5

Subsorting

Subsorts and supersorts are often useful in CASL specifications.

Many examples naturally involve subsorts and supersorts. CASL provides means for the declaration of a sort as a subsort of another one when the values of the subsort are regarded a special case of those in the other sort. The aim of this chapter is to discuss and illustrate how to handle subsorts and supersorts in CASL specifications.

5.1 Subsort Declarations and Definitions

Subsort declarations directly express relationships between carrier sets.

spec GENERIC_MONOID_1 [**sort** *Elem*] =
sorts *Elem* < *Monoid*
ops *1* : *Monoid*;
_ * _ : *Monoid* × *Monoid* → *Monoid*, *assoc*, *unit 1*
end

The above example declares the sort *Elem* to be a subsort of *Monoid*, or, symmetrically, the sort *Monoid* to be a supersort of *Elem*. Hence the specification GENERIC_MONOID_1 is quite similar to the specification GENERIC_MONOID given in Chap. 3, p. 30, the only difference being the use of a subsorting relation between *Elem* and *Monoid* instead of an explicit *inj* operation to embed values of sort *Elem* into values of sort *Monoid*.

M. Bidoit and P.D. Mosses: CASL User Manual, LNCS 2900, pp. 57–66, 2004.

In contrast to most other algebraic specification languages providing subsorting facilities, subsorts in CASL are interpreted by arbitrary *embeddings* between the corresponding carrier sets. In the above example, the subsort declaration $Elem < Monoid$ induces an implicit (unnamed) embedding from the carrier of the sort *Elem* into the carrier of the sort *Monoid*. Thus the main difference between GENERIC_MONOID and GENERIC_MONOID_1 is that the embedding is explicit and named *inj* in GENERIC_MONOID while it is implicit in GENERIC_MONOID_1.

Note that interpreting subsorting relations by embeddings rather than inclusions does not exclude models where the (carrier of the) subsort happens to be a subset of (the carrier of) the supersort, and the embedding a proper inclusion. Embeddings are just slightly more general than inclusions, and technically not much more complex.

> *Operations declared on a sort are automatically inherited by its subsorts.*

```
spec  VEHICLE =
      NATURAL
then  sorts Car, Bicycle < Vehicle
      ops   max_speed       : Vehicle → Nat;
            weight          : Vehicle → Nat;
            engine_capacity : Car → Nat
end
```

The above example introduces three sorts, *Car*, *Bicycle* and *Vehicle*, and declares both *Car* and *Bicycle* to be subsorts of *Vehicle*. A subsort declaration entails that any term of a subsort is also a term of the supersort, so here, any term of sort *Car* is also a term of sort *Vehicle*, and we can apply the operations *max_speed* and *weight* to it (and similarly for a term of sort *Bicycle*).

In other words, with the single declaration $max\_speed : Vehicle \to Nat$, we get the effect of having declared also two other operations, $max\_speed : Car \to Nat$ and $max\_speed : Bicycle \to Nat$.[1]

Obviously, operations that are only meaningful for some subsort should be defined at the appropriate level. This is the case here for the operation *engine_capacity*, which is only relevant for cars, and therefore defined with the appropriate profile exploiting the subsort *Car*.

[1] Strictly speaking, there is just one *max_speed* operation in the signature of VEHICLE. The difference between the kind of inheritance described here and operations actually declared on subsorts becomes important when writing symbol maps, see Chap. 7.

Inheritance applies also for subsorts that are declared afterwards.

spec MORE_VEHICLE = VEHICLE **then sorts** *Boat* < *Vehicle*

The order in which a subsort and an operation on the supersort are declared is irrelevant. In MORE_VEHICLE, we introduce a further subsort *Boat* of *Vehicle*, and as a consequence, we again get the effect of having both *max_speed* and *weight* available for boats, as was already the case for cars and bikes.

Subsort membership can be checked or asserted.

spec SPEED_REGULATION =
VEHICLE
then ops *speed_limit* : *Vehicle* → *Nat*;
car_speed_limit, *bike_speed_limit* : *Nat*
∀*v* : *Vehicle*
• $v \in Car \Rightarrow speed\_limit(v) = car\_speed\_limit$
• $v \in Bicycle \Rightarrow speed\_limit(v) = bike\_speed\_limit$
end

A subsort membership assertion, written '$t \in s$', where t is a term and s is a sort, is a special kind of atomic formula: it holds if and only if the value of the term t is the embedding of some value of sort s. For instance, in the above example, $v \in Car$ holds if and only if v denotes a vehicle which is the embedding of a car value. Note that '$\in$' is input as '`in`', but displayed as '$\in$'.

Datatype declarations can involve subsort declarations.

The sequence of declarations:

sorts *Car*, *Bicycle*, *Boat*
type *Vehicle* ::= *sort Car* | *sort Bicycle* | *sort Boat*

is equivalent to the declaration **sorts** *Car*, *Bicycle*, *Boat* < *Vehicle*. There may be some values of sort *Vehicle* which are not the embedding of any value of sort *Car*, *Bicycle*, or *Boat*. Intuitively, the above datatype declaration just means that *Vehicle* 'contains' the *union* (which may not be disjoint) of *Car*, *Bicycle* and *Boat*. Note that the subsorts used in the datatype declaration must already be declared beforehand.

The sequence of declarations:

sorts *Car*, *Bicycle*, *Boat*
generated type *Vehicle* ::= *sort Car* | *sort Bicycle* | *sort Boat*

is more restrictive, since the generatedness constraint implies that any value of the supersort *Vehicle* must be the embedding of some value of the declared subsorts *Car*, *Bicycle* and *Boat*. Intuitively, the above datatype declaration means that *Vehicle* 'is exactly' the union (which again may not be disjoint) of *Car*, *Bicycle* and *Boat*. In particular, this declaration prevents subsequent introduction of further subsorts (unless the values of the new subsorts are intended to correspond to some values of the already declared subsorts). For instance, if we were now to extend the above specification with **sorts** $Plane < Vehicle$, all values of sort *Plane* would have to correspond to *Car*, *Bicycle* or *Boat* values (which is presumably not what we were intending).

The sequence of declarations:

sorts *Car*, *Bicycle*, *Boat*
free type *Vehicle* ::= *sort Car* | *sort Bicycle* | *sort Boat*

entails the same generatedness constraint as in the previous example, and, moreover, the freeness constraint requires that there is no 'common' value in the subsorts of *Vehicle*. Intuitively, the above declaration means that *Vehicle* 'is exactly' the *disjoint* union of *Car*, *Bicycle* and *Boat*. This means in particular that the introduction of a further common subsort of both *Car* and *Boat* (say, **sorts** $Amphibious < Car, Boat$) is impossible.

Subsorts may also arise as classifications of previously specified values, and their values can be explicitly defined.

spec NATURAL_SUBSORTS =
NATURAL_ARITHMETIC
then **pred** $even : Nat$
• $even(0)$
• $\neg\ even(1)$
$\forall n : Nat \bullet even(suc(suc(n))) \Leftrightarrow even(n)$
sort $Even = \{x : Nat \bullet even(x)\}$
sort $Prime = \{x : Nat \bullet 1 < x \ \wedge$
$\forall y, z : Nat \bullet x = y * z \Rightarrow y = 1 \vee z = 1\}$
end

The subsort definition **sort** $Even = \{x : Nat \bullet even(x)\}$ is equivalent to the declaration of a subsort *Even* of *Nat* (i.e., **sorts** $Even < Nat$) together with the assertion $\forall x : Nat \bullet x \in Even \Leftrightarrow even(x)$.

The main advantage of defining the subsort *Even* in addition to the predicate *even* is that we may then use the subsort when declaring operations (e.g., **op** $times2 : Nat \to Even$) and variables.

The subsort definition for *Prime* above illustrates that it is not always necessary to introduce and define an explicit predicate characterizing the values of the subsort: the formula used in a subsort definition is not restricted to predicate applications. In fact whenever a (unary) predicate p on a sort s could be defined by **pred** $p(x : s) \Leftrightarrow f$ for some formula f, we may instead define **sort** $P = \{x : s \bullet f\}$, and use sort membership assertions $t \in P$ instead of predicate applications $p(t)$, avoiding the introduction of the predicate p altogether.

The following example is a further illustration of subsort definitions. We declare a subsort *Pos* of *Nat* and ensure that values of *Pos* correspond to non-zero values of *Nat*. (Several alternative ways of specifying the sort *Pos* will be considered later in this section.)

spec POSITIVE =
NATURAL_PARTIAL_PRE
then **sort** $Pos = \{x : Nat \bullet \neg(x = 0)\}$

5.2 Subsorts and Overloading

It may be useful to redeclare previously defined operations, using the new subsorts introduced.

spec POSITIVE_ARITHMETIC =
POSITIVE
then **ops** $1 : Pos$;
$suc : Nat \to Pos$;
$\_\_ + \_, \_\_ * \_ : Pos \times Pos \to Pos$;
$\_\_ + \_ : Pos \times Nat \to Pos$;
$\_\_ + \_ : Nat \times Pos \to Pos$
end

Since, in POSITIVE, we have defined *Pos* as a subsort of *Nat*, all operations defined on natural numbers, like *suc*, '+' and '$*$' (see NATURAL_ARITHMETIC in Chap. 3, p. 38, which is extended into NATURAL_PARTIAL_PRE in Chap. 4, p. 52), are automatically inherited by *Pos* and can be applied to terms of sort *Pos*. However, according to their declarations, these operations, when applied to terms of sort *Pos*, yield results of sort *Nat*. To indicate that these results always correspond to values in the subsort *Pos*, it is necessary to explicitly *overload* these operations by similar ones with the appropriate profiles. This is

the aim of the first three lines of operation declarations in the above example. The last two operation declarations further overload '+' to specify that '+' also yields a result of sort *Pos* as soon as one of its arguments is a term of sort *Pos*.

It is quite important to understand that the above overloading declarations are enough to achieve the desired effect, and that *no axioms are necessary.* The fundamental rule is that, in models of CASL specifications with subsorting, embedding and overloading have to be compatible: embeddings commute with overloaded operations. This can be rephrased into the following intuitive statement: *If terms look the same, then they have the same value in the same sort.* Thus, in our example, the value of '*1 + 1*' is the same in *Nat* whatever the combination of the overloaded constant '*1*' and operation '+' is chosen, and there is no need for any axiom to ensure this, since this is implicit in the semantics of subsorting.

5.3 Subsorts and Partiality

A subsort may correspond to the definition domain of a partial function.

spec POSITIVE_PRE =
POSITIVE_ARITHMETIC
then **op** *pre* : *Pos* → *Nat*

Since we have introduced the subsort *Pos* of non-zero natural numbers, it makes sense to overload the partial *pre* operation on *Nat* by a *total* one on *Pos*, as illustrated above, to emphasize the fact that indeed *pre* is a total operation on its definition domain. Note again that no further axiom is necessary, and that the semantics of subsorting will ensure that both the partial and total *pre* operations will give the same value when applied to the same non-zero value.[2]

Using subsorts may avoid the need for partial functions.

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

[2] This should not be confused with the same name, same meaning principle, which does not apply here: the total *pre* and the partial one have different profiles, and hence are just overloaded.

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

end

It is indeed tempting to exploit subsorting to avoid the declaration of partial functions, as illustrated by the above NATURAL_POSITIVE_ARITHMETIC specification, which is an alternative to POSITIVE_PRE and avoids the introduction of the partial predecessor operation. Note that in the above example, we have fully used the facilities for defining free datatypes with subsorts, and in particular non-linear visibility for the declared sorts, which allows us to refer to the subsort *Pos* in the first line before defining it in the second one.

Avoiding the introduction of the partial predecessor operation has some drawbacks, since some previously well-formed terms (with defined values) have now become ill-formed, e.g., $pre(pre(suc(1)))$, where $pre(suc(1))$ is a (well-formed) term of sort *Nat* but *pre* expects an argument of sort *Pos*. (The fact that $pre(suc(1)) = 1$ is a consequence of the specified axioms, and that *1* is of sort *Pos*, does not of course entail that $pre(suc(1))$ is of sort *Pos* too, since axioms are disregarded when checking for well-formedness.) See below for possible workarounds using explicit casts. Moreover, it is not always possible, or easy, to avoid the declaration of partial operations by using appropriate subsorts—just consider, e.g., subtraction on natural numbers.

As a last remark on this issue, the reader should be aware of the fact that, while overloading a partial operation on a supersort (say, *pre* on *Nat*) with a total one on a subsort (*pre* on *Pos*) is fine, overloading a total operation on a *supersort* with a partial one on a *subsort* forces the partial operation to be total, and hence, the latter would be better declared as total too.[3]

Casting a term from a supersort to a subsort is explicit and the value of the cast may be undefined.

[3] Overloading a total *cons* on *List* with a partial *cons* on the subsort *OrderedList* would either lead to a total *cons* operation on *OrderedList*, or to an inconsistent specification, depending on how the definition domain of the partial *cons* is specified.

In CASL, a term of a subsort can always be considered as a term of any supersort, and embeddings are implicit. On the contrary, casting a term from a supersort to a subsort is explicit, and since casting is essentially a partial operation, the resulting casted term may not denote any value. Casting a term t to a sort s is written t *as* s. Consider the term $pre(\ pre(suc(1))$ *as Pos* $)$ which is well-formed in the context of the NATURAL_POSITIVE_ARITHMETIC specification. This term does indeed denote a value, but the value is not a positive natural number, so the value of the term $pre(\ pre(suc(1))$ *as Pos* $)$ *as Pos* is undefined.

Note that *def* $(t$ *as* $s)$ is equivalent to $t \in s$, for a well-formed term t of a supersort of s.

Supersorts may be useful when generalizing previously specified sorts.

spec INTEGER_ARITHMETIC_1 =
NATURAL_POSITIVE_ARITHMETIC
then **free type** $Int ::= sort\ Nat \mid -\_\_(Pos)$
ops $\_\_ + \_\_ : Int \times Int \to Int,\ assoc,\ comm,\ unit\ 0;$
$\_\_ - \_\_ : Int \times Int \to Int;$
$\_\_ * \_\_ : Int \times Int \to Int,\ assoc,\ comm,\ unit\ 1;$
$abs : Int \to Nat$
$\forall x : Int;\ n : Nat;\ p, q : Pos$
- $suc(n) + (-1) = n$
- $suc(n) + (-suc(q)) = n + (-q)$
- $(-p) + (-q) = -(p + q)$
- $x - 0 = x$
- $x - p = x + (-p)$
- $x - (-q) = x + q$
- $0 * (-q) = 0$
- $p * (-q) = -(p * q)$
- $(-p) * (-q) = p * q$
- $abs(n) = n$
- $abs(-p) = p$

end

The specification INTEGER_ARITHMETIC_1 extends NATURAL_POSITIVE_ARITHMETIC and defines the sort *Int* as a supersort of the sort *Nat*. As a consequence, terms which have two parses in sort *Int*, depending whether the embedding from *Nat* to *Int* is applied to the arguments or to the result of overloaded operations, are required by the semantics of subsorting to have the same value for both parses, and they can therefore be used without explicit disambiguation.

The situation would be quite different if one would be using a combination of NATURAL_ARITHMETIC and INTEGER_ARITHMETIC (see Chap. 3), say by extending both in a structured specification (see the next chapter for more details on structured specifications). In such a combination, a term such as $suc(0)$ would have two parses, one in sort *Nat* and one in sort *Int*; in the absence of any subsort declaration relating *Nat* and *Int*, and hence of any implicit embedding, this term would then be ambiguous, and would require explicit disambiguation to become a well-formed term.

Supersorts may also be used for extending the intended values by new values representing errors or exceptions.

spec SET_ERROR_CHOOSE [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **sorts** $Elem < ElemError$
op $choose : Set \to ElemError$
pred $\_is\_in\_ : ElemError \times Set$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S) \in Elem \wedge choose(S)\ is\_in\ S$
end

The above specification SET_ERROR_CHOOSE is another variant of the various specifications of sets equipped with a partial choose function given in Chap. 4. This variant avoids the declaration of a partial function *choose* by using instead a supersort of *Elem*, namely *ElemError*, as the target sort of *choose*. The idea here is that values of *ElemError* which are not (embeddings of) values of *Elem* represent errors, e.g., the application of *choose* to the empty set. Note that to obtain the desired effect, it is necessary to explicitly state that $choose(S) \in Elem$ when S is not the empty set; moreover, to make the term $choose(S)\ is\_in\ S$ well-formed, we have to explicitly overload the predicate $\_is\_in\_ : Elem \times Set$ provided by GENERATED_SET by a predicate $\_is\_in\_ : ElemError \times Set$ as shown above. This example demonstrates that avoiding partial functions by the use of 'error supersorts' is not as innocuous as it may seem, since in general one would need to enlarge the signatures considerably by adding all required overloadings.

spec SET_ERROR_CHOOSE_1 [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **sorts** $Elem < ElemError$
op $choose : Set \to ElemError$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow (choose(S)\ as\ Elem)\ is\_in\ S$
end

The specification SET_ERROR_CHOOSE_1 above is a last attempt to avoid the use of partial functions; again, we introduce a supersort *ElemError* as

in SET_ERROR_CHOOSE, but to avoid the need for overloading the predicate *is_in*, we explicitly *cast* the term $choose(S)$ in $(choose(S)\ as\ Elem)$ *is_in* S. Note that when, for some value of S, $(choose(S)\ as\ Elem)$ *is_in* S holds, this implies that $choose(S)$ *as Elem* is defined, and hence that $choose(S) \in Elem$ holds as well. This last version may seem preferable to the previous one. However, one should be aware that here, despite our attempt to avoid the use of partial functions, we rely on explicit casts, hence on terms that may not denote values: partiality has not been eliminated, the partial functions have merely been factorized as compositions of total functions with casting.