9

Libraries

Libraries are named collections of named specifications.

In the foregoing chapters, we have seen many examples of named specifications, and of references to them in later specifications. This chapter explains how a collection of named specifications can itself be named, as a *library*. The creation of libraries facilitates the *reuse* of specifications. For practical applications, it is important to be able to reuse (at least) existing specifications of basic datatypes, such as those described in Chap. 12.

Local libraries are self-contained.

A library is called *local* when it is self-contained, i.e., for each reference to a specification name in the library, the library includes a specification with that name. Local libraries might appear at first sight to be all that we need, but actually they provide poor support for reuse of specifications. The problem is that when a specification from one local library is reused in another, it has to be repeated *verbatim*. There is no formal link between the original specification and the copy, despite them having the same name: the names used in a library can be chosen freely, and different libraries could use the same name for completely different specifications.

Distributed libraries support reuse.

Distributed libraries allow duplication of specifications to be avoided altogether. Instead of making an explicit copy of a named specification from

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 111–122, 2004.

one library for use in another, the second library merely indicates that the specification concerned can be *downloaded* from the first one.

> *Different versions of the same library are distinguished by hierarchical version numbers.*

In practice, specifications *evolve*, e.g., to provide further operations or predicates on the specified sorts, or to define new subsorts. The libraries containing the specifications can evolve too, by adding or removing named specifications. Without some form of version control, even a trifling change in one library might cause specifications in other libraries to become ill-formed, or affect their meanings. CASL allows different versions of the same library to coexist (distinguishing them by hierarchical version numbers), and allows downloadings in a library to indicate that a particular version of another library is required.

Creation of new libraries is essential in connection with larger specification projects, and projects of any scale can benefit from reuse of specifications from existing libraries. The rest of this chapter illustrates the constructs used to specify local libraries, distributed libraries, and versions, and gives some advice on the organization of libraries.

9.1 Local Libraries

> *Local libraries are self-contained collections of specifications.*

library USERMANUAL/EXAMPLES
...
spec NATURAL = ...
...
spec NATURAL_ORDER = NATURAL **then** ...
...

The collection of all the illustrative examples given in the foregoing chapters is self-contained, so it could be made into a local library and named USERMANUAL/EXAMPLES, as outlined above. To provide a separate local library for each chapter would however involve a considerable amount of duplication, since many of the specifications that are defined in the earlier chapters are also referenced in later chapters (e.g., SET_PARTIAL_CHOOSE in Chap. 4 in-

stantiates GENERATED_SET, which is defined in Chap. 3). Using *distributed* libraries, as explained in Sect. 9.2, this duplication can be avoided.[1]

The 'same name, same thing' principle of CASL applies only within specifications, and it is possible for a library to include alternative specifications for the same symbols (e.g., using different sets of axioms). However, when such alternative specifications are both extended (perhaps indirectly) in the same specification, the principle does apply, and unintended name clashes might then arise. Thus in general, it is advisable for the developers of a library to respect the 'same name, same thing' principle when choosing symbols throughout the library. In any case, this is obviously helpful to those who might later browse the library. Alternative specifications for the same symbols should therefore be given in separate libraries.[2]

Specifications can refer to previous items in the same library.

library USERMANUAL/EXAMPLES
...
spec STRICT_PARTIAL_ORDER = ...
...
spec TOTAL_ORDER = STRICT_PARTIAL_ORDER **then** ...
...
spec PARTIAL_ORDER = STRICT_PARTIAL_ORDER **then** ...
...

Although we may often regard libraries as *sets* of named specifications, they are actually *sequences*, and the order in which the specifications occur is significant.

Specification names have linear visibility: each specification can refer only to the names of the specifications that precede it. Thus a series of extensions has to be presented in a bottom-up fashion, starting with a specification that is entirely self-contained, containing no references to other specifications at all. Each specification name in a library has a unique defining occurrence, so overriding cannot arise. Extensions that do not refer to each other may be given in any order (e.g., PARTIAL_ORDER above could just as well be given before TOTAL_ORDER).

Linear visibility of specification names means that mutual recursion between specifications is prohibited. When two specifications each make use of symbols declared in the other, the declarations of those symbols have to be duplicated, or moved to a preceding specification that can then be referenced by them both.

[1] A distributed library for each chapter of Part II is available via the CoFI web pages; copies are provided on the CD-ROM accompanying this book.

[2] If we intended our comprehensive USERMANUAL/EXAMPLES library for general use, we would remove all the illustrative alternative specifications.

All kinds of named specifications can be included in libraries.

```
library UserManual/Examples
...
spec Strict_Partial_Order = ...
...
spec Generic_Monoid [sort Elem] = ...
...
view Integer_as_Total_Order : ...
...
view List_as_Monoid [sort Elem] : ...
...
arch spec System = ...
...
unit spec Cont_Comp = ...
...
```

Items in libraries can be any kind of named specification, as illustrated above: simple named specifications, generic specifications, named view definitions, generic view definitions, and architectural and unit specifications. We shall henceforth refer to them generally as *library items*.

Libraries themselves never include anonymous specifications, such as declarations of sorts and operations. Moreover, the symbols declared by a library item are *not* automatically available for use in subsequent items: an explicit reference to the name of the library item is required to 'import' the item.

Technically, each library item is said to be *closed*, being interpreted without any pre-declared symbols at all. This facilitates downloading items from distributed libraries, see Sect. 9.2.

Display, parsing, and literal syntax annotations apply to entire libraries.

```
library UserManual/Examples
...
%display __<=__     %LATEX _ ≤ _
%display __>=__     %LATEX _ ≥ _
%display __union__  %LATEX _ ∪ _

%prec {_+_, _-_} < {_ * _}
%left_assoc __+__, _ * _
...
spec Strict_Partial_Order = ...
...
```

spec PARTIAL_ORDER = STRICT_PARTIAL_ORDER **then** $\ldots \leq \ldots$
...
spec GENERATED_SET [**sort** *Elem*] = $\ldots \cup \ldots$
...
spec INTEGER_ARITHMETIC_ORDER = $\ldots \leq \ldots \geq \ldots$
...

Annotations affecting the way terms are written or displayed apply to an entire library, and have to be collected at the beginning of the library. These annotations include display and precedence annotations, illustrated above.

Recall that various reserved words and symbols in CASL specifications are input in ASCII, but displayed as mathematical signs (e.g., universal quantification is input as '`forall`', and displayed as '$\forall$' when this sign is available in the current display format). *Display annotations* provide analogous flexibility for declared symbols. For example, the display annotations illustrated above determine how infix symbols input as '`<=`', '`>=`', and '`union`' are displayed when using LaTeX to format the specification. Note that a display annotation applies to all occurrences of the input symbol in the library, regardless of overloading.

Display annotations can give alternative displays for different formats: apart from LaTeX, both RTF and HTML are presently supported. The display of the annotation itself shows only the input syntax of the symbol and the result produced by the current formatter. The input form of one of the above annotations might be as follows:

```
%display __union__ %HTML __<sup>U</sup> %LATEX __\cup__
```

When no display annotation is given for a particular format, the input format itself is displayed. Thus the symbol displayed as '$\cup$' in the present LaTeX version of this User Manual would be displayed as '*union*' in an RTF version, unless the above annotation were to be extended with an RTF part.

Parsing annotations allow omission of grouping parentheses when terms are input. A single annotation can indicate the relative precedence or the associativity (left or right) of a group of operation symbols. The precedence annotation for infix arithmetic operations given above, namely:

%prec $\{\_+\_, \_-\_\} < \{\_*\_\}$

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

%left_assoc $\_+\_, \_*\_$

allows $(a + b) + c$ to be input as $a + b + c$, and similarly for $*$; but the parentheses cannot be omitted in $(a + b) - c$ (not even if '$\_-\_$' were to be included in the same left-associativity annotation).

When an operation symbol is declared with the associativity attribute *assoc*, an associativity *annotation* for that symbol is provided automatically.[3] Thus in practice, explicit associativity annotations are needed only for non-associative operations such as subtraction and division.

Libraries and library items can have author and date annotations.

library UserManual/Examples
%authors(Michel Bidoit `<bidoit@lsv.ens-cachan.fr>`,
Peter D. Mosses `<pdmosses@brics.dk>` **)%**
%dates 15 Oct 2003, 1 Apr 2000
...
spec Strict_Partial_Order = ...
...
%authors Michel Bidoit `<bidoit@lsv.ens-cachan.fr>`
%dates 10 July 2003
spec Integer_Arithmetic_Order =
...

An *author annotation* at the beginning of a library indicates the collective authorship of the entire library; one preceding an individual library item indicates its specific authorship.

A *date annotation* at the beginning of a library should indicate the release date of the current version of the library. It may also give the release dates of some previous major versions, possibly including that of the original version. A date annotation on an individual library item should indicate when that item was last changed, and (optionally) the dates of previous changes.

9.2 Distributed Libraries

Libraries can be installed on the Internet for remote access.

library Basic/Numbers
...
%left_assoc _@@_
%number _@@_
%floating _:::_ , _*E*_
%prec {_*E*_} < {_:::_}

[3] This implicit parsing annotation is local to the enclosing specification and to specifications that reference it, in contrast to ordinary parsing annotations.

```
...
spec  NAT =
      free type  Nat ::= 0 | suc(Nat)
      ...
      ops   1 : Nat = suc(0); ... ; 9 : Nat = suc(8);
            __@@__(m, n : Nat) : Nat = (m * suc(9)) + n
      ...
spec  INT = NAT then ...
spec  RAT = INT then ...
spec  DECIMALFRACTION = RAT then
      ...
      ops   __:::__ : Nat × Nat → Rat;
            __E__ : Rat × Int → Rat
      ...
```

The above example is an extract from one of the CASL libraries of basic datatypes, described in Chap. 12 and available on the Internet. It illustrates the overall structure of a library intended for general use, as well as some helpful annotations concerning literal syntax for numbers, which are explained below.

Validated libraries can be registered for public access.

CoFI will maintain a register of useful libraries. Registered CASL libraries are identified by hierarchical path names. For instance, all the CASL libraries of basic datatypes have names starting with 'BASIC/', and path names starting with 'CASL/' are reserved for libraries connected with the CASL language itself (e.g., the specification of the abstract syntax of CASL in CASL).

Registered libraries will be mirrored at several sites, to ensure their continuous accessibility. The URLs of a library can be obtained from the library name using a table provided on the CoFI web pages.

Libraries have to be *validated* before registration. The validation of a library ensures not only that it is well-formed, but also that semantic annotations expressing consistency of specifications (or conservativity over the parameters, in case of generic or unit specifications) have been added, and that all proof obligations (corresponding both to well-formedness conditions and to semantic annotations in the library) have been verified.

It is likely that new versions of existing libraries will be produced, e.g., providing further operations whose usefulness was not realized beforehand. Although the assignment and use of library version numbers allows users to protect their specifications from changes due to new versions (see Sect. 9.3), at least the names used in a registered library should not change much between versions.

Libraries should include appropriate annotations.

In particular, parsing and display annotations can be provided, as explained in Sect. 9.1. The above example illustrates a further kind of annotation, used to provide *literal syntax* for numbers in CASL. The effect of the illustrated annotations is that, after downloading the appropriate specifications from the library BASIC/NUMBERS, conventional decimal notation can be used for integers and decimal fractions, e.g., *42*, *2.718*, *10E−12*. The digits *42* are interpreted as the term *4*@@*2*, and *2.718* is interpreted as the term *2*:::*718* (where *718* is subsequently interpreted as (*7*@@*1*)@@*8*). The definition of the operation __@@__ is shown above; those of __:::__ and __*E*__ are a bit more involved, and omitted here. Notice that the library BASIC/NUMBERS is *not* hard-wired into CASL, and users could provide annotations to interpret the literal syntax for integers and decimal fractions as terms involving different operations, e.g., on different sorts.

Libraries can include items downloaded from other libraries.

library BASIC/STRUCTUREDDATATYPES
...
from BASIC/NUMBERS **get** NAT, INT
...
spec LIST [**sort** *Elem*] **given** NAT = ...
...
spec ARRAY ... **given** INT = ...
...

Individual specifications and other items can be *downloaded* from other libraries. For example, the library BASIC/STRUCTUREDDATATYPES, outlined above, does not itself provide the specifications of natural numbers and integers that are needed in the specifications LIST and ARRAY, but instead downloads NAT and INT from the BASIC/NUMBERS library.

The names of the items to be downloaded from a library have to be listed explicitly: one cannot request the downloading of all the items that happen to be provided by a library. However, although the name of each item provided by a downloading is always explicit, no indication is given of its kind (i.e., whether it is an ordinary, generic, or architectural specification, or a view) nor of what symbols it declares. Thus the well-formedness of a library depends on what items are actually downloaded from other libraries.

The construct '**from** ... **get** ...' above has the effect of downloading the specifications that are named NAT and INT in BASIC/NUMBERS, preserving their names. It is also possible to give downloaded specifications different

names, e.g., to avoid clashes with specification names that are already in use locally:

from Basic/Numbers **get** Nat ↦ Natural, Int ↦ Integer

Items that are referenced by downloaded items are *not* themselves automatically downloaded, e.g., downloading Int does not entail the downloading of Nat. This is because downloading involves the *semantics* of the named items, not their *text*. The semantics of Int consists of a signature and a class of models, and is a self-contained entity – recall from Chap. 6 that the models of a structured specification have no more structure than do those of a flat, unstructured specification. Thus downloading Int gives exactly the same result as if the reference to Nat in its text had already been replaced by the text of Nat *before* downloading. For the same reason, the presence of another item with the name Nat in the current library makes no difference to the result of downloading Int. In terms of software packages, downloading specifications from Casl libraries is analogous to installing packages from binaries, rather than from sources.

Downloading any item from another library B in a library A causes all the parsing and display annotations of B to be inserted at the beginning of A. (Conflicting annotations from different libraries are ignored altogether, and local annotations override conflicting downloaded annotations.) The copied annotations allow terms to be written and displayed in A in the same way as in B.

Substantial libraries of basic datatypes are already available.

The organization of the following libraries of basic datatypes is explained in Chap. 12:

Basic/Numbers: natural numbers, integers, and rationals.
Basic/RelationsAndOrders: reflexive, symmetric, and transitive relations, equivalence relations, partial and total orders, boolean algebras.
Basic/Algebra_I: monoids, groups, rings, integral domains, and fields.
Basic/SimpleDatatypes: booleans, characters.
Basic/StructuredDatatypes: sets, lists, strings, maps, bags, arrays, trees.
Basic/Graphs: directed graphs, paths, reachability, connectedness, colorability, and planarity.
Basic/Algebra_II: monoid and group actions on a space, euclidean and factorial rings, polynomials, free monoids, and free commutative monoids.
Basic/LinearAlgebra_I: vector spaces, bases, and matrices.
Basic/LinearAlgebra_II: algebras over a field.
Basic/MachineNumbers: bounded subtypes of naturals and integers.

These libraries form a coherent collection of highly-polished specifications. The libraries themselves are provided in full in the CASL *Reference Manual* [20], and are available on the Internet.

> *Libraries need not be registered for public access.*

library `http://www.cofi.info/CASL/Test/Security`
...
from `http://casl:password@www.cofi.info/CASL/RSA` **get** KEY
...
spec DECRYPT = KEY **then** ...
...

Libraries under development, and libraries provided for restricted groups of users, are named and accessed by their URLs (instead of the simple path names used for registered libraries). This allows the CASL library constructs to be fully exploited for libraries that are not yet – and perhaps never will be – registered for public access. Moreover, validation of libraries can be a demanding and time-consuming process, and getting a library approved and registered is appropriate only when it provides specifications that are likely to be found useful by persons not directly involved in its development.

The primary Internet access protocols HTTP and FTP both support password protection of libraries and the insertion of usernames and passwords in URLs allows downloading between protected libraries (With HTTP, the username and password can be unrelated to those used for the host file system.)

9.3 Version Control

> *Subsequent versions of a library are distinguished by explicit version numbers.*

library BASIC/NUMBERS **version** 1.0
...
spec NAT = ...
...
spec INT = NAT **then** ...
...
spec RAT = INT **then** ...
...

As illustrated above, a library can be assigned an explicit version number, allowing it to be distinguished from previous and future versions of the same library. CASL allows conventional hierarchical version numbers, familiar from version numbers of software packages: the initial digits indicate a major version, digits after a dot indicate sub-versions, and digits after a further dot indicate patches to correct bugs. (Distinctions between alpha, beta, and other pre-release versions are not supported.)

The smallest version number is written simply '0', and can be omitted when specifying the initial version of a library; this is the case with the version of BASIC/NUMBERS shown in Sect. 9.2, it implicitly has version number '0', but in general the first-installed version of a library could have any version number at all. The numbers of successively installed versions do not have to be contiguous, nor even increasing: e.g., a patched version 0.99.1 could be installed after version 1.0.

Individual library items do not have separate version numbers. Date annotations can be used to indicate which items have changed between two versions of a library.

Libraries can refer to specific versions of other libraries.

library BASIC/STRUCTUREDDATATYPES **version** 1.0
...
from BASIC/NUMBERS **version** 1.0 **get** NAT, INT
...
spec LIST [**sort** *Elem*] **given** NAT = ...
...
spec ARRAY ... **given** INT = ...
...

Downloading items from particular versions of libraries is necessary if one wants to ensure coherence between libraries. For example, as illustrated above, version 1.0 of BASIC/STRUCTUREDDATATYPES downloads NAT and INT from version 1.0 of BASIC/NUMBERS. Omitting the version number when downloading gives implicitly the *current version* of the library, which may of course change. By the way, the current version of a library is *not* necessarily the one most recently installed: it is the one with the *largest version number*. As previously mentioned, a patched version 0.99.1 could be installed after version 1.0, but a downloading without an explicit version number would still refer to version 1.0.

Even though the developers of libraries may try to ensure backwards compatibility between versions, it could happen that symbols introduced in a new version of a downloaded specification clash with symbols already in use in the library that specified the downloading, causing ill-formedness or inconsistency. So for safety, it is advisable to give explicit version numbers when

downloading (also when downloading from version '0' of another library). If one subsequently wants to use symbols that are introduced only in some later version of another library, all that is needed is to change the version number in the downloading(s).

An alternative strategy is to ensure consistency with the *current* versions of all libraries from which specifications are downloaded, by observing the changes in the new versions and adapting the downloading library accordingly. For instance, one might download INT from the current version of BASIC/NUMBERS, instead of from version 1.0 of that library. This may involve extra work when a new version of BASIC/NUMBERS appears, but it has several advantages over the more cautious approach. CASL leaves the choice to the user, although registered libraries will generally be required to use explicit version numbers when downloading from other libraries.

All downloadings should be collected at the beginning of a library.

Although CASL allows downloadings to be interleaved with specification definitions, it is advisable to collect the downloadings at the beginning of libraries (together with any parsing and display annotations). This makes it easy to see dependencies between libraries, and to ensure that different downloadings from the same library all refer to the same version of it.