6

Structuring Specifications

Large and complex specifications are easily built out of simpler ones by means of (a small number of) specification-building operations.

In the previous chapters, we have focused attention on basic specifications and detailed how to use the various constructs of CASL to write meaningful, but relatively simple, specifications. The aim of this chapter is to discuss and illustrate how to assemble simple pieces of specifications into more complex, structured ones. In particular we explain how to extend specifications, make the union of several specifications, as well as how to rename or hide symbols when assembling specifications. Parametrization and instantiation of generic specifications are explained in the next chapter.

6.1 Union and Extension

Union and extension can be used to structure specifications.

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

end

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 67–75, 2004.

The above example shows how to make the union (expressed by '**and**') of two specifications List_Selectors (see Chap. 4, p. 54) and Generated_Set (see Chap. 3, p. 35), and then further extend this union by an operation and some axioms (using '**then**'). Union and extension are the most commonly used specification-building operations. In contrast with extension, whose purpose is to extend a given piece of specification by new symbols and axioms, union is generally used to combine two self-contained specifications. Union of specifications is obviously associative and commutative.

All symbols introduced by a specification are by default exported by it and visible in its extensions or in its unions with other specifications. (Variables are not considered as symbols, and never exported.) Remember also the '*same name, same thing*' principle: in the above List_Set specification, it is therefore the same sort *Elem* which is used to construct both lists and sets.[1]

Specifications may combine parts with loose, generated, and free interpretations.

spec List_Choose [**sort** *Elem*] =
List_Selectors [**sort** *Elem*]
and Set_Partial_Choose_2 [**sort** *Elem*]
then **ops** *elements_of__* : *List* → *Set*;
choose : *List* →? *Elem*
∀ *e* : *Elem*; *L* : *List*
• *elements_of empty* = *empty*
• *elements_of cons*(*e*, *L*) = {*e*} ∪ *elements_of L*
• *def choose*(*L*) ⇔ ¬(*L* = *empty*)
• *choose*(*L*) = *choose*(*elements_of L*)
end

spec Set_to_List [**sort** *Elem*] =
List_Set [**sort** *Elem*]
then **op** *list_of__* : *Set* → *List*
∀ *S* : *Set* • *elements_of*(*list_of S*) = *S*
end

The specification List_Choose is built as an extension of the union of List_Selectors and Set_Partial_Choose_2 (see Chap. 4, p. 50). This extension introduces an operation *elements_of* (as in List_Set) and a partial operation *choose*, which are defined by some axioms. In List_Selectors, lists are defined by a free datatype construct (with selectors), hence have a

[1] The constant *empty* is overloaded, since we have a constant *empty* : *List* for lists and a constant *empty* : *Set* for sets.

free interpretation. SET_PARTIAL_CHOOSE_2 is itself an extension of (SET_PARTIAL_CHOOSE which is an extension of) GENERATED_SET, where sets are defined by a generated datatype construct. However, note that as discussed in Chap. 3, p. 35, the apparently loose specification GENERATED_SET is in fact not so. Moreover, the *choose* partial function on sets is loosely defined in SET_PARTIAL_CHOOSE_2, and so is therefore also the *choose* partial function on lists defined in LIST_CHOOSE. It is easy to see that the operation *elements_of* is uniquely defined. The sort *Elem* has of course a loose interpretation.

Thus the specification LIST_CHOOSE combines parts with a free interpretation, parts with a generated interpretation, and parts with a loose interpretation. The situation is similar to that with LIST_SET (and SET_TO_LIST), where the operation *list_of* is loosely defined with the help of the operation *elements_of*.

6.2 Renaming

> *Renaming may be used to avoid unintended name clashes, or to adjust names of sorts and change notations for operations and predicates.*

spec STACK [**sort** *Elem*] =
LIST_SELECTORS [**sort** *Elem*] **with sort** *List* $\mapsto$ *Stack*,
ops *cons* $\mapsto$ *push__onto_*,
head $\mapsto$ *top*,
tail $\mapsto$ *pop*
end

While the 'same name, same thing' principle has proven to be appropriate in numerous examples given in the previous chapters and above, it may still happen that, when combining specifications, this principle leads to unintended name clashes. An unintended name clash arises for instance when one combines two specifications that both export the same symbol (with the same profile in case of an operation or a predicate), while this symbol is not intended to denote the same 'thing' in the combination. In such cases, it is necessary to explicitly *rename* some of the symbols exported by the specifications put together in order to avoid the unintended name clashes.

When reusing a named specification, it may be convenient to rename some of its symbols; moreover, in the case of operation or predicate symbols, one may also change the style of notation. This is illustrated in the specification STACK above which is obtained as a renaming of the specification LIST_SELECTORS. (A renaming is introduced by the keyword '**with**'.) First, the sort *List* is renamed into *Stack*, then the operation *cons* is renamed into a mixfix

operation *push_onto_*, and finally the selectors *head* and *tail* are renamed into *top* and *pop*, respectively. Note that '↦' is input as '|->'.

The user only needs to indicate how symbols provided by the renamed specification are mapped to new symbols. A *signature morphism* is automatically deduced from this 'symbol map'. For instance, the signature morphism inferred from the symbol map specified in STACK maps the operation symbol $cons : Elem \times List \to List$ to the operation symbol $push\_onto\_ : Elem \times Stack \to Stack$: not only the operation name is changed, but also its profile according to the renaming of *List* into *Stack*.

In a symbol map, one can qualify the symbol to be renamed by its kind, using the keywords **sort**, **op**, and **pred** (or their plural forms), as appropriate; this is illustrated in STACK above. Qualification in symbol maps is generally recommended since it improves their readability.

While it is possible to change the syntax of an operation or predicate symbol, as illustrated above for *cons* mapped to *push_onto_*, it is not possible to change the order of the arguments of the renamed operation or predicate.

In general, one does not need to rename all the symbols provided by the specification to be renamed. In the symbol map describing the intended renaming, it is indeed enough to mention only the symbols that change. By default, any symbol not explicitly mentioned is left unchanged (although its profile may be updated according to the renaming specified for some sorts). This is illustrated here in STACK where there is no need to rename the constant *empty*, which will therefore have the same name for both lists and stacks. However, the induced signature morphism maps the constant symbol $empty : List$ into the constant symbol $empty : Stack$.

One can also explicitly rename a symbol to itself, say by writing '$empty \mapsto empty$', or just mention it without providing a new name, as in '**with** *empty*', which is equivalent to '**with** $empty \mapsto empty$'.

By default, overloaded symbols are renamed simultaneously. For instance, in INTEGER_ARITHMETIC_1 **with** $\_\_ + \_\_ \mapsto plus$, all the five overloaded infix '+' operations exported by INTEGER_ARITHMETIC_1 (see Chap. 5, p. 64) are renamed into five *plus* operations, with a functional syntax and the appropriate profiles.

In general, it is possible to specifically rename one of some overloaded symbols, by specifying its profile in the symbol map. For instance, in LIST_SET **with** $empty : List \mapsto nil$, only the constant *empty* of sort *List* is renamed into *nil*, while the constant *empty* of sort *Set* remains unchanged. However, care is needed in the presence of subsorts, since the signature morphism induced by the specified symbol map should preserve the overloading relations associated with subsorts. For instance, if we attempt to only rename in the specification INTEGER_ARITHMETIC_1 the addition '+' of two positive numbers into *plus* and write INTEGER_ARITHMETIC_1 **with** $\_\_ + \_\_ : Pos \times Pos \to Pos \mapsto plus$, we merely obtain an ill-formed specification. Thus in the specification INTEGER_ARITHMETIC_1, all the five overloaded '+' operations must be renamed simultaneously, again into five overloaded symbols.

When combining specifications, origins of symbols can be indicated.

spec LIST_SET_1 [**sort** *Elem*] =
LIST_SELECTORS [**sort** *Elem*] **with** *empty*, *cons*
and GENERATED_SET [**sort** *Elem*] **with** *empty*, {_}, _ ∪ __
then **op** *elements_of* __ : *List* → *Set*
∀ *e* : *Elem*; *L* : *List*
• *elements_of empty* = *empty*
• *elements_of cons*(*e*, *L*) = {*e*} ∪ *elements_of L*
end

Since, as explained above, '**with** *empty*, *cons*' means '**with** *empty* ↦ *empty*, *cons* ↦ *cons*', identity renaming can be used just to emphasize the fact that a given specification exports some symbols. This is illustrated in the specification LIST_SET_1 above, which is quite similar to LIST_SET, but for the fact that here we emphasize that LIST_SELECTORS exports in particular the operations *empty* and *cons*, and that GENERATED_SET exports in particular the operations *empty*, '{_}', and '∪'.

6.3 Hiding

Auxiliary symbols used in structured specifications can be hidden.

spec NATURAL_PARTIAL_SUBTRACTION_3 =
NATURAL_PARTIAL_SUBTRACTION_1 **hide** *suc*, *pre*
end

spec NATURAL_PARTIAL_SUBTRACTION_4 =
NATURAL_PARTIAL_SUBTRACTION_1
reveal *Nat*, *0*, *1*, _ + __, __ − __, __ * _, _ < __
end

When writing large specifications, it is quite frequent to rely on auxiliary operations (and predicates) to specify the operations (and predicates) of interest. Once these are defined, the auxiliary operations are no longer needed, and are better removed from the exported signature of the specification, which should include only the symbols that were required to be specified. This is the purpose of the **hide** construct.

Consider for instance the specification NATURAL_PARTIAL_SUBTRACTION_1 given in Chap. 4, p. 52. Once addition and subtraction are defined, the

two basic operations *suc* and *pre* are no longer needed (since $suc(x)$ is more conveniently written $x + 1$, and similarly $pre(x)$ is expressed by $x - 1$), and can therefore be hidden. This is illustrated by the specification NATURAL_PARTIAL_SUBTRACTION_3 given above.

Depending on the relative proportion of symbols to be hidden or not, in some cases it may be more convenient to explicitly list the symbols to be exported by a specification rather than those to be hidden. The construct '**reveal**' can be used for that purpose, and '**hide**' and '**reveal**' are just two symmetric constructs to achieve the same effect. The use of '**reveal**' is illustrated in NATURAL_PARTIAL_SUBTRACTION_4 above, and the reader can convince himself that both NATURAL_PARTIAL_SUBTRACTION_3 and NATURAL_PARTIAL_SUBTRACTION_4 export exactly the same symbols. However, in this case the first specification is clearly more concise. A more convincing example of the use of '**reveal**' is provided by the following example.

spec PARTIAL_ORDER_2 = PARTIAL_ORDER **reveal pred** $\_ \leq \_$

Similar rules to the ones explained for renaming apply to the **hide** and **reveal** constructs. One can qualify a symbol to be hidden or revealed by its kind (**sort**, **op** or **pred**), and by default, overloaded symbols are hidden (or revealed) simultaneously.

Note that hiding a sort entails hiding all the operations or predicates that use it in their profile. Similarly, revealing an operation or a predicate entails revealing all the sorts involved in its profile. For instance, in the specification PARTIAL_ORDER_2 above, revealing the predicate '$\leq$' entails revealing also the sort *Elem*.

As a consequence, hiding sorts should be used with care in the presence of subsorts. For instance, hiding the sort *Nat* in the specification POSITIVE given in Chap. 5, p. 61, leads to a specification of positive natural numbers with a sort *Pos* which has the expected carrier set, but without any operation or predicate available on it. Hiding the sort *Nat* in the specification POSITIVE_ARITHMETIC (see Chap. 5, p. 61) may seem more appropriate, but one should still note that the predicate '$<$' is no longer available in POSITIVE_ARITHMETIC **hide sort** *Nat*.

As a last remark, note that when convenient, **reveal** can be combined with a renaming of (some of) the exported symbols. For instance, in the above PARTIAL_ORDER_2 specification, we could have written '**reveal pred** $\_ \leq \_ \mapsto leq$' if, in addition to a restriction of the signature of PARTIAL_ORDER, we wanted to rename the infix predicate '$\_ \leq \_$' into a predicate *leq* with a functional notation.

6.4 Local Specifications

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

spec LIST_ORDER [TOTAL_ORDER **with sort** *Elem*, **pred** __ < __] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **op** $insert : Elem \times List \to List$
$\forall e, e' : Elem;\ L : List$
• $insert(e, empty) = cons(e, empty)$
• $insert(e, cons(e', L)) = cons(e', insert(e, L))\ when\ e' < e$
$else\ cons(e, cons(e', L))$
within **op** $order : List \to List$
$\forall e : Elem;\ L : List$
• $order(empty) = empty$
• $order(cons(e, L)) = insert(e, order(L))$
end

In many cases, auxiliary symbols are introduced for immediate use, and they do not need to be exported by the specification where they are declared. Then the best is to limit the scope of the declarations of such auxiliary symbols by using the '**local** ... **within** ...' construct. This is illustrated in the above specification LIST_ORDER, where the *insert* operation is introduced only for the purpose of the axiomatization of *order*. The declaration of *insert* has its scope limited to the part that follows '**within**', and *insert* is therefore not exported by the specification LIST_ORDER.

It is generally advisable to ensure that auxiliary symbols are declared in local parts of specifications.

spec LIST_ORDER_SORTED
[TOTAL_ORDER **with sort** *Elem*, **pred** _ < _] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
• $empty\ is\_sorted$
• $cons(e, empty)\ is\_sorted$
• $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
$cons(e', L)\ is\_sorted \wedge \neg(e' < e)$
within **op** $order : List \to List$
$\forall L : List \bullet order(L)\ is\_sorted$
end

The specification LIST_ORDER_SORTED above is a variant of the specification LIST_ORDER illustrating again the use of the '**local** ... **within** ...'

construct – this time to declare an auxiliary predicate. (Actually, the two specifications are not equivalent, since LIST_ORDER_SORTED is much looser and only requires that $order(L)$ is a sorted list, but perhaps not with the same elements as L.)

Care is needed with local sort declarations.

spec WRONG_LIST_ORDER_SORTED
[TOTAL_ORDER **with sort** *Elem*, **pred** $\_ < \_$] =
LIST_SELECTORS [**sort** *Elem*]
then local **pred** $\_\_is\_sorted : List$
sort $SortedList = \{L : List \bullet L\ is\_sorted\}$
$\forall e, e' : Elem;\ L : List$
• $empty\ is\_sorted$
• $cons(e, empty)\ is\_sorted$
• $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
$cons(e', L)\ is\_sorted \wedge \neg(e' < e)$
within op $order : List \to SortedList$
end

Note that the above specification WRONG_LIST_ORDER_SORTED, which may at first glance be considered as a slight variant of LIST_ORDER_SORTED, is *ill-formed*: *order* is exported by WRONG_LIST_ORDER_SORTED, and hence all sorts occurring in its profile should also be exported, which cannot be, since the sort *SortedList* is auxiliary. So, if the specifier really intends to insist that the result sort of *order* is *SortedList*, this subsort should be exported, as shown below.

spec LIST_ORDER_SORTED_2
[TOTAL_ORDER **with sort** *Elem*, **pred** $\_ < \_$] =
LIST_SELECTORS [**sort** *Elem*]
then local **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
• $empty\ is\_sorted$
• $cons(e, empty)\ is\_sorted$
• $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
$cons(e', L)\ is\_sorted \wedge \neg(e' < e)$
within sort $SortedList = \{L : List \bullet L\ is\_sorted\}$
op $order : List \to SortedList$
end

In fact the '**local** ... **within** ...' construct abbreviates a combination of extension and explicit hiding. The specification LIST_ORDER_SORTED_2 above, for instance, abbreviates:

spec LIST_ORDER_SORTED_3
[TOTAL_ORDER **with sort** *Elem*, **pred** $\_ < \_$] =
LIST_SELECTORS [**sort** *Elem*]
then **{** **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
• $empty\ is\_sorted$
• $cons(e, empty)\ is\_sorted$
• $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
$cons(e', L)\ is\_sorted \wedge \neg(e' < e)$
then **sort** $SortedList = \{L : List \bullet L\ is\_sorted\}$
op $order : List \to SortedList$
} hide $\_\_is\_sorted$
end

The main advantage of using the '**local** ... **within** ...' construct is that hiding the symbols introduced in the **local** part is left implicit. The convenience of this generally outweighs the danger of overlooking a locally-declared sort that is needed for the profile of an exported symbol. In any case, CASL allows both styles, and users can simply choose the one they prefer.

6.5 Named Specifications

Naming a specification allows its reuse.

It is in general advisable to define as many named specifications as felt appropriate, since this improves the reusability of specifications: a named specification can easily be reused by referring to its name.

Not only do the names serve as abbreviations when writing specifications, they also make it easy for readers to notice reuse. Moreover, when the name of a specification is aptly chosen, e.g., NATURAL_ARITHMETIC, readers may well be able to guess its signature – and perhaps even the specified axioms – from the name itself. (In Chap. 9, we shall see how named specifications and other items can be collected in libraries, and particular versions of them made available for use over the Internet.)

References to named specifications are particularly convenient for specifications structured using unions and extensions, where verbatim insertion of unnamed specifications would tend to obscure the structure. When needed, the signature of a referenced specification can be adjusted through appropriate combinations of renaming and hiding (although this should not often be necessary, provided that auxiliary symbols are made local, as explained in the previous section).