Methodological Guidelines

CoFI Note: M-6
Version: 0.7 June 2002

Markus Roggenbach Till Mossakowski
E-mail address for comments: cofi@informatik.uni-bremen.de
CoFI: The Common Framework Initiative
http://www.brics.dk/Projects/CoFI

Abstract

This note provides a style guide how to write specifications in CASL. The guidelines presented here were developed writing the note "Basic Datatypes in CASL" [4]. Thus they have proven to be useful in practice.

The aim of this note is twofold: on the one hand it documents the methodology behind the "Basic Datatypes in CASL" [4] and thus helps to understand the design decision behind them. On the other hand the here presented "Methodological Guidelines" may be useful as a starting point for other methodologies.

[1]`ftp://ftp.brics.dk/Projects/CoFI/Notes/M-6/`

Version History

This note revises the note M-6 “Basic Datatypes in CASL”, version 0.1, March 1999, and version 0.2, July 1999. From version 0.3 on, this note splits up into two parts: this version 0.7 of note M-6 “Methodological Guidelines” is devoted to methodological aspects, while note L-12 “Basic Datatypes in CASL” includes the Basic Datatypes proper.

Contents

Introduction

The basic datatypes of [4] are written obeying the following methodological guidelines. We formulate them explicitly for several reasons:

1. Many design decisions in the basic datatypes make only sense in context of the guidelines.
2. The guidelines are useful in general for writing specifications in Casl.
3. The guidelines can serve as a starting point for a new set of methodological guidelines for other applications.

For the guidelines we adopt the style of the book "A Pattern Language" [1]. Reflecting the marking of [1], all of our guidelines should be marked with one asterisk. I.e. with the words of [1] we claim that "we have made some progress", "but with careful work it will certainly be possible to improve on the solution" – not astonishing, as these guidelines are the first "style-guide" how to write specifications in Casl.

If we had pointed out in the guidelines themselves that they are rules of thumb, we would have had to stress phrasings like "if possible", "if adequate", "whenever possible" too much. Thus we formulate them as general statements. Examples from the basic datatypes illustrate their use. The discussion of a guideline justifies the underlying design decision and – as there is no rule without a meaningful exception – shows its limitations.

At certain points, e.g. the naming scheme of axioms, this note differs from the current version 0.7 of the Basic Datatypes. The next revision of the Basic Datatypes will take care of all guidelines presented in this note.

We propose two new annotations in this note: %**mono** and %**implied**. Their purpose is explained on pages 16 and 24, respectively. We freely use them throughout the note.

1 Naming Conventions

Naming schemes are both: tedious and important. Nobody likes to study them, but without an underlying naming scheme libraries as the "Basic Datatypes" are unreadable. The following guidelines allow for "typing by name". Special care has been taken to design the naming conventions for axiom labels. Here, the label name describes the role an axiom plays within the context of a certain specification. Thus, the specifier can document the intention behind a certain axiom by qualifying its label. These qualifications might even be useful for Casl tools.

1.1 General Guidelines

This section provides guidelines how to build names for arbitrary CASL identifiers. It also suggests naming conventions for predicates.

1.1.1 First Letter in a Name

Sort names should begin with a capital letter.
Predicate names should begin with a small letter.
Operation names should begin with a small letter.
Specification names should begin with a capital letter.
View names should begin with a capital letter.

Discussion: Only in the specification BOOLEAN we do not respect this guideline: the reason is that '*true*', '*false*', '*and*' etc. are reserved words in CASL.

1.1.2 Capitalization in Names, Underscore

Distinct parts of a name should begin with a capital letter.
The underscore should only be used within
***view** names or axiom labels.*

Examples:

spec CHAR =
  ...
  ops $'\backslash 000' : Char = chr(0)$; %(slash_000)%
  preds *isLetter*, *isDigit*, *isPrintable* : *Char*
end

view COMMUTATIVEFIELD_IN_RAT: ... **end**

1.1.3 Prefixes for Predicates

Use the prefix "is/are" or "has/have" for predicate names.

Examples:

spec BAG [**sort** *Elem*] =
  ...
  pred *isEmpty* : *Bag*

op *empty* : *Bag*
...
end

spec EXTCOMMUTATIVERING [COMMUTATIVERING] **given** INT =
ExtRing[RING]
then
preds *hasNoZeroDivisors* : ();
• *hasNoZeroDivisors* $\Leftrightarrow$
$\forall x, y : Elem \bullet (x * y = 0 \Rightarrow x = 0 \vee y = 0)$

...
end

Discussion: This guideline helps to identify predicates by means of speaking names. The specification BAG shows how to avoid confusion with an operation name. But it can be useful in also in a wider context, c.f. the specification EXTCOMMUTATIVERING.

1.2 Naming of Axioms

The following guidelines provide an extensive naming scheme for axioms ensuring a clear and intuitive labelling: Unique labels are necessary as references in proofs, suggestive names relate the labels closely with the axioms.

1.2.1 Structure of labels

Labels may consist of the components (in this order):
1. *the name of the predicate/operation/property, which is defined,*
2. *a qualification,*
3. *a sort name, and*
4. *the (abbreviated) name of the specification they belong to.*

Examples:
spec NAT =
...
ops $1 : Nat = suc(0)$; %(1_def_Nat)%
$\forall m, n : Nat$
• $0 \leq n$ %(leq_def1_Nat)%
• $not(suc(n) \leq 0)$ %(leq_def2_Nat)%
• $suc(m) \leq suc(n) \Leftrightarrow m \leq n$ %(leq_def3_Nat)%
• $0! = 1$ %(factorial_0)%

```
  • suc(n)! = suc(n) * n!                              %(factorial_suc)%
then %mono
  sort Pos = {p : Nat • p > 0}
  op 1 : Pos = suc(0);                                 %(1_def_Pos)%
end

spec ReflexiveRelation =
  Relation
then
  ∀x : Elem
  • x ∼ x                                              %(refl)%
end
```

Discussion: The above part of the specification Nat shows axiom labels of different form: %(leq_def1_Nat)% consists of the predicate name 'leq', a qualification 'def', and the specification name 'Nat'; here, the specification name has been added as other specifications like the integers certainly will have their own predicate 'leq' with their own defining axioms. %(factorial_0)% is built from the operation name 'factorial' and the qualification '0'; in this case the operation factorial is expected to be related only with the naturals. Finally, the labels %(1_def_Nat)% and %(1_def_Pos)% show disambiguation by the sort names 'Nat' and 'Pos', resp.

ReflexiveRelation gives an example of an axiom's label denoting an abstract property.

Reference: See **Qualifications**, subsection 1.2.3, **Labels for Overloaded Definitions**, subsection 1.2.7, and **Labels for Universal Axioms**, subsection 1.2.9.

1.2.2 Capitalization in Labels, Underscore

The guideline concerning capitalization of the first letter of sort names, predicate names, operation names, specification names holds within labels as well. The components of a label are separated by an underscore "_".

Reference: See **First Letter in a Name**, subsection 1.1.1, and **Capitalization in Names, Underscore**, subsection 1.1.2.

1.2.3 Qualifications

An axiom qualification *is of the form*
"_def" (cf. subsection 1.2.4),
a constructor name (cf. subsection 1.2.5),
a number (cf. subsection 1.2.6),
"_partial", "_total" (cf. subsection 1.2.7), or
"_dom" (cf. subsection 1.2.8).

1.2.4 Labels for Definitional Axioms

All labels on axioms defining an operation or predicate
should have the qualification "def".

Example:
spec NAT =
  . . .
  op $min : Nat \times Nat \to Nat$
  $\forall m, n : Nat$
  • $min(m, n) = m$ *when* $m \le n$ *else* n %(min_def_Nat)%
end

1.2.5 Labels for Inductive Definitions

For inductive definitions,
the names of the respective constructors
should be used as qualifications instead of "def".

Example for constructors:
spec NAT =
  . . .
  op $\_\_ + \_\_ : Nat \times Nat \to Nat$
  $\forall m, n : Nat$
  • $0 + m = m$ %(add_0_Nat)%
  • $suc(n) + m = suc(n + m)$ %(add_suc_Nat)%
end

1.2.6 Labels for Case Distinctions

Case distinctions should just be numbered,
if there is no better name.

Example:
spec BOOLEAN =
  . . .
  ops $\_\_And\_\_ : Boolean \times Boolean \rightarrow Boolean$
  . . .
  • $False\ And\ False = False$ %(And_def1)%
  • $False\ And\ True = False$ %(And_def2)%
  . . .
end

1.2.7 Labels for Overloaded Definitions

Disambiguation of overloaded names should be done by sort names and/or the qualifications "total" and "partial".

Example for Sort Names:
spec NAT =
  . . .
  op $min : Nat \times Nat \rightarrow Nat$
  $\forall m, n : Nat$
  • $min(m, n) = m\ when\ \ m \leq n\ else\ n$ %(min_def_Nat)%
end

spec INT =
  Nat
then
  . . .
  op $min : Int \times Int \rightarrow Int$
  $\forall x, y : Int$
  • $min(x, y) = x\ when\ \ x \leq y\ else\ y$ %(min_def_Int)%
end

Example for 'total' and 'partial':
spec NAT =
  . . .
  ops $\_\_div\_\_ : Nat \times Nat \rightarrow ?\ Nat;$
    $\_\_div\_\_ : Nat \times Pos \rightarrow Nat;$
  $\forall m, n, s : Nat; p : Pos$
  • $m\ div\ n = s\ \Leftrightarrow$ %(div_partial_Nat)%
    $(\exists r : Nat \bullet m = n * s + r \wedge 0 \leq r \wedge r < n)$
then %**implies**

$\forall m, n, s : Nat; p : Pos$
- $m\ div\ p = s \Leftrightarrow$ %(div_total_Nat)%
  $(\exists r : Nat \bullet m = p * s + r \wedge 0 \leq r \wedge r < p)$

end

1.2.8 Labels for Definedness Axioms

Definedness formulas for partial functions should be named by the qualification "dom".

spec NAT =
  ...
  $\forall m, n : Nat$
- $def\ (m\ div\ n) \Leftrightarrow \neg n = 0$ %(div_dom_Nat)%

end

1.2.9 Labels for Universal Axioms

Add the name of the specification, if an axiom can belong to many datatypes.

Example:
spec DEFINEBOOLEANALGEBRA =
  ...
  $\forall x, y, z : Elem$
- $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ %(distr1_DefBA)%
- $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ %(distr2_DefBA)%

end

Discussion: Theorem provers as e.g. Isabelle require unique labels for axioms. For some laws it is not possible do disambiguate the labels by adding the sort: the distributive laws hold e.g. in a boolean algebra and in a ring, and both specifications have the sort "*Elem*".

1.2.10 Generated labels

Do not use labels starting with "ga_", since these are be generated by the CASL *tool set.*

Discussion: The CASL tool set generates all the axioms implicitly expressed by specifications, such as associativity of operations attributed with **assoc**, or injectivity of constructors for **free types**. Since the syntax of labels is arbitrary, there is no chance to generate labels that the user cannot input.

2 Basic Specifications

Being the elementary building blocks of CASL specifications, basic specification should be carefully designed. In order to obain readability, the different kinds of CASL basic items should be arranged in a prescribed order. Signatures should be both: 'naturally' small and 'complete'. The combination of datatypes and subsorting needs to be dealt with. Finally, there are guidelines how to characterize the domain of partial functions.

2.1 Ordering Elements

Here we discuss how to arrange the different items of a basic specification – in general and for axioms. We also provide a guideline concerning variable declarations.

2.1.1 Order in a Basic Specification

Arrange a basic specification in the order
types ... ***sorts*** ... ***preds*** ... ***ops*** ... $\forall$ ....

Example:
```
spec CHAR =
...
then
  sort Char
  preds isLetter, isDigit, isPrintable : Char
  ops chr : Nat →? Char;
      ord : Char → Nat;
  ∀n : Nat; c : Char
  •  ...
end
```

Discussion: In the above guideline the keyword 'types' stands for datatype declarations, free datatye declarations, as well as sort generation.

This style allows to keep overview of specifications: There is a specific place for all kinds of information that might arise in a basic specification. Furthermore, collecting the signature elements in one place allows for 'blind' use – assuming that the specifiers got the axioms right, one is only interested in how rich the signature is which a specification provides. One might argue that the separation of the declaration of signature elements from the axioms describing their properties is a drawback of this style. But this does not become a serious problem if the axioms are ordered as described in guideline 2.1.2 and guideline 4.1.1 on structuring specifications in small pieces of code is obeyed.

Reference: See also **Order of Axioms**, subsection 2.1.2, and **Size of Single Specifications**, subsection 4.1.1.

2.1.2 Order of Axioms

Arrange the axioms in a local variable declaration in an order corresponding to the declaration of the **preds** *and* **ops**., *resp.*

spec NAT =
  . . .
  ops $\_\_! :$ $Nat \to Nat;$
      $\_\_ + \_\_, \_\_ * \_\_, \_\_\hat{}\_\_,$
      $min, max :$ $Nat \times Nat \to Nat;$
      . . .
  $\forall m, n, r : Nat; p : Pos$
  • $0! = 1$ %(factorial_0)%
  • $suc(n)! = suc(n) * n!$ %(factorial_suc)%
  • $0 + m = m$ %(add_0_Nat)%
  • $suc(n) + m = suc(n + m)$ %(add_suc_Nat)%
  . . .
end

2.1.3 Variable Declarations

Avoid global variable declarations.

Discussion: To avoid

- mismatch between the use of a variable in a formula and its declaration, and

- confusion for the reader of a specification,

a variable should be declared as local as possible. We reflect this guideline in the basic datatypes by using only local variable declarations.

2.2 Minimal Signatures

Often, there are many possibilities to specify the profile of a certain operation: Restricting the operation's arguments to suitable subsorts can lead to a more specific result or totalize a partial operation. We present guidelines how to select profiles.

2.2.1 Minimal Coverage

Provide a minimal coverage of the possible profiles of an operation.

Example:
spec NAT =
  . . .
  op $\_\_+\_\_ : Nat * Nat \to Nat$
  . . .
  op $1 : Nat = suc(0)$; %(1_def_Nat)%
  . . .
  ops $\_\_*\_\_ : Nat * Nat \to Nat,$
      $comm, assoc, unit\ 1$; %**implied**
then %**mono**
  sort $Pos = \{p : Nat \bullet p > 0\}$
  ops $1 : Pos = suc(0)$; %(*1_def_Pos*)%
    $\_\_*\_\_ : Pos * Pos \to Pos$;
    $\_\_+\_\_ : Pos * Nat \to Pos$;
    $\_\_+\_\_ : Nat * Pos \to Pos$;
    $suc : Nat \to Pos$
  . . .
end

Discussion: This guideline aims at small signatures allowing to infer as much type information as possible ('minimal'), i.e. for a given set of arguments the minimal target sort is determined – which also are general ('coverage'), i.e. no argument type is forgotten. Given an operation with several possible profiles the selection of the profiles provided by the specification should follow the two-step procedure below:

1. For any target sort of an operation take the most general argument profile (apply the subsort relation componentwise).
2. Optimize this set of profiles: if there are profiles with identical argument sorts and targets in subsort relation, take just the profile with minimal target.

For instance, given the subsort relation $Pos < Nat$, the operation $\_\_ + \_\_$ on the naturals has

(i) $Nat \times Nat \rightarrow Nat$,

(ii) $Pos \times Nat \rightarrow Pos$,

(iii) $Nat \times Pos \rightarrow Pos$, and

(iv) $Pos \times Pos \rightarrow Pos$

as possible profiles. Step 1. of the above procedure selects the profile (i) for the target sort *Nat*, and profiles (ii) and (iii) for the target sort *Pos*. There is no optimization possible according to step 2.

As another example consider the constant '1'. Following step 1. we obtain the profiles **op** *1* : *Nat* and **op** *1* : *Pos* . Applying step 2. yields that only the profile **op** *1* : *Pos* is necessary for minimal coverage. In spite of this the above specification NAT declares it also as **op** *1* : *Nat* . The constant '1' is needed before the subsort *Pos* is introduced, e.g. to formulate that '1' is the unit of multiplication.

2.2.2 Partial and Total Functions

Develop 'natural' signatures incorporating either the total or the partial variant of an operation.

Example: Prefer one of these specifcations

spec LISTPARTIAL [**sort** *Elem*] **given** NAT =
  . . .
  ops *first* : *List*[*Elem*] →? *Elem*;
  . . .
end

spec LISTTOTAL [**sort** *Elem*] **given** NAT =
  . . .
  ops *first* : *NeList*[*Elem*] → *Elem*;
  . . .
end

over the following:

spec LISTBOTH [**sort** *Elem*] **given** NAT =
  . . .
  ops *first* : *List*[*Elem*] →? *Elem*;
    first : *NeList*[*Elem*] → *Elem*
  . . .
end

Discussion The operation *first* can be specified as a partial operation on general lists. Introducing nonempty lists as a subsort allows to totalize this operation. Having both variants in one specification leads to too complicated signatures, as ususally there are several operations which can be totalized on suitable choosen subsorts. I.e. the signature grows in both components: operations and sorts.

2.3 Predicative subsorts

Define subsorts of datatypes afterwards predicatively,
not in the datatype definition itself.

Example: Prefer this style

spec GENERATENAT =
  free type *Nat* ::= *0* | *suc*(*pre* :?*Nat*)
  sort *Pos* = $\{n : Nat \bullet \neg n = 0\}$
end

over the following:

spec GENERATENAT =
  free types *Nat* ::= *0* | *sort Pos*;
    Pos ::= *suc*(*pre* : *Nat*)
end

Discussion: The first specification directly leads to the desired induction principle, which has to be proved rather clumsily with the second specification.

2.4 Domains of Partial Functions

If possible, the domain of a partial function should be characterized within a specification. Here we discuss how this can be achieved in a uniform way without running into consistency problems.

2.4.1 Implicit Characterization

Specify domains of partial functions implicitly by strong equations.
Add explicit characterizations as intended consequences.

Example:
spec NAT =
...
  ops $\_\_div\_\_ : Nat \times Nat \rightarrow ? \; Nat;$
  $\forall m, n, s : Nat$
  • $m \; div \; n = s \;\Leftrightarrow (\exists r : Nat \bullet m = n * s + r \wedge 0 \leq r \wedge r < n)$
%(div_partial_Nat)%
then %**implies**
  $\forall m, n : Nat$
  • $def \; (m \; div \; n) \Leftrightarrow \neg(n = 0)$ %(div_dom_Nat)%
end

Discussion: The definition of a partial function by a strong equation results in an implicit characterization of its domain. The definedness formula in the intended consequences provides useful information (if only for documentary purposes) and can be used for an additional correctness test of the specification.

2.4.2 Explicit Characterization

If – as an exception – you specify domains explicitly,
guard the axioms with the definedness assertion.

Example: (from the CoFI-Note M-7 [3])

spec DEFINEBASICREAL =
  RAT
then
  sort $Rat < Real$
  op $\_\_/\_\_ : Rat \times Rat \rightarrow ? \; Rat$
...
then
  %% Sequences and sequence combinators:
  sort *Sequence*
  pred $nonZero : Real$

ops $\_\_!\_\_ : \mathit{Sequence} \times \mathit{Nat} \to \mathit{Real};$
 %% $s!i$ denotes the i^{th} element of a sequence s *of* real numbers
 $\_\_/\_\_ : \mathit{Sequence} \times \mathit{Sequence} \to ?\ \mathit{Sequence};$
 %% divide sequences componentwise
$\forall n : \mathit{Nat}; r : \mathit{Real}; a, b : \mathit{Sequence}$
- $\mathit{nonZero}(r) \Leftrightarrow \neg r = \mathit{0}$ %(nonZero_def)%
- $\mathit{def}\ (a/b) \Leftrightarrow \forall k : \mathit{Nat} \bullet \mathit{nonZero}(b!k)$ %(div_dom_Seq)%
- $(a/b)!n = ((a!n)/(b!n))\ \mathit{if}\ \mathit{def}\ (a/b)$ %(div_def_Seq)%

end

Discussion: In this example, adding the equation

$$(a/b)!n = ((a!n)/(b!n))$$

instead of the axiom %(div_Seq_def)% would lead to an inconsistency. This is because $(a!n)/(b!n)$ may be defined or undefined, depending on n, while the definedness of a/b is independent of n. Thus, we are forced to specify the domain of $\_\_/\_\_$ explicitly. (In higher-order CASL, using the description operator, an implicit specification is possible.)

3 Requirement versus Design Specifications

The guidelines presented in this section depend on the type of a specification. We distinguish between requirement and design specifications in the following sense: *requirement specifications* are loose, i.e. the specifier conciously leaves certain properties unspecified. Thus, the specified datatypes are polymorphic and, as a consequence, the specification is 'incomplete': given a property in terms of a formula, neither this formula nor its negation might hold for the whole model class. In contrast to this, *design specifications* are complete: all design decision have been taken and the specified model class is monomorphic.

For parametrized specifications, this pictures becomes more involved. In general, CASL specification definitions are of the form

spec SN [SP$_1$] ... [SP$_n$] **given** SP"$_1$, ..., SP"$_m$ =
 SP
end

The annotation %**mono** shall express that the extension of

$$\{\text{SP"}_1 \textbf{ and } \ldots \textbf{ and } \text{SP"}_m \} \textbf{ then } \{\text{SP}_1 \textbf{ and } \ldots \textbf{ and } \text{SP}_n \}$$

to the specification

```
    {SP"_1 and ... and SP"_m } then {SP_1 and ... and SP_n }
then
    Sp
```

is unique up to isomorphism. In this case, we consider the specification SN as a design specification, otherwise as a requirement specification.

With these definitions, we obtain as guidelines concerning instantiations:

1. Instantiating the parameters of a requirement specification yields a requirement specification.
2. Instantiating the parameters of a design specification with design specifications yields a design specification.
3. Instantiating the parameters of a design specification with requirement specifications, which stem from a foramal parameter, yields a design specification. Otherwise, one obtains a requirement specification.

3.1 Paramters and Imports

3.1.1 Classification of Parameters

In a specification definition
a parameter is expected to be a requirement specification.

Discussion: As the parameter of a specification definition allows for instantiation, it should be a polymorphic datatype.

3.1.2 Classification of Imports

The import collects those parts of a specification
whose design is already been fixed in a named design specification.

Examples:

```
spec List [sort Elem] given Nat = %mono
  GenerateList [sort Elem]
then
  ...
end

spec MyRing [Ring] given Int =
  ExtRing [Ring]
then
```

op $\_\_/?\_\_ : \mathit{Elem} * \mathit{Elem} \to ?\ \mathit{Elem}$;
$\forall x, y, z : \mathit{Elem}$
• $(x/?y = z) \Rightarrow x = z * y$ %(divide_NonZero)%
end

Discussion: Following the above terminology, LIST is a design specification: for any kind of elements the list structure is fixed up to isomorphism. The natural numbers – used e.g. to determine the lenght of a list – are specified elsewhere and are therefore imported.

On the other hand, MYRING (not part of the Basic Datatypes) is a requirement specification: if the partial operation $\_\_/?\_\_$ is defined, it shall be the inverse of the usual ring multiplication. The integers are used for the power operation within EXTRING.

Collecting the natural numbers and the integers, resp., in the import clause indicates that

- their design is already fixed and that
- this is done elsewhere.

Furthermore, this style allows for the instantiation of the

- parameter with the import,

i.e. the parameter **sort** *Elem* can be instantiated with NAT, and the parameter RING can be instantiated with INT. It avoids non-wellformed specifications due to sharing symbols between actual parameter and body which do not belong to the formal parameter or import, c.f. section 4.3 **Parameterized specifications**.

3.2 Gathering Requirements

3.2.1 Use of the Operation Attributes [Requirement]

Use operation attributes only in requirement specifications.

Example:
spec CONTAINER [**sort** *Elem*] =
sort *Container*
ops *empty* : *Container*;
$\{\_\_\}$: $\mathit{Elem} \to \mathit{Container}$;
$\_\_ + +\_\_$: $\mathit{Container} \times \mathit{Container} \to \mathit{Container}$, *assoc*
end

Discussion: The concatenation of containers shall be associative, while it is left open if containers are later implemented as sets, lists, bags, etc.

In contrast, in a design specification, attributes should *follow* from the operational inductive definitions, while requiring these attributes would complicate consistency proofs. This means, in order to establish properties like *assoc*, *comm*, *idem*, and *unit* for an operator (or an operator and a constant, resp.) defined in an operational manner specify a **view**, see **Use of Views**, subsection 4.1.6.

3.2.2 Observers [Requirement]

Use observers to implicitely specify operations.

Example:

```
spec SORTING [TOTALORDER] =
{
       LIST [sort Elem]
  then
       preds is_ordered   : List[Elem];
             permutation  : List[Elem] × List[Elem];
       forall x, y        : Elem;
              L, L1, L2   : List[Elem]
       • is_ordered([])
       • is_ordered([x])
       • is_ordered(x :: (y :: L)) ⇔ x ≤ y ∧ is_ordered(y :: L)
       • permutation(L1, L2) ⇔ (∀ x : Elem • x ∈ L1 ⇔ x ∈ L2)
  then
       op sorter : List[Elem] → List[Elem];
       forall L : List[Elem]
       • is_ordered(sorter(L))
       • permutation(L, sorter(L))
  }    hide is_ordered, permutation
end
```

Discussion: Observers can be used to define properties of functions or predicates in an abstract way – separating requirements from design decisions. In the above example, a sorting function *sorter* is uniquely characterized in terms of the observers *is_ordered* and *permutation.* Thus, there is no need to specify *sorter* following the recursion scheme of a sorting algorithm like bubblesort or quicksort.

3.3 Datatypes

3.3.1 Definitions on Generated Datatypes [Design]

Use inductive definitions by cases in the context of generated datatypes:
Define predicates by equivalences.
Define operations by strong equations.
Both may be guarded by disjoint and jointly exhaustive conditions.

Example:
spec FINITESET [**sort** *Elem*] **given** NAT =
  ...
  $\forall x : \mathit{Elem}; S, T, U : \mathit{FinSet}[\mathit{Elem}]$
- $\{\} \subseteq S$ %(subset_empty)%
- $\mathit{set}(x) \subseteq S \Leftrightarrow x \;\epsilon\; S$ %(subset_set)%
- $(S \cup T) \subseteq U \Leftrightarrow S \subseteq U \wedge T \subseteq U$ %(subset_union)%

- $\{\} \cap S = \{\}$ %(intersect_empty)%
- $\mathit{set}(x) \cap S = \{\} \;\mathit{if}\; \neg\, x \;\epsilon\; S$ %(intersect_set1)%
- $\mathit{set}(x) \cap S = \mathit{set}(x) \;\mathit{if}\; x \;\epsilon\; S$ %(intersect_set2)%
- $(S \cup T) \cap U = (S \cap U) \cup (T \cap U)$ %(intersect_union)%

end

Discussion: It can be shown that following these principles, for free types one automatically gets a definitional extension. In the case of generated types, well-definedness w.r.t. the specified equality must be shown.

3.4 Supersorts

3.4.1 Extension to a Supersort [Requirement]

The extension of operations and predicates to a supersort
can be shortened by adding abstract properties.

Example:
spec COMPACTINTREQUIREMENT =
  INT
then
  free types $\mathit{PosInf} ::= \mathit{infinity};$
    $\mathit{NegInf} ::= -\_\_(\mathit{PosInf});$
    $\mathit{CompactInt} ::= \mathit{sort}\ \mathit{Int} \mid \mathit{sort}\ \mathit{PosInf} \mid \mathit{sort}\ \mathit{NegInf};$

  pred $\_\_ \leq \_\_ : CompactInt \times CompactInt$
  $\forall n : Int$
  • $-infinity \leq n$ %(order_infinity_def1)%
  • $n \leq infinity$ %(order_infinity_def2)%
then
  TotalOrder **with** $Elem \mapsto CompactInt$
  ...
end

Discussion: In the above example, a supersort *CompactInt* of sort *Int* is introduced, which consists besides the integers also of the values *infinity* and $-infinity$. The elements of *CompactInt* are required to be totally ordered by $\_\_ \leq \_\_ : CompactInt \times CompactInt$, such that $-infinity \leq n \leq infinity$, $n \in Int$, holds. These requirements are captured directly by the above specification. See section 3.4.2 for an equivalent, but less intuitive specification.

The problematic point here is that the specification of abstract properties on the new supersort might impose consequences on the already defined model class. Thus, contradictions easily arise and, consequently, consistency proofs are required. For instance, in our example the requirement

TotalOrder **with** $Elem \mapsto CompactInt$

enforces that also $\_\_ \leq \_\_ : Int \times Int$ is a total order – which is luckily the case.

3.4.2 Extension to a Supersort [Design]

Specify the extension of operations and predicates to a supersort only for those values that are not in the subsort.

Example:
spec CompactIntDesign =
  Int
then
  free types $PosInf ::= infinity;$
      $NegInf ::= -\_\_(PosInf);$
      $CompactInt ::= sort\ Int \mid sort\ PosInf \mid sort\ NegInf;$
  pred $\_\_ \leq \_\_ : CompactInt \times CompactInt$
  $\forall n : Int$
  • $n \leq infinity$ %(order_def1)%
  • $-infinity \leq n$ %(order_def2)%

- $\neg n \leq -infinity$ %(order_def3)%
- $\neg infinity \leq n$ %(order_def4)%
- $-infinity \leq infinity$ %(order_def5)%
- $\neg infinity \leq -infinity$ %(order_def6)%

...

end

Discussion: In the above example, a supersort *CompactInt* of sort *Int* is introduced, which consists besides the integers also of the values *infinity* and *−infinity*. The order $\_\_ \leq \_\_ : Int \times Int$ is extended to the supersort *CompactInt* by the axioms %(order_def1)%, ..., %(order_def6)%.

Compared to the requirement specification CompactIntRequirement of section 3.4.1, CompactIntDesign is

1. longer, and
2. it needs a proof that $\_\_ \leq \_\_ : CompactInt \times CompactInt$ is a total order.

But there are less consistency problems: As the axioms concern only the new predicate, the design specification is consistent if the specification Int is consistent and there is no contradiction between the new introduced axioms. I.e., the already finished design of Int is not influenced by the extension to the supersort.

4 Structured Specifications

Larger specifications should be structured into smaller parts in order to increase readability and also re-use. Casl has a number of language constructs allowing to write specifications in a structured way; we here demonstrate and explain their use. Parameterized and free specifications (and how to avoid some pitfalls when using them) are explained in separate subsections.

4.1 Dividing Specifications into Parts

4.1.1 Size of Single Specifications

Structure specifications in
small and easlily understandable subspecifications.

4.1.2 Separate Sort Generation

Treat sort generation in GENERATEDATATYPEX.
Deal with additive aspects in DATATYPEX.

Example:
spec GENERATEFINITESET [**sort** *Elem*] =
  generated type *FinSet*[*Elem*] ::= {} | __ + __(*Elem*; *FinSet*[*Elem*]);
then %**def**
  pred $\_\_\epsilon\_\_$: *Elem* ∗ *FinSet*[*Elem*];
  $\forall x, y : Elem; M, N : FinSet[Elem]$
  • $\neg\ x\epsilon\{\}$
  • $x\epsilon(y + M) \Leftrightarrow x = y \vee x\epsilon M$
then
  $\forall M, N : FinSet[Elem]$
  • $M = N \Leftrightarrow (\forall x : Elem \bullet x\epsilon M \Leftrightarrow x\epsilon N)$
end

spec FINITESET [ELEM **with sort** *Elem*] =
  GENERATEFINITESET [ELEM]
then
  . . .
end

Discussion: As generation of sorts is a rather subtle part of a specification, this style hopefully avoids confusion. Furthermore, this style allows to attach a representation theorem for the generated sorts to the specification GENERATEX – see also **Add System of Representatives**, subsection 4.1.3.

4.1.3 Add System of Representatives

Add your intended system of representatives
as a comment in GENERATEDATATYPEX.

Example:
spec GENERATEFINITESET [**sort** *Elem*] =
  generated type *FinSet*[*Elem*] ::= {} | __ + __(*Elem*; *FinSet*[*Elem*]);
then %**def**
  pred $\_\_\epsilon\_\_$: *Elem* ∗ *FinSet*[*Elem*];
  $\forall x, y : Elem; M, N : FinSet[Elem]$
  • $\neg\ x\epsilon\{\}$

  • $x\epsilon(y + M) \Leftrightarrow x = y \vee x\epsilon M$
then
  $\forall M, N : \mathit{FinSet}[\mathit{Elem}]$
  • $M = N \Leftrightarrow (\forall x : \mathit{Elem} \bullet x\epsilon M \Leftrightarrow x\epsilon N)$
%% intended system of representatives:
%% $x_1 + \ldots + x_n + \{\}$,
%% where $n \geq 0$, $x_i \in \mathit{Elem}$, $x_i \neq x_j$ for $i \neq j$, and
%% $x_1 < x_2 < \ldots < x_n$ (< is an arbitrary order on Elem)
end

Discussion: The system of representatives gives a canonical model. Hence, it increases the intuitive understanding of the specification, since readers can think about the specification in terms of the canonical model.

4.1.4 Qualify Extensions

If possible, qualify extensions by the semantical annotations "implied", "def", "mono", or "cons".

Example:
spec GENERATENAT = %**mono**
  free type $\mathit{Nat} ::= 0 \mid \mathit{suc}(\mathit{pre} :? \mathit{Nat})$
then %**def**
  op $\_\_ + \_\_ : \mathit{Nat} \times \mathit{Nat}$
  $\forall m, n : \mathit{Nat}$
  • $0 + m = m$ %(add_0_Nat)%
  • $\mathit{suc}(n) + m = \mathit{suc}(n + m)$ %(add_suc_Nat)%
then %**implies**
  op $\_\_ + \_\_ : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$, *assoc*, *comm*, *unit 0*
  $\forall m, n : \mathit{Nat}$
  • $\mathit{def}\ \mathit{pre}(n) \Leftrightarrow \mathit{not}\ n = 0$ %(dom_pre_Nat)%
  • $\mathit{def}\ \mathit{pre}(n) \Rightarrow \mathit{pre}(n) + m = \mathit{pre}(n + m)$ %(add_pre_Nat)%
then %**cons**
  op $\mathit{bound} : \mathit{Nat}$
end

Use "implied" instead of "implies" to state intended consequences of the specification closely belonging to the specification itself.

Example: Using %**implied**, the above example can be reformulated as follows.

spec GenerateNat = %**mono**
  free type $Nat ::= 0 \mid suc(pre :?Nat)$
  $\forall n : Nat$
  • $def\ pre(n) \Leftrightarrow not\ n = 0$ %(dom_pre_Nat)% %**implied**
then %**def**
  op $\_\_ + \_\_ : Nat \times Nat \rightarrow Nat,$
    $assoc,\ comm,\ unit\ 0$ %**implied**
  $\forall m, n : Nat$
  • $0 + m = m$ %(add_0_Nat)%
  • $suc(n) + m = suc(n + m)$ %(add_suc_Nat)%
then %**implies**
  $\forall m, n : Nat$
  • $def\ pre(n) \Rightarrow pre(n) + m = pre(n + m)$ %(add_pre_Nat)%
then %**cons**
  op $bound : Nat$
end

Here, %**implied** means that the formulas induced by the annotated construct shall follow from the enclosing basic specification (plus the local environment).

4.1.5 Abstract Definitional Extension

Try to minimize requirements in DatatypeX.
Add derived concepts later as a definitional extension in ExtDatatypeX*[*DatatypeX*].*
Provide a non parametrized version of ExtDatatypeX *as* RichDatatypeX.

Example:
spec DefineBooleanAlgebra =
  sort $Elem$
  ops $0 : Elem;$
    $1 : Elem;$
    $\_\_ \sqcap \_\_ : Elem \times Elem \rightarrow Elem,$
      $assoc, comm, unit\ 1;$
    $\_\_ \sqcup \_\_ : Elem \times Elem \rightarrow Elem,$
      $assoc, comm, unit\ 0;$
  ...
end

spec BooleanAlgebra
  [DefineBooleanAlgebra **with sort** $Elem$, **ops** $0, 1, \_\_ \sqcap \_\_, \_\_ \sqcup \_\_$] =
%**def**

```
  SigOrder with preds __ ≤ __, __ ≥ __, __ < __, __ > __
then
  ∀x, y : Elem
  %% induced partial order:
  •  %(BoolAlg_leq_def] x ≤ y ⇔ x ⊓ y = x
end
```

Discussion: This style reduces the proof obligations induced by a view from DefineDatatypeX to another specification. In the above example this could be the view

```
view DefineBooleanAlgebra_in_FinitePowerSet
  [FiniteSet[Elem]] [op X : FinSet[Elem]]:
DefineBooleanAlgebra to
FinitePowerSet [FiniteSet[Elem]] [op X : FinSet[Elem]] =
  sort Elem ↦ FinitePowerSet[X],
  ops 0 ↦ {},
      1 ↦ X,
      __ ⊓ __ ↦ __ ∩ __,
      __ ⊔ __ ↦ __ ∪ __
end
```

Obviously this view deals only with the operations $0, 1, \sqcap, \sqcup$, but the predicate $\leq$ is not part of it. In spite of this, one automatically obtains the predicate $\leq$ in a specification SpecX by instantiating the specification BooleanAlgebra under the view DefineBooleanAlgebra_in_FinitePowerSet, i.e.

```
spec SpecX=
  ...
then
  BooleanAlgebra
      [view DefineBooleanAlgebra_in_FinitePowerSet]
  ...
end
```

Furthermore, one obtains a correctness check for the specified datatype: there is a proof obligation that the derived concepts that are defined in DatatypeX[DefineDatatypeX] are definitional extensions of the specification DefineDatatypeX.

Finally, using the proposed separation, it is possible to specify a certain category of models with DefineDatatypeX, which need not coincide with the category of reducts of DatatypeX[DefineDatatypeX] because the latter might introduce additional operations and predicates that decrease the number of homomorphisms.

4.1.6 Use of Views

*Use a **view*** SP1_IN_SP2 *to separate*
the definition of predicates and operations in SP2 *(design specification)*
from
the specification of their desired properties in SP1 *(requirement specification).*

```
spec NAT =
  . . .
  ∀m, n : Nat
  •  0 + m = m                                        %(add_Nat_0  )%
  •  suc(n) + m = suc(n + m)                          %(add_Nat_suc)%
end

view COMMUTATIVEMONOID_IN_NAT_ADD:
  COMMUTATIVEMONOID to NAT
=
  sort Elem ↦ Nat,
  ops  e ↦ 0,
       __ * __ ↦ __ + __
end
```

Discussion: This style has advantages from different points of view:

- Methodologically it separates the definition of an operator/predicate from the specification of its desired properties. Thus, it avoids confusion for the specifier.
- Concerning correctness it adds a proof obligation to the specification. Discharging this obligation (with a theorem proving tool) increases the trust in the correctness of the specifications.
- It enables efficient use of theorems and their proofs: If one proves a theorem in the above specification COMMUTATIVEMONOID it automatically holds in NAT as well as in all other specifications related by a view with COMMUTATIVEMONOID.

Concerning the tools this style means that they should transfer theorems as well as operation attributes via a view.

Use a generic view,
if the target of the view is a parameterized specification.

Example:
view MONOID_IN_LIST[ELEM] :
  MONOID **to** LIST[ELEM]
=
  sort *Elem* ↦ *List*[*Elem*],
  ops *0* ↦ [],
    __ + __ ↦ __ ++__
end

4.2 Use of structuring constructs

4.2.1 Union [Requirement]

In requirement specifications,
unions are used to combine arbitrary properties.

Example:
spec COMMUTATIVEGROUP =
  COMMUTATIVEMONOID **and** GROUP
end

4.2.2 Union [Design]

In design specifications,
the arguments of a union should have disjoint signatures.

Example:
spec LISTWITHNAT[ELEM] =
  NAT **and**
  LIST[ELEM]
end

Discussion: Unions can introduce new inconsistencies through interaction of properties of the united specifications. While this risk has to be taken in requirement specifications (which should be close to the problem description), for design specifications, this risk should be avoided: with disjoint signatures, the consistency of the united specification follows from that of the components.

4.2.3 Union versus Extension

If two specifications are independent,
combine them by union and not by extension.

spec TREEWITHINT[ELEM] =
  INT **and**
  TREE[ELEM]
end

Discussion: The “and” clearly indicates the indepdence of the specifications.

4.2.4 Renaming

Use renaming to give more mnemonic names
when using a specification in a new context.

Example:
spec STRING=
  LIST[ELEM] **with** $\mathit{List}[\mathit{Elem}] \mapsto \mathit{String}$
end

Use renaming to avoid name clashes.

Example:
spec FIELD =
  GROUP **and**
  GROUP **with** $\mathit{Elem} \mapsto \mathit{NonZeroElem}, \mathit{0} \mapsto \mathit{1}, + \mapsto *$
  ...
end

Use redundant renamings to indicate origin of symbols
(but only rarely, i.e. when really helpful).

Example:
spec FIELD =
  GROUP **with** $\mathit{0}, +$
and
  Group **with** $\mathit{Elem} \mapsto \mathit{NonZeroElem}, \mathit{0} \mapsto \mathit{1}, + \mapsto *$
  ...
end

4.2.5 Hiding

Use hiding to generate export signatures,
i.e. hiding auxiliary symbols that are only locally relevant.

Example:
```
spec SORTER =
  LIST[ELEM]
then
  ops sorter : List[Elem] → List[Elem]          ...
      insert : Elem × List[Elem] → List[Elem]
  hide insert
end
```

4.2.6 Local specifications

Alternatively, declare auxiliary symbols to be local.

Example:
```
spec SORTER =
  LIST[ELEM]
then local
  op insert : Elem × List[Elem] → List[Elem]
  within
  op sorter : List[Elem] → List[Elem]
     ...
end
```

4.2.7 Closed specifications

Avoid the use of closed specifications, if possible.

Example: Instead of

```
spec CONSTRUCTFIELD =
  COMMUTATIVERING
then • not e = 0
  sort NonZeroElem = {x : Elem • not x = 0}
then closed
  { GROUP with sort Elem ↦ NonZeroElem, ops e, __ * __ }
end
```

write

spec ConstructField =
  CommutativeRing
then • *not* $e = 0$
  sort *NonZeroElem* $= \{x : Elem$ • *not* $x = 0\}$
and { Group **with sort** *Elem* $\mapsto$ *NonZeroElem*, **ops** e, __ $*$ __ }
end

Discussion: In the first specification, the "closed" is crucial, since without it, the renaming of *Elem* into *NonZeroElem* also would affect CommutativeRing, which is neither legal nor wanted. The "closed" ensures that CommutativeRing is not visible in the renaming of Group — the same effect is achieved with the "and" in the second specification.

Use "closed" to avoid that
a formal parameter gets into the scope of a renaming.

Example:
spec ListRequirement[**sort** *Elem*] =
  closed { Monoid **with sort** *Elem* $\mapsto$ *List*[*Elem*],
                ops $e \mapsto []$, __ $*$ __ $\mapsto$ __ $++$__ }
then
  op __ :: __ : *Elem* $\times$ *List*[*Elem*] $\to$ *List*[*Elem*]
  $\forall x : Elem; L : List[Elem]$
  • $x :: L = (x :: []) ++ L$
end

Discussion: Here, the "closed" is necessary to avoid the renaming to affect also the formal paramter sort *Elem*.

4.2.8 Qualify in maps and symbol lists

Qualify symbol maps / symbol lists
with the keywords ***sort****,* ***pred****,* ***op****.*

Example:
view CommutativeMonoid_in_Nat_Add:
  CommutativeMonoid **to** Nat
=
  sort *Elem* $\mapsto$ *Nat*,

ops $e \mapsto 0$,
$\_\_ * \_\_ \mapsto \_\_ + \_\_$
end

4.3 Parameterized specifications

Split the parameters of a parameterized specification
into minimal useful pieces.

Discussion: Implicit instantiation of a parameterized specification, e.g. LIST[**sort** *Nat*], can be used only if there are not too many symbols in the parameter (otherwise, the fitting map will not be unique). By splitting parameters, one can achieve to have only a small number of symbols in each parameter.

Example:
spec PAIR[**sort** *Elem1*][**sort** *Elem2*] =
...
end

In the body of a parameterized specification,
use compound identifiers for new sorts,
as well as for new operation and predicate symbols
acting solely on parameter sorts.

Discussion: Compound identifiers help disambiguating symbols that come from different instantiations of one and the same parameterized specification. By the above guideline, it suffices in most cases to ensure that just the new sorts in the body are compound identifiers. Then, operations and predicates on the new sorts are automatically disambiguated by their profiles. Hence, only for operations and predicates acting solely on parameter sorts, compound identifiers need to be used.

Example:
spec LIST[ELEM] =
sort *List*[*Elem*]
ops [] : *List*[*Elem*];
$\_\_ :: \_\_ : Elem \times List[Elem] \to List[Elem]$
...
end

Prefer instantiation to renaming, if possible.

Discussion: The effect of implicit renaming of components of compound identifiers can only be achieved via fitting maps in an instantiation of a parameterized specification, not via a renaming.

Instantiate with the formal parameter, if you want to stay generic.

Example:
spec COMMUTATIVEMONOID[ELEM] =
  MONOID[ELEM]
then
  $\forall x, y : \mathit{Elem} \bullet x + y = y + x$
end

If in the body of a parameterized specification, another parameterized specification is instantiated twice (in ways depending differently on the parameter), the results of the instantiations should be renamed.

Examples: Consider the following specifications ignoring the above guideline:

spec SET [**sort** *Elem*] = **sort** *Set*[*Elem*] **end**

spec FINITEMAP[**sort** *S*][**sort** *T*] =
  SET[**sort** *S*]
and
  SET[**sort** *T*]
then
  sort *FiniteMap*[*S*, *T*]
end

spec SP = FINITEMAP[**sort** *S*][**sort** *S*] **end**

leads to the following error message

```
Fitting morphism leads to forbidden identifications
{ (sort Set[S] , sort Set[T]) }
```

indicating that the result of the instantiation FINITEMAP[**sort** *S*][**sort** *S*] is not a pushout.

When obeying the guideline, everything works fine:

spec SET [**sort** *Elem*] = **sort** *Set*[*Elem*] **end**

spec FINITEMAP[**sort** *S*][**sort** *T*] =
  SET[**sort** *S*] **with** *Set*[*S*] $\mapsto$ *SourceSet*[*S*]

and
  SET[**sort** T] **with** $Set[T] \mapsto TargetSet[T]$
then
  sort $FiniteMap[S, T]$
end

spec SP = FINITEMAP[**sort** S][**sort** S] **end**

4.4 Structured Free Extension

Use structured free specifications to specify predicates inductively.

Example I
spec BINARYRELATION =
  sort *Elem*
  pred $\_\_ \sim \_\_ : Elem \times Elem$
end

spec TRANSITIVECLOSURE [BINARYRELATION] =
  free
  { **pred** $\_\_ \sim^* \_\_ : Elem \times Elem$
    $\forall x, y, z : Elem$
    • $x \sim y \Rightarrow x \sim^* y$
    • $x \sim^* y \wedge y \sim^* z \Rightarrow x \sim^* z$
  }
end

In this specification, the use of a structured free extension cannot be avoided.

Example II: Avoiding inductive specification of predicates.

spec NAT_WITHSTRUCTFREEEXT =
  GENERATENAT
then
  free
  { **pred** $\_\_ \leq \_\_ : Nat \times Nat$
    $\forall m, n : Nat$
    • $0 \leq n$ %(leq_def_free1_Nat)%
    • $suc(m) \leq suc(n) \Rightarrow m \leq n$ %(leq_def_free2_Nat)%
  }
end

spec NAT =
  GENERATENAT
then
  pred $\_\_ \leq \_\_ : Nat \times Nat$
  $\forall m, n : Nat$
  • $0 \leq n$ %(leq_def1_Nat)%
  • $suc(m) \leq suc(n) \;\Leftrightarrow\; m \leq n$ %(leq_def2_Nat)%
  • $\neg(p \leq 0)$ %(leq_def3_Nat)%
end

The specification NAT makes the properties of the predicate $\leq$ more obvious: The axiom leq_def2_Nat of the specification NAT also holds in the specification NAT_WITHSTRUCTFREEEXT, but due to the structured free extension it is enough to specify this property by the axiom leq_def_free2_Nat. The same holds for axiom leq_def3_Nat, which again is not needed in the specification NAT_WITHSTRUCTFREEEXT due to the free construct.

Use a **free type** *instead of a structured free specifications if no axioms are present.*

Example
spec GENERATENAT_WITHSTRUCTFREEEXT =
  free **{type** $Nat ::= 0 \mid suc(pre :?Nat)$ **}**
end

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

The curly brackets in GENERATENAT_WITHSTRUCTFREEEXT can be omitted without any effect on the semantics of the sorts *Nat* and *Pos* or on the semantics of the operations 0, *suc* and *pre*. Thus we prefer the specification GENERATENAT.

A structured free datatype specification with axioms can be replaced by a **generated type**, *with equality characterized by observers.*

Example
spec GENERATEFINITESET_WITHSTRUCTFREEEXT [**sort** *Elem*] =
free
  { **type** $FinSet[Elem] ::= \{\}$
  $\mid \{\_\_\}(Elem)$
  $\mid \_\_ \cup \_\_(FinSet[Elem]; FinSet[Elem])$

op $\_\_ \cup \_\_ : \mathit{FinSet}[\mathit{Elem}] \times \mathit{FinSet}[\mathit{Elem}] \to \mathit{FinSet}[\mathit{Elem}],$
$\quad \mathit{assoc}, \mathit{comm}, \mathit{idem}, \mathit{unit}\ \{\}$

}

end

spec GENERATEFINITESET_WITHOUTSTRUCTFREEEXT [**sort** *Elem*] =

generated type $\mathit{FinSet}[\mathit{Elem}] ::= \{\}$
$\quad | \ \{\_\_\}(\mathit{Elem})$
$\quad | \ \_\_ \cup \_\_(\mathit{FinSet}[\mathit{Elem}]; \mathit{FinSet}[\mathit{Elem}])$

pred $\_\_\mathit{elemOf}\_\_ : \mathit{Elem} \times \mathit{FinSet}[\mathit{Elem}]$

$\forall x, y : \mathit{Elem}; s, s1, s2 : \mathit{FinSet}[\mathit{Elem}]$

- $\neg\ x\ \mathit{elemOf}\ \{\}$ %(elemOf_empty)%
- $x\ \mathit{elemOf}\ \{y\} \Leftrightarrow x = y$ %(elemOf_singleton)%
- $x\ \mathit{elemOf}\ (s1\ \cup\ s2) \Leftrightarrow$
  $x\ \mathit{elemOf}\ s1 \vee x\ \mathit{elemOf}\ s2$ %(elemOf_union)%
- $s1 = s2 \Leftrightarrow$
  $\forall u : \mathit{Elem} \bullet u\ \mathit{elemOf}\ s1 \Leftrightarrow u\ \mathit{elemOf}\ s2$ %(extensionality)%

end

The effect of the free extension can be obtained by using a generated type and characterize equality on this type using an observer. In this example, the observer is $elemOf$, for bags, one would use a counting operation, for lists an indexing operation.

The effect of the free extension can be obtained by using a generated type and characterize equality on this type using an observer. In this example, the observer is $elemOf$, for bags, one would use a counting operation, for lists an indexing operation.

A structured free datatype specification with axioms sometimes can be replaced by a **generated type***, with equality specified directly.*

Example

spec GENERATERAT_WITHSTUCTFREEEXT =

INT

then

free

{

type $\mathit{Rat} ::= \_\_/\_\_(\mathit{nom} : \mathit{Int}; \mathit{denom} : \mathit{Pos})$

$\forall i, j : \mathit{Int}; p, q : \mathit{Pos}$

- $i/p = j/q \Leftrightarrow i * q = j * p$ %(equality_Rat)%

}

end

spec GENERATERAT_WITHOUTSTUCTFREEEXT =

    INT
then
    generated type $Rat ::= \_\_/\_\_(nom : Int; denom : Pos)$
    $\forall i, j : Int; p, q : Pos$
    • $i/p = j/q \Leftrightarrow i * q = j * p$ %(equality_Rat)%
end

In this case, we can avoid structured free extensions by directly specifying the equality on generated type, i.e. we even do not need auxiliary observers.

Discussion: The main reasons for these guidelines are:

- The semantics of structured free extensions is complex, i.e. specifications without structured free extensions are easier to understand.
- Theorem proving supported for structured free extensions can be expected only for basic specifications in Horn clauses form, since in this case, the semantics can be obtained as a quotient term algebra. For free extensions within arbitrary first-order logic, we do not know of theorem proving support (the only idea here seems to be to use a meta-theoretic specification of the CASL semantics, which is rather clumsy when used as a proof tool). Moreover, putting a free around a first-order specification even need not to imply generatedness. The same holds for structured specifications that are not flattenable to basic specifications (i.e. those containing a **hide**, **free** or **local**).
- However, even in the Horn clause case, theorem proving supported for structured free extensions is rather weak. Of course, it is easy to exploit the inductive properties, but this can also be done when using generated types. The difficult problem is to exploit what distingishes free types from generated types: the negative consequence, i.e. inequalites. Structured free extensions need to be translated to a second-order specification of the quotient term algebra. Negative consequences can be then obtained only by explicitly constructing a congruence satisfying the axioms which does not include the equality that shall be disproved. An alternative way is to prove confluence and then compare normal forms — however, this is not always applicable (in particular not for proving refinements of free specifications into other ones) and also rather sophisticated, in particular when it shall be integrated with other proof techniques. In contrast, an explicit specification of equality on a generated type (either directly or using observers) allows a much easier derivation of negative consequences.
- Only for inductively specified predicates, there may be no alternative to the use of a structured free extension, cf. the first guideline above.

References

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[2] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI/`.

[3] Till Mossakowski and Markus Roggenbach. The datatypes REAL and COMPLEX in CASL. Note M-7, in [2], April 1999.

[4] Markus Roggenbach and Till Mossakowski. Basic datatypes in CASL. Note L-12, version 0.4.1, in [2], May 2000.