7 Generic Specifications

Making a specification generic (when appropriate) improves its reusability.

As mentioned in the previous chapter, naming specifications is a good idea. In many cases, however, datatypes are naturally *generic*, having sorts, operations, and/or predicates that are deliberately left loosely specified, to be determined when the datatype is used. For instance, datatypes of lists and sets are generic regarding the sort of elements. *Generic specifications* allow the genericity of a datatype to be made explicit by declaring *parameters* when the specification is named: in the case of lists and sets, there is a single parameter that simply declares the sort *Elem*.[1] A fitting argument specification has to be provided for each parameter of a generic specification whenever it is referenced; this is called *instantiation* of the generic specification.

The aim of this chapter is to discuss and illustrate how to define generic specifications and instantiate them. We have seen plenty of simple examples of generic specifications and instantiations in the previous chapters. In more complicated cases, however, explicit fitting symbol maps may be required to determine the exact relationship between parameters and arguments in instantiations, and so-called *imports* should be separated from the bodies of generic specifications.

[1] Generic specifications are also useful to ensure loose coupling between several named specifications, replacing an explicit extension by a parameter including only the necessary symbols and their required properties. This is illustrated in the Steam-Boiler Control System case study, see Chap. 13.

7.1 Parameters and Instantiation

Parameters are arbitrary specifications.

Any specification, named or not, can be used as the parameter of a generic specification. Commonly, the parameter is a rather trivial specification consisting merely of a single sort declaration, as in most of the examples given in the previous chapters, e.g.:

spec GENERIC_MONOID [**sort** *Elem*] = **%{** See Chap. 3, p. 30 **}%**

spec LIST_SELECTORS [**sort** *Elem*] = **%{** See Chap. 4, p. 54 **}%**

However, the parameter can also be a more complex, possibly structured, specification, as in:

spec LIST_ORDER [TOTAL_ORDER **with sort** *Elem*, **pred** __ < __] =
%{ See Chap. 6, p. 73 **}%**

Recall that '**with**' requires the signature of the specification to include the listed symbols; here, in fact, the signature of TOTAL_ORDER does not contain any further symbols, so those are all the symbols that have to be supplied when instantiating LIST_ORDER.

The argument specification of an instantiation must provide symbols corresponding to those required by the parameter.

spec LIST_ORDER_NAT = LIST_ORDER [NATURAL_ORDER]

The correspondence between the symbols provided by the argument specification and those required by the parameter can be described by a fitting symbol map or left implicit when it is not ambiguous, which is often the case.

In the above example, the argument specification NATURAL_ORDER provides the sort *Nat*, the operation symbols *0* and *suc*, and the binary predicate symbol '<'. Hence this specification indeed provides symbols corresponding to those required by the parameter specification TOTAL_ORDER and the correspondence can be left implicit because the argument NATURAL_ORDER has only single symbols of the right kind. (The coincidence of the predicate symbol in the parameter and argument is irrelevant here.)

How to describe explicit fitting symbol maps and when they can be omitted is detailed later in this section.

The argument specification of an instantiation must ensure that the properties required by the parameter hold.

spec NAT_WORD = GENERIC_MONOID [NATURAL]

A *(fitting) signature morphism* from the signature of the parameter part to the signature of the argument specification is automatically deduced, taking into account the explicitly specified fitting symbol map if any (the situation here is quite similar to a renaming, where a signature morphism is deduced from a symbol map). The instantiation is *defined* if all models of the argument specification, when reduced along the induced fitting signature morphism, provide models of the parameter part. In particular the symbols provided by the argument specification must have the properties, if any, specified in the parameter for their counterparts. When this is the case, we get not only a signature morphism, but also a *(fitting) specification morphism* from the argument specification to the parameter specification.[2]

In the above NAT_WORD example, since the parameter of GENERIC_MONOID is trivial, it is obvious that the instantiation is defined.

The effect of the instantiation is to make the union of the argument specification and of the (non generic equivalent of the) generic specification, renamed according to the induced fitting signature morphism. In particular, a side-effect of the instantiation is to rename the symbols of the generic specification according to the fitting signature morphism induced by the instantiation. In our NAT_WORD example, the operation symbol $inj : Elem \rightarrow Monoid$ is renamed into $inj : Nat \rightarrow Monoid$, while the operation symbols '*1*' and '$*$' are left unchanged (as well as the sort *Monoid*). Thus, the specification NAT_WORD abbreviates the following specification:
NATURAL **and** { NON_GENERIC_MONOID **with** *Elem* ↦ *Nat* }.

When convenient, the instantiation can be completed by a renaming, as illustrated in the following variant of NAT_WORD.

spec NAT_WORD_1 =
    GENERIC_MONOID [NATURAL]
    with *Monoid* ↦ *Nat_Word*
end

In the case of the specification LIST_ORDER_NAT above, checking the definedness of the instantiation corresponds to a non-trivial proof obligation. The instantiation is defined since the predicate '<' provided by NATURAL_ORDER is indeed a total ordering relation, hence the properties required by

[2] Note that consistency is entirely orthogonal to definedness: a defined instantiation may be consistent or not.

TOTAL_ORDER are fulfilled, even if there is no syntactic correspondence between the axioms given in TOTAL_ORDER and those in NATURAL_ORDER.

There must be no shared symbols between the argument specification and the body of the instantiated generic specification.

spec THIS_IS_WRONG = GENERIC_MONOID [MONOID]

The intention in the above example may have been to specify monoids of monoids. However, 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.

Section 7.3 provides useful hints on how to structure generic specifications in order to avoid as far as possible undesirable clashes of symbols in instantiations. A correct specification of monoids of monoids is provided in Sect. 7.2, p. 86.

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 ...

When the parameter and the argument have symbols in common, these parameter symbols are implicitly taken to fit directly to the corresponding argument symbols. Thus it is *never* necessary to make explicit that a symbol is mapped identically. In the above example, for instance, the parameter specification of GENERIC_MONOID is exactly the same as the argument specification in its instantiation, so the fitting can be left implicit.

The fitting of parameter sorts to unique argument sorts can also be left implicit.

When the argument specification has only a single sort, the fitting of all parameter sorts to that sort is obvious, and can again be left implicit, as illustrated earlier by the NAT_WORD specification. Of course, this does not apply the other way round: if the parameter has a single sort (which is often the case in practice) but the argument specification has more than one sort, the parameter sort could be mapped to any of the argument sorts, so the fitting symbol map has to be made explicit – except when the parameter sort is identical to one of the argument sorts, as previously explained, or when the fitting of sorts can be implied from the fitting of other symbols, as explained below.

Fitting of operation and predicate symbols can sometimes be left implicit too, and can imply fitting of sorts.

spec LIST_ORDER_POSITIVE = LIST_ORDER [POSITIVE]

Fitting of operation and predicate symbols can imply fitting of sorts. For instance, when a parameter predicate symbol is fitted to an argument predicate symbol whose profile involves different sorts, this implies that the parameter sorts involved have to be fitted to the corresponding sorts in the argument specification.

This is illustrated in the above LIST_ORDER_POSITIVE specification. In a first step, the fitting of the parameter sort *Elem* to one of the argument sorts *Nat* and *Pos* provided by the specification POSITIVE (see Chap. 5, p. 61) may seem ambiguous. However, no explicit fitting of symbols is necessary here, since the argument specification provides only one binary predicate symbol, and the fitting of the corresponding binary predicate symbol of the parameter specification to it entails the fitting of the sort *Elem* to the sort *Nat* (Again, the coincidence of the predicate symbol in the parameter and argument is irrelevant here.)

As may be clear by now, the exact rules for when the fitting between parameter and argument symbols can be left implicit are quite sophisticated. It seems best to make the intended fitting explicit whenever it is not completely obvious, using the notation for fitting arguments illustrated in the following examples.

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** $Elem \mapsto Nat$]

The correspondence between the symbols required by the parameter and those provided by the argument specification can be made explicit using so-called *fitting symbol maps*. For instance, the above NAT_WORD_2 specification, which differs from NAT_WORD only regarding the presence of subsorts of

Nat, is obtained as an instantiation of GENERIC_MONOID, fitting the parameter part '**sort** *Elem*' to the NATURAL_SUBSORTS specification. The mapping between the parameter sort *Elem* and the sort *Nat* provided by NATURAL_SUBSORTS is described by the fitting symbol map '**fit** *Elem* $\mapsto$ *Nat*'.

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

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

Using several parameters is merely a notational convenience, since they are equivalent to their union. For instance, the above PAIR specification is nothing but a variant of the specification PAIR_1 with just one parameter '**sorts** *Elem1*, *Elem2*' defined in Chap. 4, p. 54.

Note that writing:

spec HOMOGENEOUS_PAIR_1 [**sort** *Elem*] [**sort** *Elem*] =
  free type *Pair* ::= *pair*(*first* : *Elem*; *second* : *Elem*)

merely defines pairs of values of the same sort, and HOMOGENEOUS_PAIR_1 is (equivalent to and) better defined as follows:

spec HOMOGENEOUS_PAIR [**sort** *Elem*] =
  free type *Pair* ::= *pair*(*first* : *Elem*; *second* : *Elem*)

since the two parameters in HOMOGENEOUS_PAIR_1 are equivalent to just one '**sort** *Elem*' parameter.

From a methodological point of view, it is generally advisable to use as many parameters as convenient: the part of the specification that is intended to be specialized at instantiation time is better split into logically coherent units, each one corresponding to a parameter. Consider for instance:

spec TABLE [**sort** *Key*] [**sort** *Val*] = ...

Here, using two parameters in TABLE emphasizes that *Key* and *Val* are logically distinct entities which can be specialized as desired independently of each other.

> *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* $\mapsto$ *RGB*]

In the above example, the first parameter '**sort** *Elem1*' of PAIR is instantiated by NATURAL_ARITHMETIC, which exports only one sort *Nat*, hence no explicit fitting symbol map is necessary. The second parameter '**sort** *Elem2*' of PAIR is instantiated by COLOR: in this case a fitting symbol map must be provided, since COLOR exports two sorts, *RGB* and *CMYK*.

Using the specification PAIR_1 would require us to write:

spec PAIR_NATURAL_COLOR_1 =
    PAIR_1 [NATURAL_ARITHMETIC **and** COLOR
        fit $Elem1 \mapsto Nat,\ Elem2 \mapsto RGB$]

which clearly demonstrates the benefit of using two parameters as in PAIR instead of just one as in PAIR_1.

When parameters are trivial ones (i.e., just one sort), one can always avoid explicit fitting maps. Consider for instance the following alternative to PAIR_NATURAL_COLOR:

spec PAIR_NATURAL_COLOR_2 =
    PAIR [**sort** *Nat*] [**sort** *RGB*]
and NATURAL_ARITHMETIC **and** COLOR

This may be convenient when the argument specification exports several sorts. Compare for instance:

spec PAIR_POS =
    HOMOGENEOUS_PAIR [**sort** *Pos*] **and** INTEGER_ARITHMETIC_1

with:

spec PAIR_POS_1 =
    HOMOGENEOUS_PAIR [INTEGER_ARITHMETIC_1 **fit** $Elem \mapsto Pos$]

Note that the instantiation:
HOMOGENEOUS_PAIR_1 [NATURAL] [COLOR **fit** $Elem \mapsto RGB$]
is ill-formed, since it entails mapping the sort *Elem* to both *Nat* and *RGB*.

More generally, care is needed when the several parameters of a generic specification share some symbols, which in general is not advisable.

As a last remark, note that it is easy to specialize a generic specification with several parameters, using a 'partial instantiation', as in the following version of TABLE:

spec MY_TABLE [**sort** *Val*] =
    TABLE [NATURAL_ARITHMETIC] [**sort** *Val*]

where we still have a parameter for the values to be stored, but have decided that the keys are natural numbers.

Composition of generic specifications is expressed using instantiation.

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

The above generic specification SET_OF_LIST describes sets of lists of arbitrary elements, and is obtained by an instantiation of the generic specification GENERATED_SET, whose parameter '**sort** *Elem*' is instantiated by the specification LIST_SELECTORS, itself trivially instantiated. Since the (trivially instantiated) specification LIST_SELECTORS exports two sorts *Elem* and *List*, it is of course necessary to specify, in the instantiation of GENERATED_SET, the fitting symbol map from the parameter sort *Elem* to the argument sort *List*.

Note especially that the following specification:

spec MISTAKE [**sort** *Elem*] =
  GENERATED_SET [LIST_SELECTORS [**sort** *Elem*]]

does *not* provide sets of lists of elements: The sort *Elem* in the parameter of GENERATED_SET is mapped by the identity fitting symbol map to the sort *Elem* provided by the instantiation of the generic specification LIST_SELECTORS [**sort** *Elem*], rather than to the sort *List*.[3] Thus MISTAKE just provides sets of arbitrary elements *and* lists of arbitrary elements. If this was indeed the desired effect, then one should rather write instead:

spec SET_AND_LIST [**sort** *Elem*] =
  GENERATED_SET [**sort** *Elem*] **and** LIST_SELECTORS [**sort** *Elem*]

As illustrated by SET_OF_LIST, composition of generic specifications is fairly easy in CASL. Note however that this composition is achieved by means of appropriate instantiations (some possibly trivial), and that CASL does not provide higher-order genericity.

It may be worth mentioning that the following composition of generic specifications is ill-formed:

spec THIS_IS_STILL_WRONG =
  GENERIC_MONOID [GENERIC_MONOID [**sort** *Elem*]
                  fit *Elem* $\mapsto$ *Monoid*]

[3] However, the situation would be different if the parameter of GENERATED_SET had been, e.g., '**sort** *Val*', since then the absence of an explicit fitting symbol map would have led to an ambiguity: in that case the specifier would have to specify whether the sort *Val* is to be mapped to *Elem* or to *List*.

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 GENERIC_MONOID [**sort** *Elem*] (where this time the generic specification GENERIC_MONOID is trivially instantiated). The next section provides (p. 86) a correct specification of monoids of monoids.

7.2 Compound Symbols

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

```
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 ]
```

A compound sort is a sort of the form '*Name*[*Name1*, ..., *NameN*]'. In the specification LIST_REV, we introduce a compound sort *List*[*Elem*] to denote lists (of arbitrary elements), instead of the simple sort *List* used in the previous examples. When the specification LIST_REV is instantiated as in LIST_REV_NAT, the translation induced by the (implicit) fitting symbol map is applied to the component *Elem* also where it occurs in *List*[*Elem*], providing a sort *List*[*Nat*]. Thus, compound sorts can be seen as a convenient way of implicitly completing the instantiation by an appropriate renaming of the (compound) sorts introduced by the generic specification.

```
spec  TWO_LISTS =
      LIST_REV [ NATURAL ]  %% Provides the sort List[Nat]
and   LIST_REV [ COLOR fit Elem ↦ RGB ]  %% Provides the sort List[RGB]
```

Using a compound sort *List*[*Elem*] proves particularly useful in the above example TWO_LISTS, where we make the union of two distinct instantiations of LIST_REV. If we had used an ordinary sort *List*, then an unintentional name clash would have arisen,[4] and we would have to complete each instantiation by an explicit renaming of the sort *List*.

Note that in the specification TWO_LISTS, we have two sorts *List*[*Nat*] and *List*[*RGB*], hence two overloaded constants *empty* (one of each sort), which may need disambiguation when used in terms. (How to disambiguate terms is explained in Chap. 3, p. 31.)

Similarly, we have overloaded operation symbols *cons*, *head*, *tail*, ++, and *reverse*, but in general their context of use in terms will be enough to disambiguate which one is meant.

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

Since the specification INTEGER_ARITHMETIC_1 provides three sorts *Nat*, *Pos*, and *Int*, an explicit fitting symbol map is needed in the above instantiations, which provide the sorts *List*[*Nat*] and *List*[*Int*]. Note that the subsorting relation $Nat < Int$ does *not* entail $List[Nat] < List[Int]$, but of course this can be added if desired in an extension by a subsorting declaration.

Using compound sorts, we can now easily specify monoids of monoids.

```
spec  MONOID_C [sort Elem] =
      sort Monoid[Elem]
      ops  inj    : Elem → Monoid[Elem];
           1      : Monoid[Elem];
           __ * __ : Monoid[Elem] × Monoid[Elem] → 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]]
```

The instantiation in MONOID_OF_MONOID is now correct, since the use of a compound sort *Monoid*[*Elem*] ensures there is no clash of symbols between the body of the instantiated generic specification and the argument specification.

[4] And the specification TWO_LISTS would have been inconsistent, due to the same name, same thing principle and the fact that *List* is defined by a free type construct.

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   LIST_REV_WITH_TWO_ORDERS =
       LIST_REV_ORDER
         [INTEGER_ARITHMETIC_ORDER fit Elem ↦ Int, __ < __ ↦ __ < __]
         %% Provides the sort List[Int] and the operation order[__ < __]
and    LIST_REV_ORDER
         [INTEGER_ARITHMETIC_ORDER fit Elem ↦ Int, __ < __ ↦ __ > __]
         %% Provides the sort List[Int] and the operation order[__ > __]
then %implies
       ∀L : List[Int] • order[__ < __](L) = reverse(order[__ > __](L))
end
```

The above example illustrates the use of compound identifiers for operation symbols, and the same rules apply to predicate symbols. While in most cases using compound identifiers for sorts will be sufficient, in some situations it is also convenient to use them for operation or predicate symbols, as done here for *order*[__ < __]. When LIST_REV_ORDER is instantiated, not only does the sort *List*[*Elem*] get renamed (here, to *List*[*Int*]), but also the operation symbol *order*[__ < __], according to the fitting symbol map corresponding to the instantiation. If we had not used a compound identifier for the *order* operation, then an unintentional name clash would have arisen. Note that on the other hand we rely on the same name, same thing principle to ensure that the sorts *List*[*Int*] provided by each of the two instantiations are the same, which indeed is what we want for this example.

Of course we do not bother to use a compound identifier for the *insert* operation symbol. This operation being local, it is not exported by LIST_REV_ORDER and cannot be the source of unintentional name clashes in instantiations.

7.3 Generic Specifications with Imports

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

spec LIST_WEIGHTED_ELEM [**sort** *Elem* **op** $weight : Elem \to Nat$]
  given NATURAL_ARITHMETIC =
 LIST_REV [**sort** *Elem*]
then **op** $weight : List[Elem] \to Nat$
 $\forall e : Elem;\ L : List[Elem]$
 • $weight(empty) = 0$
 • $weight(cons(e, L)) = weight(e) + weight(L)$
end

In the above example, we specialize lists of arbitrary elements to lists of elements equipped with a *weight* operation, which is then overloaded by a *weight* operation on lists. Therefore we specify that the generic specification LIST_WEIGHTED_ELEM has for parameter a specification extending the '**given**' specification NATURAL_ARITHMETIC by a sort *Elem* and an operation symbol *weight*. Thereby the intention is to emphasize the fact that only the sort *Elem* and the operation *weight* are intended to be specialized when the specification LIST_WEIGHTED_ELEM is instantiated, and not the 'fixed part' NATURAL_ARITHMETIC. In CASL, the specifications listed after the '**given**' keyword are called *imports*. One could have written instead:

spec LIST_WEIGHTED_ELEM
 [NATURAL_ARITHMETIC **then sort** *Elem* **op** $weight : Elem \to Nat$]
 = ...

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 '**given** NATURAL_ARITHMETIC' clause would make the declaration:

spec LIST_WEIGHTED_ELEM [**sort** *Elem* **op** $weight : Elem \to Nat$] = ...

ill-formed, since the sort *Nat* is not available.

To summarize, the '**given**' construct is useful to distinguish the 'true' parameter from the part which is 'fixed'. Both the parameter and the body of the generic specification extend what is provided by the imports (i.e., by the specifications listed after the '**given**' keyword), whose exported symbols are therefore 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** $Elem \mapsto Pair$,
$weight \mapsto first$]

The instantiation specified in LIST_WEIGHTED_PAIR_NATURAL_COLOR is correct since the fitting symbol map is the identity on all the symbols exported by the 'fixed part' NATURAL_ARITHMETIC (which happens here to be included in the argument specification PAIR_NATURAL_COLOR). More generally, the argument specification is always regarded as an extension of the imports, and the fitting symbol map should be the identity on all symbols provided by these imports. This is illustrated in the next example:

spec LIST_WEIGHTED_INSTANTIATED =
LIST_WEIGHTED_ELEM [**sort** *Value* **op** $weight : Value \to Nat$]

Here we rely on a rather trivial instantiation (whose purpose is merely to illustrate our point) where the fitting symbol map can be omitted since no ambiguity arises and where the argument specification '**sort** *Value* **op** $weight : Value \to Nat$' is well-formed because it is regarded as an extension of the imports of LIST_WEIGHTED_ELEM (i.e., as an extension of NATURAL_ARITHMETIC), which implies that the sort *Nat* is available.

Imports are also useful to prevent ill-formed instantiations.

spec LIST_LENGTH [**sort** *Elem*] **given** NATURAL_ARITHMETIC =
LIST_REV [**sort** *Elem*]
then **op** $length : List[Elem] \to Nat$
$\forall e : Elem;\ L : List[Elem]$
• $length(empty) = 0$
• $length(cons(e, L)) = length(L) + 1$
then %implies
$\forall L : List[Elem]$ • $length(reverse(L)) = length(L)$
end

The specification LIST_LENGTH needs the sort *Nat* and the usual arithmetic operations provided by NATURAL_ARITHMETIC to specify the *length* operation. In this case it is clear that the imports have nothing to do with the (trivial) parameter of LIST_LENGTH. The reason to specify NATURAL_ARITHMETIC as an import is that this will make instantiations of LIST_LENGTH similar to the following one well-formed.

spec LIST_LENGTH_NATURAL =
LIST_LENGTH [NATURAL_ARITHMETIC]

To understand this point, consider the following variant of LIST_LENGTH:

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

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, as already explained p. 80. 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.

As illustrated by the above examples, one should remember the following essential point. 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 (such as NATURAL_ARITHMETIC for LIST_LENGTH) into imports, and generic specifications of the form '$F\,[\,X\,] = SP$ **then** . . .' are better written '$F\,[\,X\,]$ **given** $SP = \ldots$' to allow the instantiation '$F\,[\,SP\,]$'.

7.4 Views

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

view INTEGER_AS_TOTAL_ORDER :
TOTAL_ORDER **to** INTEGER_ARITHMETIC_ORDER =
$Elem \mapsto Int,\ \_\_ < \_\_ \mapsto \_\_ < \_\_$

view Integer_as_Reverse_Total_Order :
  Total_Order **to** Integer_Arithmetic_Order =
  $\mathit{Elem} \mapsto \mathit{Int}$, $\_\_ < \_\_ \mapsto \_\_ > \_\_$

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
  $\forall L : \mathit{List}[\mathit{Int}] \bullet \mathit{order}[\_\_ < \_\_](L) = \mathit{reverse}(\mathit{order}[\_\_ > \_\_](L))$
end

A view is nothing but a convenient way to name a specification morphism (induced by a symbol map) from a (parameter) specification to an (argument) specification. The rules regarding the omission of 'evident' symbol maps in explicit fittings apply to views too. A view proves particularly useful when the same instantiation (with the same fitting symbol map) is intended to be used several times: naming a specification morphism once and for all makes its reuse easier. Once a view is defined, as e.g. Integer_as_Total_Order above, it can be referenced in instantiations as in List_Rev_Order [**view** Integer_as_Total_Order], where the keyword '**view**' makes it clear that the argument is not merely a named specification with an implicit fitting map, which would be written differently.

Since a view is defined only when the given symbol map induces a specification morphism (i.e., all models of the target specification, when reduced along the signature morphism induced by the given symbol map, provide models of the source specification), it may be convenient to use views just to explicitly document the existence of some specification morphisms, even when these are not intended to be used in any instantiation. For instance, the view Integer_as_Total_Order can be seen as the assertion that Integer_Arithmetic_Order indeed specifies '<' to be a total ordering relation, and would therefore make sense even without being used later on in instantiations.

Views can also be generic.

view List_as_Monoid [**sort** *Elem*] :
  Monoid **to** List_Rev [**sort** *Elem*] =
  $\mathit{Monoid} \mapsto \mathit{List}[\mathit{Elem}]$, $\mathit{1} \mapsto \mathit{empty}$, $\_\_ * \_\_ \mapsto \_\_ {+}{+} \_\_$

A view can be generic, being then defined with some parameters (as illustrated above in the List_as_Monoid view) and possibly some imports. The reader should be aware that, in a generic view, the target specification (here, the trivially instantiated specification List_Rev) is not interpreted as such, but as the body of a generic specification with the same parameters and imports as the view. (The source specification is on the contrary interpreted exactly as provided.)

The above example illustrates again the use of a view as a 'proof obligation', asserting that lists (equipped with the '++' operation) form a monoid.