int < Rat$
preds $\_\_\leq\_\_,\ \_<\_,\ \_\geq\_\_,\ \_>\_ : Rat \times Rat$
ops $-\_\_,\ abs : Rat \to Rat;$
$\_\_+\_\_,\ \_-\_,\ \_\_\times\_\_,\ min,\ max : Rat \times Rat \to Rat;$
$\_\_/\_ : Rat \times Rat \to ? \; Rat;$
$\_\_\hat{}\_\_ : Rat \times Int \to Rat$
end

12.2 Library BASIC/STRUCTUREDDATATYPES

This library provides specifications of the familiar structured datatypes as used e.g. for the design of algorithms or within programming languages. Its main focus is data structures like (finite) sets, lists, strings, (finite) maps, (finite) bags, arrays, and various kinds of trees. Common to all these concepts is that they are generic. Consequently, all of the specifications of this library are generic.

Finite sets, maps and bags are specified as generated datatypes, with equality determined by means of observers.

Finite sets, finite maps and finite bags are specified using a generated sort. An observer operation or predicate is then introduced in order to define equality on this sort. Concerning finite sets, equality on the sort $Set[Elem]$ is characterized using the predicate $\_eps\_\_$ (displayed as ϵ) in the specification GENERATESET. This leads to the extensionality axiom:

- $M = N \Leftrightarrow \forall x : Elem \bullet x \in M \Leftrightarrow x \in N$ %(*equality_Set*)%

library BASIC/STRUCTUREDDATATYPES

spec GENERATESET [**sort** *Elem*] =
  generated type $Set[Elem] ::= \{\} \mid \_+\_(Set[Elem]; Elem)$
  pred $\_\_\epsilon\_\_ : Elem \times Set[Elem]$
    %% a system of representatives for sort Set[Elem] is
    %%
    %% {} and {} + x_1 + x_2 + ... + x_n
    %%
    %% where x_1 < x_2 < ... < x_n, n >= 1, x_i of type Elem,
    %% for an arbitrary strict total order _<_ on Elem.
end

spec SET [**sort** *Elem*] **given** NAT =
  GENERATESET [**sort** *Elem*]
then %def
  preds $isNonEmpty : Set[Elem]$;
    $\_\_\subseteq\_\_ : Set[Elem] \times Set[Elem]$
  ops $\{\_\} : Elem \to Set[Elem]$;
    $\sharp\_\_ : Set[Elem] \to Nat$;
    $\_\_+\_\_ : Elem \times Set[Elem] \to Set[Elem]$;
    $\_\_-\_\_ : Set[Elem] \times Elem \to Set[Elem]$;
    $\_\_\cap\_\_$, $\_\cup\_$, $\_\_-\_\_$, $\_symDiff\_\_$:
      $Set[Elem] \times Set[Elem] \to Set[Elem]$
end

Finite maps, i.e. elements of the sort $Map[S, T]$, are considered to be identical if looking up any value in S yields the same result in both cases:

- $M = N \Leftrightarrow \forall s : S \bullet lookup(s, M) = lookup(s, N)$ %(*equality_Map*)%

On top of this, the specification MAP adds e.g. predicates for elementhood (ϵ) as well as for determining the profile of a map ($f :: x \to y$ means that f is a map from x to y).

The specification TOTALMAP restricts maps to everywhere-defined maps (which are isomorphic to tuples). Since maps are finite, totality is only possible for maps over finite argument sorts. The latter is specified in the specification FINITE, using a partial surjection from the natural numbers. Since this specification is rather unusual, we make an exception and also show its axioms.

spec GENERATEMAP [**sort** S] [**sort** T] =
  generated type $Map[S,T] ::= empty \mid \_[\_\_/\_](Map[S,T]; T; S)$
  op $lookup : S \times Map[S,T] \to ? \; T$
end

spec Map [**sort** S] [**sort** T] **given** Nat =
  GenerateMap [**sort** S] [**sort** T]
and Set [**sort** S]
and Set [**sort** T]
then %**def**
  free type $Entry[S,T] ::= [\_\_/\_\_](target{:}T; source{:}S)$
  preds $isEmpty : Map[S,T]$;
    $\_\_\epsilon\_\_ : Entry[S,T] \times Map[S,T]$;
    $\_\_::\_\_\rightarrow\_\_ : Map[S,T] \times Set[S] \times Set[T]$
  ops $\_\_+\_\_,\ \_-\_ : Map[S,T] \times Entry[S,T] \rightarrow Map[S,T]$;
    $\_\_-\_\_ : Map[S,T] \times S \rightarrow Map[S,T]$;
    $\_\_-\_\_ : Map[S,T] \times T \rightarrow Map[S,T]$;
    $dom : Map[S,T] \rightarrow Set[S]$;
    $range : Map[S,T] \rightarrow Set[T]$;
    $\_\_\cup\_\_ : Map[S,T] \times Map[S,T] \rightarrow ? Map[S,T]$
end
spec Finite [**sort** $Elem$] =
  { Nat
  then op $f : Nat \rightarrow ? Elem$
    • $\forall\ x{:}\ Elem \bullet \exists\ n{:}\ Nat \bullet f(n) = x$ %(f_surjective)%
    • $\exists\ n{:}\ Nat \bullet \forall\ m{:}\ Nat \bullet def\ f(m) \Rightarrow m < n$ %(f_bounded)%
  }
  reveal $Elem$
end

spec TotalMap [Finite [**sort** S]] [**sort** T] =
  { Map [**sort** S] [**sort** T]
  then sort $TotalMap[S,T] =$
      $\{M{:}\ Map[S,T] \bullet \forall\ x{:}\ S \bullet def\ lookup(x, M)\}$
    ops $\_\_[\_\_/\_\_] : TotalMap[S,T] \times T \times S \rightarrow TotalMap[S,T]$;
      $lookup : S \times TotalMap[S,T] \rightarrow T$;
      $\_\_+\_\_ : TotalMap[S,T] \times Entry[S,T] \rightarrow TotalMap[S,T]$;
      $range : TotalMap[S,T] \rightarrow Set[T]$;
      $\_\_\cup\_\_ : TotalMap[S,T] \times TotalMap[S,T] \rightarrow ? TotalMap[S,T]$
    pred $\_\_\epsilon\_\_ : Entry[S,T] \times TotalMap[S,T]$
  }
  hide $Map[S,T]$
end

In the specification GenerateBag, those elements of sort $Bag[Elem]$ are identified that show the same number of occurrences (observed by the operation *freq*) for all entries:

• $M = N \Leftrightarrow \forall x : Elem \bullet freq(M, x) = freq(N, x)$ %(*equality_Bag*)%

spec GENERATEBAG [**sort** *Elem*] **given** NAT =
  generated type $Bag[Elem] ::= \{\} \mid \_+\_(Bag[Elem]; Elem)$
  op $freq : Bag[Elem] \times Elem \to Nat$
end

spec BAG [**sort** *Elem*] **given** NAT =
  GENERATEBAG [**sort** *Elem*]
then preds $isEmpty : Bag[Elem]$;
  $\_\_\in\_\_ : Elem \times Bag[Elem]$;
  $\_\_\subseteq\_\_ : Bag[Elem] \times Bag[Elem]$
  ops $\_\_+\_\_ : Elem \times Bag[Elem] \to Bag[Elem]$;
  $\_\_-\_\_ : Bag[Elem] \times Elem \to Bag[Elem]$;
  $\_\_-\_\_,\ \_\cup\_\_,\ \_\_\cap\_\_ : Bag[Elem] \times Bag[Elem] \to Bag[Elem]$
end

Lists are specified as a free datatype.

In the specification GENERATELIST, lists are built up from the empty list by adding elements in front. The usual list operations are provided: *first* and *last* select the first or last element of a list, while *rest* or *front* select the remaining list; $\#\_\_$ counts the number of elements in a list, while *freq* counts the number of occurrences of a given element; *take* takes the first n elements of a list, while *drop* drops them.

spec GENERATELIST [**sort** *Elem*] =
  free type $List[Elem] ::= [] \mid \_::\_(first{:}?Elem; rest{:}?List[Elem])$
end

spec LIST [**sort** *Elem*] **given** NAT =
  GENERATELIST [**sort** *Elem*]
then preds $isEmpty : List[Elem]$;
  $\_\_\in\_\_ : Elem \times List[Elem]$
  ops $\_\_+\_\_ : List[Elem] \times Elem \to List[Elem]$;
  $first,\ last : List[Elem] \to ?\ Elem$;
  $front,\ rest : List[Elem] \to ?\ List[Elem]$;
  $\sharp\_\_ : List[Elem] \to Nat$;
  $\_\_++\_ : List[Elem] \times List[Elem] \to List[Elem]$;
  $reverse : List[Elem] \to List[Elem]$;
  $\_\_!\_ : List[Elem] \times Nat \to ?\ Elem$;
  $take,\ drop : Nat \times List[Elem] \to ?\ List[Elem]$;
  $freq : List[Elem] \times Elem \to Nat$
end

Arrays are specified as certain finite maps.

The specification Array includes the condition $min \leq max$ as an axiom in its first parameter. This ensures a non-empty index set. Arrays are defined as finite maps from the sort *Index* to the sort *Elem*, where the typical array operations lookup (__!__) and assignment (__!__ := __) are introduced in terms of finite map operations. Finally, revealing the essential signature elements yields the desired datatype.

```
spec Array [ops min, max : Int • min ≤ max %(Cond_nonEmptyIndex)%]
           [sort Elem]
     given Int =
     sort  Index = {i: Int • min ≤ i ∧ i ≤ max}
then {     Map [sort Index] [sort Elem]
           with sort Map[Index,Elem] ↦ Array[Elem],
                op empty ↦ init
      then ops  __!__:=__ : Array[Elem] × Index × Elem → Array[Elem];
                __!__ : Array[Elem] × Index →? Elem
     }
     reveal sort Array[Elem], ops init, _!_, _!_:=_
end
```

Several kinds of tree are available, differing in the branching and in the positions of elements.

The library concludes with several specifications concerning trees. There are specifications of binary trees (BinTree, BinTree2), k-branching trees (KTree), and trees with possibly different branching at each node (NTree). Each of these branching structures can be equipped with data in different ways: Either all nodes of a tree carry data (as is the case in BinTree, KTree, and NTree), or just the leaves of a tree have a data entry (as in BinTree2).

Binary trees admit two children for each internal node.

```
spec GenerateBinTree [sort Elem] =
    free type
        BinTree[Elem] ::= nil
                        | binTree(entry:?Elem; left:?BinTree[Elem];
                                  right:?BinTree[Elem])
end

spec BinTree [sort Elem] given Nat =
    GenerateBinTree [sort Elem] and Set [sort Elem]
then preds isEmpty, isLeaf : BinTree[Elem];
          isCompoundTree : BinTree[Elem];
          __∈__ : Elem × BinTree[Elem]
    ops   height : BinTree[Elem] → Nat;
          leaves : BinTree[Elem] → Set[Elem]
end

spec GenerateBinTree2 [sort Elem] =
    free type NonEmptyBinTree2[Elem] ::=
                  leaf(entry:?Elem)
                | binTree(left:?NonEmptyBinTree2[Elem];
                          right:?NonEmptyBinTree2[Elem])
    free type BinTree2[Elem] ::= nil | sort NonEmptyBinTree2[Elem]
end

spec BinTree2 [sort Elem] given Nat =
    GenerateBinTree2 [sort Elem] and Set [sort Elem]
then %def
    preds isEmpty, isLeaf : BinTree2[Elem];
          isCompoundTree : BinTree2[Elem];
          __∈__ : Elem × BinTree2[Elem]
    ops   height : BinTree2[Elem] → Nat;
          leaves : BinTree2[Elem] → Set[Elem]
end
```

k-trees admit k children for each internal node (with k fixed).

We now come to k-branching trees. The branching is specified by using arrays of trees of size k, which are used to contain the children of a node in the tree.

```
spec GenerateKTree [op k : Int • k ≥ 1 %(Cond_nonEmptyBranching)%]
                   [sort Elem] given Int =
    Array [ops 1 : Int; k : Int
               fit ops min : Int ↦ 1,  max : Int ↦ k]
          [sort KTree[k,Elem]]
then free type
        KTree[k,Elem] ::= nil
                        | kTree(entry:?Elem;
                                branches:?Array[KTree[k,Elem]])
end

spec KTree [op k : Int • k ≥ 1          %(Cond_nonEmptyBranching)%]
           [sort Elem]
    given Int =
    GenerateKTree [op k : Int] [sort Elem]
and Set [sort Elem]
then %def
    preds isEmpty, isLeaf : KTree[k,Elem];
          isCompoundTree : KTree[k,Elem];
          __∈__ : Elem × KTree[k,Elem]
    ops   height : KTree[k,Elem] → Nat;
          maxHeight : Index × Array[KTree[k,Elem]] → Nat;
          leaves : KTree[k,Elem] → Set[Elem];
          allLeaves : Index × Array[KTree[k,Elem]] → Set[Elem]
end
```

n-trees admit arbitrary branching.

Finally, n-trees are trees with possibly different branching at each node. This is specified by equipping each node in a tree with a list of child trees.

```
spec GenerateNTree [sort Elem] =
      List [sort NTree[Elem]]
then free type
          NTree[Elem] ::= nil
                        | nTree(entry:?Elem;
                                branches:?List[NTree[Elem]])
end

spec NTree [sort Elem] given Nat =
      GenerateNTree [sort Elem] and Set [sort Elem]
then preds isEmpty, isLeaf : NTree[Elem];
           isCompoundTree : NTree[Elem];
           __∈__ : Elem × NTree[Elem]
      ops  height : NTree[Elem] → Nat;
           maxHeight : List[NTree[Elem]] → Nat;
           leaves : NTree[Elem] → Set[Elem];
           allLeaves : List[NTree[Elem]] → Set[Elem]
end
```