4

Partial Functions

Partial functions arise naturally.

Partial functions arise in a number of situations. CASL provides means for the declaration of partial functions, the specification of their domains of definition, and more generally the specification of system properties involving partial functions. The aim of this chapter is to discuss and illustrate how to handle partial functions in CASL specifications.

4.1 Declaring Partial Functions

Partial functions are declared differently from total functions.

```
spec  SET_PARTIAL_CHOOSE [sort Elem] =
      GENERATED_SET [sort Elem]
then  op choose : Set →? Elem
end
```

The *choose* function on sets is naturally a partial function, expected to be undefined on the empty set. In CASL, a partial function is declared similarly to a total one, *but* for the question mark '?' following the arrow in the profile. It is therefore quite easy to distinguish the functions declared to be total from the ones declared to be partial.

A function declared to be partial may happen to be total in some of the models of the specification. For instance, the above specification SET_PARTIAL_CHOOSE does not exclude models where the function symbol *choose* is interpreted by a total function, defined on all set values. Axioms can be used to specify the domain of definition of a partial function, and how to do this is detailed later in this chapter.

Terms containing partial functions may be undefined, i.e., they may fail to denote any value.

For instance, the (value of the) term $\mathit{choose}(\mathit{empty})$ may be undefined.[1] This is more natural than insisting that $\mathit{choose}(\mathit{empty})$ has to denote some arbitrary but fixed element of *Elem*.

Note that variables range only over defined values, and therefore a variable always denotes a value, in contrast to terms containing partial functions.

Functions, even total ones, propagate undefinedness.

If the term $\mathit{choose}(S)$ is undefined for some value of S, then the term $\mathit{insert}(\mathit{choose}(S), S')$ is undefined as well for this value of S, although *insert* is a total function.

Predicates do not hold on undefined arguments.

Casl is based on classical two-valued logic. A predicate symbol is interpreted by a relation, and when the value of some argument term is undefined, the application of a predicate to this term does not hold. For instance, if the term $\mathit{choose}(S)$ is undefined, then the atomic formula $\mathit{choose}(S)\ \mathit{is\_in}\ S$ does not hold.

Equations hold when both terms are undefined.

In Casl, equations are by default *strong*, which means that they hold not only when both sides denote equal values, but also when both sides are simultaneously undefined. For instance, let us consider the equation:

$$\mathit{insert}(\mathit{choose}(S), \mathit{insert}(\mathit{choose}(S), \mathit{empty})) = \mathit{insert}(\mathit{choose}(S), \mathit{empty})$$

[1] Note that the term $\mathit{choose}(\mathit{empty})$ is well-formed and therefore is a 'correct term'. It is *its value* which may be undefined. To avoid unnecessary pedantry, in the following we will simply write that a term is undefined to mean that its value is so. Obviously, a term with variables may be defined for some values of the variables and undefined for other values.

Either *choose*(S) is defined and then both sides are defined and denote equal values due to the axioms on *insert*, or *choose*(S) is undefined and then both sides are undefined, and the strong equation 'holds trivially'.

CASL provides also so-called *existential equations*, explained at the end of this chapter.

Special care is needed in specifications involving partial functions.

Partial functions are intrinsically more difficult to understand and specify than total ones. This is why special care is needed when writing the axioms of specifications involving partial functions. The point is that an axiom may imply the definedness of terms containing partial functions, and as a consequence that these functions are total, which may not be what the specifier intended. Here are three typical cases:

- Asserting *choose*(S) *is_in* S as an axiom implies that *choose*(S) is defined, for any S. The point here is that since predicates applied to an undefined term do not hold, in any model satisfying *choose*(S) *is_in* S, the function *choose* must be total (i.e., always defined).
- Asserting *remove*(*choose*(S), *insert*(*choose*(S), *empty*)) = *empty* as an axiom implies that *choose*(S) is defined for any S, since the term *empty* is always defined. To understand this, assume that *choose* is undefined for some set value of S; then the above equation cannot hold for this value, since the undefinedness of *choose*(S) implies the undefinedness of *remove*(*choose*(S), *insert*(*choose*(S), *empty*)), giving a contradiction with the definedness of *empty*. Hence, an equation between a term involving a partial function *PF* and a term involving total functions only may imply that the partial function *PF* is always defined.
- Asserting *insert*(*choose*$(S), S) = S$ as an axiom implies that *choose*(S) is defined for any S, since a variable always denotes a defined value. This case is indeed similar to the previous one, the only difference being that now the right-hand side of the equation is a variable (instead of a term involving total functions only).

Moreover, the 'same name, same thing' principle has a subtle side-effect regarding partial operations: if an operation is declared both as a total operation and as a partial operation with the same profile (i.e., the same argument sorts and the same result sort) then it is interpreted as a total operation in all models of the specification.

4.2 Specifying Domains of Definition

> *The definedness of a term can be checked or asserted.*

spec $\textsc{Set\_Partial\_Choose\_1}$ [**sort** *Elem*] =
$\textsc{Set\_Partial\_Choose}$ [**sort** *Elem*]
then • $\neg$ *def choose*(*empty*)
$\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$
end

A definedness assertion, written '*def t*', where t is a term, is a special kind of atomic formula: it holds if and only if the value of the term t is defined. For instance, in the above example, $\neg$ *def choose*(*empty*) explicitly asserts that *choose* is undefined when applied to *empty*. Note that this axiom does not say anything about the definedness of *choose* applied to values other than *empty*, which means that *choose* may well be undefined on those values too. The second axiom of the above example asserts *choose*(S) *is_in* S under the condition *def choose*(S), to avoid undesired definedness induced by axioms, as explained in the previous section.

Note that if the two axioms of the above example were to be replaced by:

$$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$$

then we could conclude that *choose*(S) is defined when S is not equal to *empty*, but nothing about the undefinedness of *choose*(*empty*).

> *The domains of definition of partial functions can be specified exactly.*

spec $\textsc{Set\_Partial\_Choose\_2}$ [**sort** *Elem*] =
$\textsc{Set\_Partial\_Choose}$ [**sort** *Elem*]
then $\forall S : Set \bullet def\ choose(S) \Leftrightarrow \neg(S = empty)$
$\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$
end

In the above example, the domain of definition of the partial function *choose* is exactly specified by the axiom *def choose*(S) $\Leftrightarrow \neg(S = empty)$.

Loosely specified domains of definition may be useful.

```
spec  NATURAL_WITH_BOUND_AND_ADDITION =
      NATURAL_WITH_BOUND
then  op __+?__ : Nat × Nat →? Nat
      ∀x, y : Nat
       • def(x+?y) if x + y < max_size
      %{ x + y < max_size implies both
           x < max_size and y < max_size }%
       • def(x+?y) ⇒ x+?y = x + y
end
```

In some cases, it is useful to loosely specify the domain of definition of a partial function, as illustrated in the above example for '+?', which is required to be defined for all arguments x and y such that $x + y < max\_size$, but may well be defined on larger natural numbers as well. The point in loose specifications of definition domains is to avoid unnecessary constraints on the models of the specification. For instance, the above example does not exclude a model where '+?' is interpreted by a total function (which would then coincide with '+').[2]

Indeed, in some cases, specifying exactly domains of definition can be considered as overspecification. In most specifications, however, one would expect an exact specification of domains of definition, even for otherwise loosely specified functions (see, e.g., *choose* in SET_PARTIAL_CHOOSE_2).

Domains of definition can be specified more or less explicitly.

```
spec  SET_PARTIAL_CHOOSE_3 [sort Elem] =
      SET_PARTIAL_CHOOSE [sort Elem]
then   • ¬ def choose(empty)
      ∀S : Set • ¬(S = empty) ⇒ choose(S) is_in S
end
```

SET_PARTIAL_CHOOSE_3 specifies exactly the domain of definition of *choose*, but does this too implicitly, since some reasoning is needed to conclude that the above specification entails $def\ choose(S) \Leftrightarrow \neg(S = empty)$.

[2] In this example, it is essential to choose a new name '+?' for our partial addition operation. Otherwise, since '+' is (rightly) declared as a total operation in NATURAL_WITH_BOUND, the declaration **op** $\_\_ + \_\_ : Nat \times Nat \rightarrow? Nat$ would be useless: the same name, same thing principle would lead to models with just one, total, addition operation.

To improve the clarity of specifications, it is in general advisable to specify definition domains as explicitly as possible, and SET_PARTIAL_CHOOSE_2 is somehow easier to understand than SET_PARTIAL_CHOOSE_3 (both specifications define the same class of models).

```
spec  NATURAL_PARTIAL_PRE =
      NATURAL_ARITHMETIC
then  op pre : Nat →? Nat
       • ¬ def pre(0)
      ∀x : Nat • pre(suc(x)) = x
end
```

In the above example, one can consider that the domain of definition of *pre* is (exactly) specified in an explicit enough way, since the first axiom states exactly that *pre*(*0*) is undefined while the second one implies that *pre* is defined for all natural numbers of the form $suc(x)$.

```
spec  NATURAL_PARTIAL_SUBTRACTION_1 =
      NATURAL_PARTIAL_PRE
then  op __ − __ : Nat × Nat →? Nat
      ∀x, y : Nat
       • x − 0 = x
       • x − suc(y) = pre(x − y)
end
```

The above specification is perfect from a mathematical point of view, but is clearly not explicit enough, since there is no easy way to infer when $x - y$ is defined. From a methodological point of view, the following alternative version is much better.

```
spec  NATURAL_PARTIAL_SUBTRACTION =
      NATURAL_PARTIAL_PRE
then  op __ − __ : Nat × Nat →? Nat
      ∀x, y : Nat
       • def(x − y) ⇔ (y < x ∨ y = x)
       • x − 0 = x
       • x − suc(y) = pre(x − y)
end
```

The above examples clearly demonstrate why the explicit specification of definition domains is generally advisable from a methodological point of view. However, they also indicate that this recommendation should not be applied in too strict a way, and that deciding whether a specification is explicit enough or not is to some extent a matter of taste.

Partial functions are minimally defined by default in free specifications.

```
spec  LIST_SELECTORS_1 [sort Elem] =
      LIST [sort Elem]
then  free { ops head : List →? Elem;
                 tail  : List →? List
             ∀e : Elem; L : List
              • head(cons(e, L)) = e
              • tail(cons(e, L)) = L }
end
```

In the above example, the given axioms imply that *head* and *tail* are defined on lists of the form *cons*(*e*, *L*). The freeness constraint requires that these functions are minimally defined. Since the terms *head*(*empty*) and *tail*(*empty*) are not equated to any other term, the freeness constraint implies that these terms are undefined, and hence that the functions *head* and *tail* are undefined on *empty*. The situation here is similar to the fact that predicates hold minimally in models of free specifications (see Chap. 3, p. 38).

```
spec  LIST_SELECTORS_2 [sort Elem] =
      LIST [sort Elem]
then  ops head : List →? Elem;
          tail : List →? List
      ∀e : Elem; L : List
       • ¬ def head(empty)
       • ¬ def tail(empty)
       • head(cons(e, L)) = e
       • tail(cons(e, L)) = L
end
```

The above specification LIST_SELECTORS_2 is an alternative to LIST_SELECTORS_1; both specifications define exactly the same class of models. However, LIST_SELECTORS_2 is clearly easier to understand and can be considered as technically simpler, since it involves no freeness constraint.

Operations like *head* and *tail* are usually called *selectors*, and CASL provides abbreviations to specify selectors in a very concise way, as we see next.

4.3 Partial Selectors and Constructors

Selectors can be specified concisely in datatype declarations, and are usually partial.

spec LIST_SELECTORS [**sort** *Elem*] =
  free type *List* ::= *empty* | *cons*(*head* :? *Elem*; *tail* :? *List*)

The above free datatype declaration introduces, in addition to the constructors *empty* and *cons*, two partial selectors *head* and *tail* yielding the respective arguments of the constructor *cons*. Hence, this free datatype declaration with selectors has exactly the same effect as the ordinary free datatype declaration **free type** *List* ::= *empty* | *cons*(*Elem*; *List*), together with the operation declarations and axioms of LIST_SELECTORS_2 (i.e., LIST_SELECTORS and LIST_SELECTORS_2 define exactly the same class of models). The following example is similar in spirit.

spec NATURAL_SUC_PRE = **free type** *Nat* ::= *0* | *suc*(*pre* :? *Nat*)

Selectors are usually total when there is only one constructor.

spec PAIR_1 [**sorts** *Elem1*, *Elem2*] =
  free type *Pair* ::= *pair*(*first* : *Elem1*; *second* : *Elem2*)

While selectors are usually partial operations when there is more than one alternative in the corresponding datatype declaration, they can be total, and this is generally the case when there is only one constructor, as in the above example. The free datatype declaration entails in particular axioms asserting that *first* and *second* yield the respective arguments of the constructor *pair* (i.e., $first(pair(e1, e2)) = e1$ and $second(pair(e1, e2)) = e2$).

Constructors may be partial.

spec PART_CONTAINER [**sort** *Elem*] =
  generated type
    P_Container ::= *empty* | *insert*(*Elem*; *P_Container*)?
  pred *addable* : $Elem \times P\_Container$
  vars e, e' : *Elem*; *C* : *P_Container*
  • *def* $insert(e, C) \Leftrightarrow addable(e, C)$

pred $\_\_is\_in\_\_ : Elem \times P\_Container$
- $\neg(e\ is\_in\ empty)$
- $(e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C))\ if\ addable(e', C)$

end

The intention in the above example is to define a reusable specification of partial containers. The *insert* constructor is specified as a partial operation, defined if some condition on both the element e to be added and the container C to which the element is to be added holds. This condition is abstracted here in a predicate *addable*, so far left unspecified. Later on, instantiations of the PART_CONTAINER specification can be adapted to specific purposes by extending them with axioms defining *addable*.

The above generated datatype declaration abbreviates as usual the declaration of a sort *P_Container*, a constant constructor *empty*, and a partial constructor $insert : Elem \times P\_Container \rightarrow ?\ P\_Container$. It also entails the corresponding generatedness constraint.

4.4 Existential Equality

Existential equality requires the definedness of both terms as well as their equality.

spec NATURAL_PARTIAL_SUBTRACTION_2 =
NATURAL_PARTIAL_SUBTRACTION_1
then $\forall x, y, z : Nat \bullet\ y - x \overset{e}{=} z - x \Rightarrow y = z$
%{ $y - x = z - x \Rightarrow y = z$ would be wrong,
$def(y - x) \wedge def(z - x) \wedge y - x = z - x \Rightarrow y = z$
is correct, but better abbreviated in the above axiom **}%**
end

An *existential* equation $t1 \overset{e}{=} t2$ is equivalent to $def(t1) \wedge def(t2) \wedge t1 = t2$, so it holds if and only if both terms $t1$ and $t2$ are defined and denote the same value. Existential equality '$\overset{e}{=}$' is input as '`=e=`'.

Note that while a trivial strong equation of the form $t = t$ always holds, this is not the case for existential equations. For instance, the trivial existential equation $x - y \overset{e}{=} x - y$ does not hold, since the term $x - y$ may be undefined.

In general consequences of undefinedness are undesirable. Hence a conditional equation of the form $t1 = t2 \Rightarrow t3 = t4$ is often wrong if $t1$ and $t2$ may be undefined, because the equality $t3 = t4$ would be implied when both $t1$ and $t2$ are undefined (since then the strong equation $t1 = t2$ would hold). The above specification provides a typical example of such a situation: $y - x = z - x \Rightarrow y = z$ would be wrong, since it would entail that any two arbitrary values y and z are equal (it is enough to choose an x greater than y and z to make $y - x$ and $z - x$ both undefined).

Therefore, to avoid such undesirable consequences of undefinedness, it is advisable to use existential equations instead of strong equations in the premises of conditional equations involving partial operations. An alternative is to add the relevant definedness assertions explicitly to the equations in the premises.