3

Getting Started

Simple specifications may be written in CASL essentially as in many other algebraic specification languages.

The simplest kind of algebraic specification is when each specified operation is to be interpreted as an ordinary *total* mathematical function: it takes values of particular types as arguments, and always returns a well-defined value. Total functions correspond to software whose execution always terminates normally. The types of values are named by simple symbols called *sorts*.

In practice, a realistic software specification involves *partial* as well as total functions. However, it may well be formed from simpler specifications, some of which involve only total functions. This chapter explains how to express such simple specifications in CASL, illustrating various features of the language.

The simple specifications discussed in this chapter can also be expressed in many previous specification languages; it is usually straightforward to reformulate them in CASL. Readers who know other specification languages will probably recognize some familiar examples among the illustrations given in this chapter.

CASL provides also useful abbreviations.

The technique of algebraic specification by axioms is generally well-suited to expressing properties of functions. However, when functions have commonly-occurring mathematical properties, it can be tedious to give the corresponding axioms explicitly. CASL provides some useful abbreviations for such cases. Similarly, so-called free datatype declarations allow sorts and value constructors to be specified much as in functional programming languages, using a concise and suggestive notation.

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 23–45, 2004.

CASL allows loose, generated and free specifications.

The models of a loose specification include all those where the declared functions have the specified properties, without any restrictions on the sets of values corresponding to the various sorts. In models of a generated specification, in contrast, it is required that all values can be expressed by terms formed from the specified constructors, i.e. unreachable values are prohibited. In models of free specifications, it is required that values of terms are distinct except when their equality follows from the specified axioms: the possibility of unintended coincidence between them is prohibited.

Section 3.1 below focuses on loose specifications; Sect. 3.2 discusses the use of generated specifications, and Sect. 3.3 does the same for free specifications. Loose, generated, and free specifications are often used together in CASL: each style has its advantages in particular circumstances, as explained below in connection with the illustrative examples.

3.1 Loose Specifications

CASL syntax for declarations and axioms involves familiar notation, and is mostly self-explanatory.

spec STRICT_PARTIAL_ORDER =
%% Let's start with a simple example !
sort *Elem*
pred __ < __ : *Elem* × *Elem* %% **pred** abbreviates predicate
$\forall x, y, z : Elem$
- $\neg(x < x)$ %(strict)%
- $x < y \Rightarrow \neg(y < x)$ %(asymmetric)%
- $x < y \wedge y < z \Rightarrow x < z$ %(transitive)%

%{ Note that there may exist x, y such that
neither $x < y$ nor $y < x$. }%
end

The above (basic) specification, named STRICT_PARTIAL_ORDER, introduces a sort *Elem* and a binary infix predicate symbol '<'. In the declaration of a predicate symbol, argument sorts are separated by the sign '×', which can be input directly as such in ISO Latin-1 or as '*' in ASCII. Note that CASL allows so-called *mixfix notation*, i.e., the specifier is free to indicate, using '__' (pairs of underscores) as *place-holders*, how to place arguments when building

terms (single underscores are treated as letters in identifiers).[1] Using mixfix notation generally allows the use of familiar mathematical and programming notations, which contributes substantially to the readability of specifications.

The interpretation of the binary predicate symbol '$<$' is then constrained by three axioms. A set of axioms is generally presented as a 'bulleted' list of formulas, preceded by the universally quantified declaration of the relevant variables, together with their respective sorts, as shown in the above example. In CASL, axioms are written in first-order logic with equality, using quantifiers and the usual logical connectives. The universal quantification preceding a list of axioms applies to the entire list. Axioms can be annotated by labels written %(...)%, which is convenient for proper reference in explanations or by tools.

Note that '$\forall$' is input as '`forall`', and that '$\bullet$' is input as '`.`' or '$\cdot$'. The usual logical connectives '$\Rightarrow$', '$\Leftrightarrow$', '$\wedge$', '$\vee$', and '$\neg$', are input as '`=>`', '`<=>`', '`/\`', '`\/`', and '`not`', respectively; '$\neg$' can also be input directly as an ISO Latin-1 character. The existential quantifier '$\exists$' is input as '`exists`', and '$\exists!$' is input as '`exists!`'.

It is advisable to comment as appropriate the various elements introduced in a specification. The syntax for end-of-line and grouped multi-line comments is illustrated in the above example. The '**end**' keyword ending a specification is optional.

The above STRICT_PARTIAL_ORDER specification is loose in the sense that it has many (non-isomorphic) models, among which models where '$<$' is interpreted by a total ordering relation and models where it is interpreted by a partial one.

Specifications can easily be extended by new declarations and axioms.

spec TOTAL_ORDER =
STRICT_PARTIAL_ORDER
then $\forall x, y : Elem \bullet x < y \vee y < x \vee x = y$ %(total)%
end

Extensions, introduced by the keyword '**then**', may specify new symbols, possibly constrained by some axioms, or merely require further properties of old ones, as in the above TOTAL_ORDER example, or more generally do both at the same time. In TOTAL_ORDER, we further constrain the interpretation of the predicate symbol '$<$' by requiring it to be a total ordering relation.

All symbols introduced in a specification are by default exported by it and visible in its extensions. This is for instance the case here for the sort *Elem* and

[1] Mixfix notation is so-called because it generalizes infix, prefix, and postfix notation to allow arbitrary mixing of argument positions and identifier tokens.

the predicate symbol '$<$', which are introduced in STRICT_PARTIAL_ORDER, exported by it, and therefore available in TOTAL_ORDER.[2]

> *In simple cases, an operation (or a predicate) symbol may be declared and its intended interpretation defined at the same time.*

spec TOTAL_ORDER_WITH_MINMAX =
TOTAL_ORDER
then **ops** $min(x, y : Elem) : Elem = x \ when \ x < y \ else \ y$;
$max(x, y : Elem) : Elem = y \ when \ min(x, y) = x \ else \ x$
end

TOTAL_ORDER_WITH_MINMAX extends TOTAL_ORDER by introducing two binary operation symbols *min* and *max*, for which a functional notation is to be used, so no place-holders are given. The intended interpretation of the symbol *min* is defined simultaneously with its declaration, and the same is done for *max*. For instance:

op $min(x, y : Elem) : Elem = x \ when \ x < y \ else \ y$

abbreviates:

op $min : Elem \times Elem \to Elem$
$\forall x, y : Elem \bullet min(x, y) = x \ when \ x < y \ else \ y$

(and similarly for *max*). As for predicate symbol declarations, in an operation symbol declaration, the argument sorts are separated by the sign '$\times$'; the result sort is preceded by '$\to$', which is input as '`->`'.

The '... *when* ... *else* ...' construct used above is itself an abbreviation, and:

$min(x, y) = x \ when \ x < y \ else \ y$

abbreviates:

$(x < y \Rightarrow min(x, y) = x) \wedge (\neg(x < y) \Rightarrow min(x, y) = y)$

In CASL specifications, visibility is *linear*, i.e., any symbol must be declared before being used. In the above example, *min* should be declared before being used to define *max*.

Linear visibility does not imply, however, that a fixed scheme is to be used when writing specifications: the specifier is free to present the required declarations and axioms in any order, as long as the linear visibility rule is respected. For instance, one may prefer to declare first all sorts and all operation

[2] See Chap. 6 for constructs allowing the explicit restriction of the set of symbols exported by a specification.

or predicate symbols needed, and then specify their properties by the relevant axioms. Or, in contrast, one may prefer to have each operation or predicate symbol declaration immediately followed by the axioms constraining its interpretations. Both styles are equally fine, and can even be mixed if desired. This flexibility is illustrated in the following variant of the TOTAL_ORDER_WITH_MINMAX specification, where for explanatory purposes we refrain from using the useful abbreviations explained above.

spec VARIANT_OF_TOTAL_ORDER_WITH_MINMAX =
TOTAL_ORDER
then **vars** $x, y : Elem$
op $min : Elem \times Elem \to Elem$
• $x < y \Rightarrow min(x, y) = x$
• $\neg(x < y) \Rightarrow min(x, y) = y$
op $max : Elem \times Elem \to Elem$
• $x < y \Rightarrow max(x, y) = y$
• $\neg(x < y) \Rightarrow max(x, y) = x$
end

Note that in order to avoid the tedious repetition of the declaration of the variables x and y for each list of axioms, we have used here a *global* variable declaration which introduces x and y for the rest of the specification. Variable declarations are of course not exported across specification extensions: the variables x and y declared in VARIANT_OF_TOTAL_ORDER_WITH_MINMAX are not visible in any of its extensions.

Symbols may be conveniently displayed as usual mathematical symbols by means of ***%display*** *annotations.*

%display __<=__ %LATEX __ $\leq$ __

spec PARTIAL_ORDER =
STRICT_PARTIAL_ORDER
then **pred** __ $\leq$ __$(x, y : Elem) \Leftrightarrow (x < y \vee x = y)$
end

The above example relies on a **%display** annotation: while, for obvious reasons, the specification text should be *input* using the ISO Latin-1 character set, it is often convenient to *display* some symbols differently, e.g., as mathematical symbols. This is the case here where the '<=' predicate symbol is more conveniently displayed as '$\leq$'. Display annotations, as any other CASL annotations, are auxiliary parts of a specification, for use by tools, and do not affect the semantics of the specification.[3]

[3] Display annotations should be provided at the beginning of a library, and are explained in more detail in Chap. 9.

In the above example, we have again used the facility of simultaneously declaring and defining a symbol (here, the predicate symbol '$\leq$') in order to obtain a more concise specification.

*The **%implies** annotation is used to indicate that some axioms are supposedly redundant, being consequences of others.*

spec PARTIAL_ORDER_1 =
PARTIAL_ORDER
then %implies
$\forall x, y, z : Elem \bullet x \leq y \wedge y \leq z \Rightarrow x \leq z$ %(transitive)%
end

The **%implies** annotation above is used to emphasize that the transitivity of '$\leq$' should follow from the other axioms, or, in other words, that the model class of PARTIAL_ORDER_1 is exactly the same as the model class of PARTIAL_ORDER. The **%implies** annotation applies to the whole of the specification extension where it occurs (which happens here to introduce a single axiom).

Note however that an annotation does not affect the semantics of a specification, hence removing the **%implies** annotation does not change the class of models of the above specification. The sole aim of an **%implies** annotation is to stress the specifier's intentions, and it will also help readers confirm their understanding. Some tools may of course use such annotations to generate corresponding proof obligations. For instance, here, the proof obligation is:

$\text{PARTIAL\_ORDER} \models \forall x, y, z : Elem \bullet x \leq y \wedge y \leq z \Rightarrow x \leq z$

Discharging these proof obligations increases the trustworthiness of a specification.

To fully understand that an **%implies** annotation has no effect on the semantics, the best is to consider an example where the corresponding proof obligation cannot be discharged, as shown below.

spec IMPLIES_DOES_NOT_HOLD =
PARTIAL_ORDER
then %implies
$\forall x, y : Elem \bullet x < y \vee y < x \vee x = y$ %(total)%
end

Since the loose specification PARTIAL_ORDER has models where '$<$' is interpreted by a partial ordering relation, the proof obligation corresponding to the above **%implies** annotation cannot be discharged. However, since annotations have no impact on the semantics, the specification IMPLIES_DOES_NOT_HOLD is well-formed and just constrains the interpretation of '$<$' to be a total ordering relation. The fact that the proof obligation cannot be discharged merely points out a potential mistake in the specification.

Attributes may be used to abbreviate axioms for associativity, commutativity, idempotence, and unit properties.

spec MONOID =
  sort *Monoid*
  ops *1* : *Monoid*;
    $\_ * \_$: *Monoid* × *Monoid* → *Monoid*, *assoc*, *unit 1*
end

The above example introduces a constant symbol *1* of sort *Monoid*, then a binary operation symbol '$*$', which is asserted to be associative and to have *1* as unit element. (Note that there is no '→' sign before the sort when declaring a constant.) The *assoc* attribute abbreviates, as expected, the following axiom:

$$\forall x, y, z : Monoid \bullet (x * y) * z = x * (y * z)$$

The '*unit 1*' attribute abbreviates:

$$\forall x : Monoid \bullet (x * 1 = x) \wedge (1 * x = x)$$

Note that to make the use of '*unit 1*' legal, it is necessary to have previously declared the constant *1*, to respect the linear visibility rule.

Other available attributes are *comm*, which abbreviates the obvious axiom stating that a binary operation is commutative, and *idem*, which can be used to assert the idempotence of a binary operation f (i.e., that $f(x, x) = x$).

Asserting '$*$' to be associative using the attribute *assoc* makes the term $x * y * z$ well-formed (assuming x, y, z of the right sort), while otherwise grouping parentheses would be required. Moreover, it is expected that some tools (e.g., systems based on rewriting) may make special use of the *assoc* attribute, so it is generally advisable to use this attribute instead of stating the same property by an axiom (the same applies to the other attributes).

Genericity of specifications can be made explicit using parameters.

spec GENERIC_MONOID [**sort** *Elem*] =
  sort *Monoid*
  ops *inj* : *Elem* → *Monoid*;
    1 : *Monoid*;
    $\_ * \_$: *Monoid* × *Monoid* → *Monoid*, *assoc*, *unit 1*
  $\forall x, y : Elem \bullet inj(x) = inj(y) \Rightarrow x = y$
end

The above example describes monoids built over arbitrary elements (of sort *Elem*). The intention here is to reuse the specification GENERIC_MONOID to derive from it specifications of monoids built over, say, characters, symbols, etc. In such cases, it is appropriate to emphasize the intended genericity of the specification by making explicit, in a distinguished *parameter part* (which is here [**sort** *Elem*]), the piece of specification that is intended to vary in the derived specifications. In these, it will then be possible to *instantiate* the parameter part as desired in order to specialize the specification as appropriate (to obtain, e.g., a specification of monoids built over characters). A named specification with one or more parameter(s) is called *generic*.

The body of the generic specification GENERIC_MONOID is an extension of what is specified in the parameter part. Hence an alternative to the above generic specification GENERIC_MONOID is the following, less elegant, non-generic specification (which cannot be specialized by instantiation):

spec NON_GENERIC_MONOID =
  sort *Elem*
then **sort** *Monoid*
  ops *inj* : $Elem \to Monoid$;
    1 : *Monoid*;
    $\_ * \_ : Monoid \times Monoid \to Monoid$, *assoc*, *unit 1*
  $\forall x, y : Elem \bullet inj(x) = inj(y) \Rightarrow x = y$
end

A generic specification may have more than one parameter, and parameters can be arbitrary specifications, named or not. When reused by reference to its name, a generic specification *must* be instantiated. Generic specifications and how to instantiate them are discussed in detail later in Chap. 7. Using generic specifications when appropriate improves the reusability of specification definitions.

> *References to generic specifications always instantiate the parameters.*

spec GENERIC_COMMUTATIVE_MONOID [**sort** *Elem*] =
  GENERIC_MONOID [**sort** *Elem*]
then $\forall x, y : Monoid \bullet x * y = y * x$
end

The above (generic) specification GENERIC_COMMUTATIVE_MONOID is defined as an extension of GENERIC_MONOID, which should therefore be instantiated, as explained above. Instantiating a generic specification is done by providing an argument specification that 'fits' the parameter part of the generic specification to be instantiated.

It is however quite frequent that the instantiation is 'trivial', i.e., the argument specification is identical to the parameter one. This is the case for the above example, where the generic specification GENERIC_MONOID is instantiated by providing the same argument specification '**sort** *Elem*' as the original parameter.

spec GENERIC_COMMUTATIVE_MONOID_1 [**sort** *Elem*] =
GENERIC_MONOID [**sort** *Elem*]
then **op** __ ∗ __ : *Monoid* × *Monoid* → *Monoid*, *comm*
end

GENERIC_COMMUTATIVE_MONOID_1 is an alternative version of the former specification where, instead of requiring explicitly with an axiom the commutativity of the operation '∗', we require it using the attribute *comm*. As explained before, it is in general better to describe such requirements using attributes rather than explicit axioms, since it is expected that some tools will rely on these attributes for specialized algorithms (e.g., AC term rewriting).

This example illustrates also an important feature of CASL, the *'same name, same thing'* principle. The operation symbol '∗' is indeed declared twice, with the same profile, first in GENERIC_MONOID and then again in GENERIC_COMMUTATIVE_MONOID_1 (the second declaration being enriched by the attribute *comm*). This is perfectly fine and defines only one binary operation symbol '∗' with the corresponding profile, according to the 'same name, same thing' principle. This principle applies to sorts, as well as to operation and predicate symbols. It applies both to symbols defined locally and to symbols imported from an extended specification, as it is the case here for '∗'. Of course, it does not apply between separate named specifications, i.e., the same symbol may be used in different named specifications with entirely different interpretations.

Note that for operation and predicate symbols, the 'same name, same thing' principle is a little more subtle than for sorts: the 'name' of an operation (or of a predicate) includes its profile of argument and result sorts, so two operations defined with the same symbol but with different profiles do not have the same 'name', the symbol is just *overloaded*. When an overloaded symbol is used, the intended profile is to be determined by the context (e.g., the sorts of the arguments to which the symbol is applied).[4] Explicit disambiguation can be used when needed, by specifying the profile (or result sort) in an application.[5] Note that overloaded constants are allowed in CASL (e.g., *empty* may be declared to be a constant of various sorts of collections).

[4] See also the discussion of overloading in presence of subsorts in Chap. 5, p. 61.

[5] For instance, depending on the context, the term $t1 * t2$ can be disambiguated by writing $(op\ * : Monoid \times Monoid \to Monoid)(t1, t2)$, or just $(t1 : Monoid) * (t2 : Monoid)$, or even $(t1 * t2) : Monoid$.

Datatype declarations may be used to abbreviate declarations of sorts and constructors.

spec CONTAINER [**sort** *Elem*] =
  type *Container* ::= *empty* | *insert*(*Elem*; *Container*)
  pred *__is_in__* : *Elem* × *Container*
  ∀ *e*, *e*′ : *Elem*; *C* : *Container*
  • ¬(*e is_in empty*)
  • *e is_in insert*(*e*′, *C*) ⇔ (*e* = *e*′ ∨ *e is_in C*)
end

Specifications of 'datatypes' with constructors are frequently needed. CASL provides special constructs for datatype declarations to abbreviate the corresponding rather tedious declarations. For instance, the above datatype declaration:

type *Container* ::= *empty* | *insert*(*Elem*; *Container*)

abbreviates:

sort *Container*
ops *empty* : *Container*;
  insert : *Elem* × *Container* → *Container*

A datatype declaration looks like a context-free grammar in a variant of BNF. It declares the symbols on the left of '::=' as sorts, and for each alternative on the right it declares a constructor.

A datatype declaration as the one above is *loose* since it does not imply any constraint on the values of the declared sorts: there may be some values of sort *Container* that are not built by any of the declared constructors, and the same value may be built by different applications of the constructors to some arguments.

Datatype declarations may also be specified as generated (see Sect. 3.2) or free (see Sect. 3.3). Moreover, selectors, which are usually partial operations, may be specified for each component (see Chap. 4).

Loose datatype declarations are appropriate when further constructors may be added in extensions.

spec MARKING_CONTAINER [**sort** *Elem*] =
  CONTAINER [**sort** *Elem*]
then **type** *Container* ::= *mark_insert*(*Elem*; *Container*)

pred $\_\_is\_marked\_in\_\_ : Elem \times Container$
$\forall e, e' : Elem;\ C : Container$
- $e\ is\_in\ mark\_insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
- $\neg(e\ is\_marked\_in\ empty)$
- $e\ is\_marked\_in\ insert(e', C) \Leftrightarrow e\ is\_marked\_in\ C$
- $e\ is\_marked\_in\ mark\_insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_marked\_in\ C)$

end

The above specification extends CONTAINER (trivially instantiated) by introducing another constructor *mark_insert* for the sort *Container* (hence, values added to a container may now be 'marked' or not). Note that we heavily rely on the 'same name, same thing' principle here, since it ensures that the sort *Container* introduced by the datatype declaration of CONTAINER and the sort *Container* introduced by the datatype declaration of MARKING_CONTAINER are the same sort, which implies that the combination of both datatype declarations is equivalent to:

type $Container ::= empty \mid insert(Elem;\ Container) \mid mark\_insert(Elem;\ Container)$

Note that since 'new' values may be constructed by *mark_insert*, it is necessary to extend the specification of the predicate symbol *is_in* by an extra axiom taking care of the newly introduced constructor.

3.2 Generated Specifications

Sorts may be specified as generated by their constructors.

spec GENERATED_CONTAINER [**sort** *Elem*] =
generated type $Container ::= empty \mid insert(Elem;\ Container)$
pred $\_\_is\_in\_\_ : Elem \times Container$
$\forall e, e' : Elem;\ C : Container$
- $\neg(e\ is\_in\ empty)$
- $e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$

end

When a datatype is declared as *generated*, as in the above example, the corresponding sort is constrained to be generated by the declared constructors, which means that any value of this sort is built by application of constructors. This constraint is sometimes referred to as the *'no junk'* principle. For instance, in the above example, having declared the datatype *Container* to be generated entails that in any model of GENERATED_CONTAINER, any value of

sort *Container* is denotable by a term built with *empty*, *insert*, and variables of sort *Elem* only.

As a consequence, properties of values of sort *Container* can be proved by induction on the declared constructors. A major benefit of generated datatypes is indeed that induction on the declared constructors is a sound proof principle.

The construct '**generated type** ...' used above is just an abbreviation for '**generated { type** ... **}**', and '**generated**' can be used around arbitrary signature declarations, enclosed in '**{**' and '**}**'.

Generated specifications are in general loose.

spec GENERATED_CONTAINER_MERGE [**sort** *Elem*] =
GENERATED_CONTAINER [**sort** *Elem*]
then **op** $\_\_merge\_\_ : Container \times Container \to Container$
$\forall e : Elem;\ C, C' : Container$
• $e\ is\_in\ (C\ merge\ C') \Leftrightarrow (e\ is\_in\ C \vee e\ is\_in\ C')$
end

A generated specification is in general loose. For instance, GENERATED_CONTAINER is loose since, although all values of sort *Container* are specified to be generated by *empty* and *insert*, the behavior of the *insert* constructor is still loosely specified (nothing is said about the case where an element is inserted into a container which already contains this element). Hence GENERATED_CONTAINER admits several non-isomorphic models.

GENERATED_CONTAINER_MERGE is as loose as GENERATED_CONTAINER with respect to *insert*, and in addition, the newly introduced operation symbol *merge* is also loosely specified: nothing is said about what happens when merging two containers which share some elements.

It is important to understand that looseness of a specification is not a problem, but on the contrary avoids unnecessary overspecification. In particular, loose specifications are in general well-suited to capturing requirements.

The fact that *merge* is loosely specified does not mean that it can produce new values of the sort *Container*. On the contrary, since this sort has been specified as being generated by *empty* and *insert*, it follows that any value denotable by a term of the form *merge*(...,...) can also be denoted by a term built with *empty* and *insert* (and no *merge*). Hence, for the specification GENERATED_CONTAINER_MERGE, proofs by induction on *Container* only need to consider *empty* and *insert*, and not *merge*, as was the case for GENERATED_CONTAINER.

Generated specifications need not be loose.

spec GENERATED_SET [**sort** *Elem*] =
  generated type $Set ::= empty \mid insert(Elem;\ Set)$
  pred $\_\_is\_in\_\_ : Elem \times Set$
  ops $\{\_\}(e : Elem) : Set = insert(e, empty)$;
    $\_\_ \cup \_\_ : Set \times Set \to Set$;
    $remove : Elem \times Set \to Set$
  $\forall e, e' : Elem;\ S, S' : Set$
  • $\neg(e\ is\_in\ empty)$
  • $e\ is\_in\ insert(e', S) \Leftrightarrow (e = e' \vee e\ is\_in\ S)$
  • $S = S' \Leftrightarrow (\forall x : Elem \bullet x\ is\_in\ S \Leftrightarrow x\ is\_in\ S')$ %(equal_sets)%
  • $e\ is\_in\ (S \cup S') \Leftrightarrow (e\ is\_in\ S \vee e\ is\_in\ S')$
  • $e\ is\_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is\_in\ S)$
then %implies
  $\forall e, e' : Elem;\ S : Set$
  • $insert(e, insert(e, S)) = insert(e, S)$
  • $insert(e, insert(e', S)) = insert(e', insert(e, S))$
  generated type $Set ::= empty \mid \{\_\_\}(Elem) \mid \_\_ \cup \_\_(Set;\ Set)$
  op $\_\_ \cup \_\_ : Set \times Set \to Set,\ assoc,\ comm,\ idem,\ unit\ empty$
end

Although generated specifications are in general loose, they need not be so, as illustrated by the above GENERATED_SET specification, where the axiom %(equal_sets)%, combined with the axioms defining *is_in*, fully constrains (up to isomorphism) the interpretations of the sort *Set* and of the constructors *empty* and *insert*, once an interpretation for the sort *Elem* is chosen.

Note also that this example displays the power of the annotation **%implies**. Remember that this annotation applies to the whole of the specification extension where it occurs, so here it applies not only to the two explicit axioms about *insert*, but also to the properties corresponding to the attributes of '$\cup$' as well as to the generatedness constraint. Hence, the **%implies** annotation is used here not only to stress that the usual properties of *insert* are expected to follow from the preceding declarations and axioms, but also that an alternative induction scheme, based on *empty*, $\{\_\_\}$ and $\_\_ \cup \_\_$, can be used for sets. Moreover, it asserts that $\_ \cup \_$ is expected to be associative, commutative, idempotent (i.e., $S \cup S = S$), and to have *empty* as unit. Note again that this **%implies** part heavily relies on the 'same name, same meaning' principle.

Generated types may need to be declared together.

The following specification fragment illustrates what may go wrong.

sort *Node*
generated type *Tree* ::= *mktree*(*Node*; *Forest*)
generated type *Forest* ::= *empty* | *add*(*Tree*; *Forest*)

The above is *incorrect*, due to the linear visibility rule. This can easily be fixed by replacing '**sort** *Node*' by '**sorts** *Node*, *Tree*, *Forest*'. Even when corrected, the above is *wrong*, since the corresponding semantics is not what a naive reader may expect. One may expect that only models where the carrier sets of the sorts *Tree* and *Forest* are generated by *mktree*, *empty* and *add* are acceptable, but more models satisfy the above two *separate* sort generatedness constraints. For instance, a model with both a junk tree *jt* and a junk forest *jf* fulfills the above declarations (assuming that the interpretations of *mktree* and *add* on *jt* and *jf* in this model are such that $jt = mktree(n, jf)$ for any node n and that $jf = add(jt, jf)$). Hence, one must write instead:

sort *Node*
generated types *Tree* ::= *mktree*(*Node*; *Forest*);
Forest ::= *empty* | *add*(*Tree*; *Forest*)

Here, the mutually recursive datatypes *Tree* and *Forest* are correctly defined simultaneously within the same **generated types** construct, and the resulting semantics is the expected one (without junk values for trees and forests). Note that therefore, the linear visibility rule is *not* applicable *within* a **generated types** construct (to allow such mutually recursive definitions), but that this is the only exception to the linear visibility principle. Only mutually recursive generated datatypes need to be declared together; in simpler cases, it makes no difference to have a sequence of successive generated datatype declarations or just one introducing all the desired datatypes.[6]

3.3 Free Specifications

Free specifications provide initial semantics and avoid the need for explicit negation.

spec NATURAL = **free type** *Nat* ::= *0* | *suc*(*Nat*)

[6] The same explanations apply to free datatypes, introduced in the next subsection.

A **free** datatype declaration corresponds to the so-called '*no junk, no confusion*' principle: there are no other values of sort *Nat* than those denoted by the constructor terms built with *0* and *suc*, and all distinct constructor terms denote different values.

Hence, a **free** datatype declaration such as the one above has *exactly* the same effect as the corresponding **generated** datatype declaration, *together* with axioms stating that *suc* is injective, and that *0* cannot be the successor of a natural number. An alternative to the above '**free type** $Nat ::= 0 \mid suc(Nat)$' is therefore:

$$\begin{array}{l}\textbf{generated type}\ Nat ::= 0 \mid suc(Nat)\\ \forall x, y : Nat \bullet suc(x) = suc(y) \Rightarrow x = y\\ \forall x : Nat \bullet \neg(0 = suc(x))\end{array}$$

Free datatype declarations are particularly convenient for defining enumerated datatypes.

$$\begin{array}{l}\textbf{spec}\ \textsc{Color} =\\ \quad \textbf{free type}\ RGB ::= Red \mid Green \mid Blue\\ \quad \textbf{free type}\ CMYK ::= Cyan \mid Magenta \mid Yellow \mid Black\\ \textbf{end}\end{array}$$

Using '**free**' instead of '**generated**' for defining enumerated datatypes saves the writing of many explicit distinctness assertions (for instance, here, $\neg(Red = Green)$, $\neg(Red = Blue)$, ...).

Free specifications can also be used when the constructors are related by some axioms.

$$\begin{array}{l}\textbf{spec}\ \textsc{Integer} =\\ \quad \textbf{free}\ \{\ \textbf{type}\ Int ::= 0 \mid suc(Int) \mid pre(Int)\\ \qquad \forall x : Int \bullet suc(pre(x)) = x\\ \qquad\qquad \bullet\ pre(suc(x)) = x\ \}\\ \textbf{end}\end{array}$$

When some relations are to be imposed between the constructors (as is the case here for *suc* and *pre* which are inverses of each other), a **free** datatype declaration cannot be used, since the contradiction between the 'no confusion' principle and the axioms imposed on the constructors would lead to an inconsistent specification. Instead, one should impose a '*freeness constraint*' around the datatype declaration followed by the required axioms. A freeness

constraint, expressed by the keyword **free**, can be imposed around any specification.

In the case of the above INTEGER specification, the freeness constraint imposes that the semantics of the specification is the class of all algebras isomorphic to the quotient of the constructor terms by (the minimal congruence induced by) the given axioms. This is exactly the desired semantics. More generally, a freeness constraint around a specification indicates its initial model, which may not exist, of course. It is however well-known that initial models of basic specifications with axioms restricted to Horn clauses (of which equations as in INTEGER are a special case) always exist.[7] Remember also that equality holds minimally in initial models of equational specifications.

Predicates hold minimally in models of free specifications.

spec NATURAL_ORDER =
NATURAL
then **free {** **pred** $\_ < \_ : Nat \times Nat$
$\forall x, y : Nat$
• $0 < suc(x)$
• $x < y \Rightarrow suc(x) < suc(y)$ **}**
end

A freeness constraint imposed around a predicate declaration followed by some defining axioms has the effect that the predicate only holds when this follows from the given axioms, and does not hold otherwise. For instance, in the above example, it is not necessary to explicitly state that '$\neg(0 < 0)$', since this will follow from the imposed freeness constraint. Hence, in such cases a freeness constraint has exactly the same effect as the so-called '*negation as failure*' or '*closed world assumption*' principles in logic programming. More generally, it is often convenient to define a predicate within a freeness constraint, since by doing so, one has to specify the 'positive' cases only.

Operations and predicates may be safely defined by induction on the constructors of a free datatype declaration.

spec NATURAL_ARITHMETIC =
NATURAL_ORDER

[7] Strictly speaking, existence of initial models depends on a further requirement: namely the existence of a ground term for each sort. This ensures that the term-algebra has non-empty carriers and hence is a CASL model.

then **ops** $1 : Nat = suc(0)$;
$\_\_ + \_\_ : Nat \times Nat \to Nat,\ assoc,\ comm,\ unit\ 0$;
$\_\_ * \_\_ : Nat \times Nat \to Nat,\ assoc,\ comm,\ unit\ 1$
$\forall x, y : Nat$
• $x + suc(y) = suc(x + y)$
• $x * 0 = 0$
• $x * suc(y) = (x * y) + x$
end

To define some operation on a free datatype, it is generally recommended to make a case distinction with respect to the various constructors defined. This is illustrated by the above definitions of '+' and '$*$' (although for the '+' operation, the case for the constructor *0* is already taken care of by the attribute '*unit 0*').[8]

More care may be needed when defining operations or predicates on free datatypes when there are axioms relating the constructors.

spec INTEGER_ARITHMETIC =
INTEGER
then **ops** $1 : Int = suc(0)$;
$\_\_ + \_\_ : Int \times Int \to Int,\ assoc,\ comm,\ unit\ 0$;
$\_\_ - \_\_ : Int \times Int \to Int$;
$\_\_ * \_\_ : Int \times Int \to Int,\ assoc,\ comm,\ unit\ 1$
$\forall x, y : Int$
• $x + suc(y) = suc(x + y)$
• $x + pre(y) = pre(x + y)$
• $x - 0 = x$
• $x - suc(y) = pre(x - y)$
• $x - pre(y) = suc(x - y)$
• $x * 0 = 0$
• $x * suc(y) = (x * y) + x$
• $x * pre(y) = (x * y) - x$
end

While a case distinction with respect to the constructors of a free datatype is harmless, this may not be the case for a datatype defined within a freeness constraint, since, due to the axioms relating the constructors to each other, some cases may overlap. This does not mean, however, that one cannot use the case distinction, but just that more attention should be paid than for a free datatype – since one needs to ensure that the definitions lead to the same

[8] In specification libraries, ordinary decimal notation for natural numbers can be provided by use of so-called literal syntax annotations, see Chap. 9.

results for overlapping cases. For instance, in the above example no problem arises. But one should be more careful with the next one, since a negative integer can be of the form $suc(x)$, hence asserting, e.g., $0 \leq suc(x)$, would of course be wrong.

spec INTEGER_ARITHMETIC_ORDER =
INTEGER_ARITHMETIC
then **preds** $\_\_ \leq \_$, $\_\_ \geq \_$, $\_ < \_$, $\_ > \_\_ : Int \times Int$
$\forall x, y : Int$
- $0 \leq 0$
- $\neg(0 \leq pre(0))$
- $0 \leq x \Rightarrow 0 \leq suc(x)$
- $\neg(0 \leq x) \Rightarrow \neg(0 \leq pre(x))$
- $suc(x) \leq y \Leftrightarrow x \leq pre(y)$
- $pre(x) \leq y \Leftrightarrow x \leq suc(y)$
- $x \geq y \Leftrightarrow y \leq x$
- $x < y \Leftrightarrow (x \leq y \wedge \neg(x = y))$
- $x > y \Leftrightarrow y < x$

end

Generic specifications often involve free extensions of (loose) parameters.

spec LIST [**sort** *Elem*] = **free type** $List ::= empty \mid cons(Elem; List)$

The parameter of a generic specification should be loose to cope with the various expected instantiations. On the other hand, it is a frequent situation that the body of the generic specification should have a free, initial interpretation. This is illustrated by the above example, where we want to combine a loose interpretation for the sort *Elem* with a free interpretation for lists. The following example is similar in spirit.

spec SET [**sort** *Elem*] =
free { **type** $Set ::= empty \mid insert(Elem; Set)$
pred $\_is\_in\_\_ : Elem \times Set$
$\forall e, e' : Elem; S : Set$
- $insert(e, insert(e, S)) = insert(e, S)$
- $insert(e, insert(e', S)) = insert(e', insert(e, S))$
- $\neg(e\ is\_in\ empty)$
- $e\ is\_in\ insert(e, S)$
- $e\ is\_in\ insert(e', S)\ if\ e\ is\_in\ S$ **}**

end

As for the LIST example, we want to have a loose interpretation for the sort *Elem* and a free interpretation for sets. Since some axioms are required to hold for the *Set* constructors *empty* and *insert*, we cannot use a free datatype declaration, hence we use a freeness constraint.

Note that since, as already explained, predicates hold minimally in models of free specifications, it would have been enough, in the above example, to define the predicate *is_in* by the sole axiom $e \ is\_in \ insert(e, S)$.[9] However, doing so would have decreased the comprehensibility of the specification and this is the reason why we have preferred a more verbose axiomatization of the predicate *is_in*.

Note also the use of the keyword '*if*' to write an implication in the reverse order:

$$e \ is\_in \ insert(e', S) \ if \ e \ is\_in \ S$$

is equivalent to:

$$e \ is\_in \ S \Rightarrow e \ is\_in \ insert(e', S)$$

The following example specifies the transitive closure of an arbitrary binary relation R on some sort *Elem* (both provided by the parameter).

spec TRANSITIVE_CLOSURE [**sort** *Elem* **pred** $\_\_R\_\_ : Elem \times Elem$] =
free { **pred** $\_R^+\_ : Elem \times Elem$
$\forall x, y, z : Elem$
• $x \ R \ y \Rightarrow x \ R^+ y$
• $x \ R^+ y \wedge y \ R^+ z \Rightarrow x \ R^+ z$ **}**

In the above example, it is crucial that predicates hold minimally in models of free specifications, since this property ensures that what we define as 'R^+' is actually the *smallest* transitive relation including R. Without requiring the freeness constraint, one would allow arbitrary transitive relations containing R (and these undesired relations cannot be eliminated merely by specifying further first-order axioms).

Loose extensions of free specifications can avoid overspecification.

spec NATURAL_WITH_BOUND =
NATURAL_ARITHMETIC
then **op** *max_size* : *Nat*
• $0 < max\_size$
end

[9] If an element e belongs to a set S', then this set S' can always be denoted by a constructor term of the form $insert(e, S)$, due to the axioms constraining the constructor *insert*.

The above example shows another benefit of mixing loose and initial semantics. Assume that at this stage we want to introduce some bound, of sort *Nat*, without fixing its value yet (this value is likely to be fixed later in some refinement, and all that we need for now is the existence of some bound). This is provided by the above specification NATURAL_WITH_BOUND, where we mix an initial interpretation for the sort *Nat* (defined using a free datatype declaration in NATURAL) and a loose interpretation for the constant *max_size*. Each model of NATURAL_WITH_BOUND will provide a fixed interpretation of the constant *max_size*, and all these models are captured by NATURAL_WITH_BOUND, which is in this sense loose. Using such loose extensions is in general appropriate to avoid unnecessary overspecification.

spec SET_CHOOSE [**sort** *Elem*] =
SET [**sort** *Elem*]
then **op** *choose* : $Set \to Elem$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$
end

This example shows again the benefit of mixing initial and loose semantics. Here, we want to extend sets, defined using a free constraint in SET, by a loosely specified operation *choose*.[10] At this stage, the only property required for *choose* is to provide some element belonging to the set to which it is applied, and we do not want to specify more precisely which specific element is to be chosen. Note that each model of SET_CHOOSE will provide a function implementing some specific choice strategy, and that since all these interpretations of *choose* have to be *functions*, they are necessarily 'deterministic' (e.g., applied twice to the same set argument, they return the same result).

Datatypes with observer operations or predicates can be specified as generated instead of free.

spec SET_GENERATED [**sort** *Elem*] =
generated type $Set ::= empty \mid insert(Elem;\ Set)$
pred $\_\_is\_in\_\_ : Elem \times Set$
$\forall e, e' : Elem;\ S, S' : Set$
$\bullet\ \neg(e\ is\_in\ empty)$
$\bullet\ e\ is\_in\ insert(e', S) \Leftrightarrow (e = e' \vee e\ is\_in\ S)$
$\bullet\ S = S' \Leftrightarrow (\forall x : Elem \bullet x\ is\_in\ S \Leftrightarrow x\ is\_in\ S')$
end

[10] For the purpose of this example, we disregard the fact that *choose* should be undefined on the empty set, and we just leave this case unspecified. Partial functions are discussed in Chap. 4.

The above specification is an alternative to the specification SET (see p. 40). Both SET and SET_GENERATED define exactly the same class of models. The former specification relies on a freeness constraint, while SET_GENERATED relies on the observer *is_in* to specify when two sets are equal. Indeed, the last axiom of SET_GENERATED expresses directly that two sets having exactly the same elements are equal values. This axiom, together with the first two axioms defining *is_in*, will entail as well the expected properties on the constructor *insert* (see GENERATED_SET p. 35). Note also that since, in SET_GENERATED, the predicate *is_in* is not defined within a freeness constraint, we specify when it holds using '$\Leftrightarrow$' rather than a one-way implication.

While a freeness constraint may be unavoidable to define a predicate, as illustrated by TRANSITIVE_CLOSURE, the choice between relying on a freeness constraint to define a datatype such as *Set*, or using instead a generated datatype declaration together with some observers to unambiguously determine the values of interest, is largely a matter of convenience. One may argue that SET is more suitable for prototyping tools based on term rewriting, while SET_GENERATED is more suitable for theorem-proving tools.

The ***%def*** *annotation is useful to indicate that some operations or predicates are uniquely defined.*

spec SET_UNION [**sort** *Elem*] =
SET [**sort** *Elem*]
then %def
ops $\_\cup\_ : Set \times Set \to Set$, *assoc*, *comm*, *idem*, *unit empty*;
$remove : Elem \times Set \to Set$
$\forall e, e' : Elem;\ S, S' : Set$
- $S \cup insert(e', S') = insert(e', S \cup S')$
- $remove(e, empty) = empty$
- $remove(e, insert(e, S)) = remove(e, S)$
- $remove(e, insert(e', S)) = insert(e', remove(e, S))$ *if* $\neg(e = e')$

end

The annotation **%def** expresses that SET_UNION is a *definitional extension* of SET, i.e., that each model of SET can be *uniquely* extended to a model of SET_UNION, which means that the operations introduced in SET_UNION are uniquely defined. As with the **%implies** annotation, the **%def** annotation has no impact on the semantics, but a corresponding proof obligation can be generated, to be discharged by theorem proving tools. The **%def** annotation is especially useful to stress that the specifier's intention is to impose a unique interpretation of what is defined within the scope of this annotation (once an interpretation for the part which is extended has been chosen).

Operations can be defined by axioms involving observer operations, instead of inductively on constructors.

spec SET_UNION_1 [**sort** *Elem*] =
SET_GENERATED [**sort** *Elem*]
then %def
ops $\_\_ \cup \_\_ : Set \times Set \to Set$, *assoc*, *comm*, *idem*, *unit empty*;
$remove : Elem \times Set \to Set$
$\forall e, e' : Elem;\ S, S' : Set$
• $e\ is\_in\ (S \cup S') \Leftrightarrow (e\ is\_in\ S \vee e\ is\_in\ S')$
• $e\ is\_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is\_in\ S)$
end

The specification SET_UNION_1 is an alternative to SET_UNION and defines exactly the same model class. While an inductive definition style was chosen for the operations '$\cup$' and *remove* in SET_UNION, in SET_UNION_1 these operations are defined 'implicitly' by characterizing their results through the observer *is_in*. Note that this 'observer' style does not prevent us providing a unique definition of both operations, as claimed by the **%def** annotation.

Similarly to the discussion on the respective merits of SET and of SET_GENERATED, the choice between an inductive definition style and an 'observer' definition style is partly a matter of taste. One may argue that the 'observer' definition style is more abstract in the sense that there is no hint to any algorithmic computation of the so-defined operations, while the inductive definition style mimics a recursive definition in a functional programming language. Again, the inductive definition style may be more suitable for prototyping tools based on term rewriting, while the 'observer' definition style may be more suitable for theorem-proving tools.

Sorts declared in free specifications are not necessarily generated by their constructors.

spec UNNATURAL =
free { type $UnNat ::= 0 \mid suc(UnNat)$
op $\_\_ + \_\_ : UnNat \times UnNat \to UnNat$,
assoc, *comm*, *unit 0*
$\forall x, y : UnNat \bullet x + suc(y) = suc(x + y)$
$\forall x : UnNat \bullet \exists y : UnNat \bullet x + y = 0$ **}**
end

This rather peculiar example illustrates the fact that a sort defined within a freeness constraint need not be generated by its constructors. In UNNATURAL, the specification enclosed within the **free { ... }** construct specifies

Abelian groups with one generator $suc(0)$, and the integers are the free such Abelian group. Hence, the (unique up to isomorphism) model of UnNatural corresponds to integers, and not to natural numbers as one may expect – just consider the last axiom. This example points out why in general datatypes defined using freeness constraints can be more difficult to understand than datatypes defined using generatedness constraints. However, the reader should be aware that the specification UnNatural uses a proper first-order formula with an existential quantifier in the axioms. The specification UnNatural is provided here for explanatory purposes only, and clearly the writing of similar specifications should be discouraged. When only Horn clauses are used as axioms in a freeness constraint, then the datatype will indeed be generated by its constructors.