METAL: a Metalanguage for SMoLCS*

Version 4.2 – 02/8/95

Fabio Parodi – Gianna Reggio
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
email smolcsdisi.unige.it

1 Data type specifications

In this section we present the constructs of METAL for specifying data types. A data type is:

- a family of sets, called carries;
- some constants, i.e. particular elements of the carries;
- some functions manipulating the elements of the carries; each function taken some arguments belonging to some carries returns a result belonging to some other carrier;
- some predicates expressing conditions on the elements of the carries; each predicate taken some arguments belonging to some carries is either true or false.

Example 1.1 Data type "natural numbers"
Consider a very simple data type, "natural numbers", consisting of:

– the set of the natural numbers $\mathbb{N}$;

– the constant $0 \in \mathbb{N}$;

– the unary function which taken a number returns the successor;

– the binary function which adds two numbers;

– the binary predicate which is true if the first argument is less than the second one.

End example

*This work has been supported by a grant ENEL/CRA (Milano Italy) and by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. (Italy).

Example 1.2 Data type "stack of natural numbers"
The data type stack is a structure containing some elements, in this case natural numbers, regulating the insertion and the extraction in such a way that the last number inserted will be the first to be extracted; it is as if the numbers within the stack are put one over the other, in a way that only that on the top is accessible. This data type consists of:

- the set of the natural numbers $\mathbb{N}$;
- the set of the finite sequences of naturals $\mathbb{N}^* = \cup_{i \in \mathbb{N}} \mathbb{N}^i$;
  (recall that $\mathbb{N}^i = \mathbb{N} \times \ldots \times \mathbb{N}$, i times, if $i > 0$, and that $\mathbb{N}^0 = \{\Lambda\}$);
- the constant, the functions and the predicate of the data type of the natural numbers of the Ex. **??**;
- the constant $\Lambda \in \mathbb{N}^*$, i.e. the empty stack;
- the function which taken a number and a stack returns the stack with such number put on the top;
- the function which taken a stack returns the number on the top (0 if the stack is empty);
- the function which taken a stack returns such stack where the number on the top has been taken away (Λ if the stack is empty);
- the predicate checking if a stack is empty.

End example

In the definition of the data types we can distinguish a syntactic/structural part, called signature, describing which are the components (carries, constants, functions and predicates), and a semantic part, describing how such components should be interpreted. The signature is the syntactic aspect of the data type. The components of the signature are related with the components of the data type following the schema below.

Data type

Syntax (signature)	Semantic (interpretation)
sort	carrier (set)
constant name	particular element of a carrier
operation name	function
predicate name	predicate

To specify a data type means to define first its signature, and then to define the interpretation of its components. For defining the interpretation we give the properties that it must satisfy; such properties say which result gives a function when applied to some arguments, and if a predicate applied to some arguments is true or false. In METAL these properties are expressed by syntactic constructs called axioms.

Specification of a data type

signature	components of the data type
axioms	semantic properties

The properties of a data type are expressed independently by the representation concrete of the elements of the carries and by the way in which are calculated the operations and evaluated the predicates; for example an important property of the natural numbers is that "for each natural number n, $0+n=n$", and this fact must hold if we represent the numbers as binary sequences or as sequences of decimal digits, and for whatever algorithm more or less efficient we choose for computing the sum.

1.1 Signatures

The signature expresses the (syntactic) structure of the data type.

- A name is given for each carrier, called *sort*; the notion of sort intuitively corresponds to the notion of "type" in the usual programming languages as Pascal and Ada.

  Consider the data type "stacks of natural numbers" of the Ex. **??**: the sort `nat` may be the name of the carrier $\mathbb{N}$; the sort `stack` that of the carrier $\mathbb{N}^*$. Then we can say that $\mathbb{N}$ is the carrier associated with `nat` and that $\mathbb{N}^*$ is the carrier associated with `stack`.

- A name and a sort are given for each constant; the sort denotes the carrier to whom the constant belong.

- A name and an arity are given for each function; the arity of an operation is the list of the sorts of the arguments and the sort of the result.

- A name and an arity are given for each predicate; the arity of a predicate is the list of the sorts of the arguments.

Example 1.3 Signature of the data type "natural numbers"
Referring to the data type "natural numbers" defined in the Ex. **??**, in METAL its signature may be defined as follows:

```
** this is a comment
sort  nat  **  sort corresponding to the carrier natural numbers
cn  Zero: nat}   **  constant
op  Succ: nat: nat   **  unary operation
op  Sum: nat nat -> nat  **  binary operation
pr  Ge: nat nat  **  binary predicate
```

Notice that the sorts are denoted by lower case identifiers, while constants, operations and predicates are represented by identifiers which start with an upper case letter.
End example

Example 1.4 Signature of the data type "stacks of natural numbers"
Referring to the data type "stacks of natural numbers" defined in the Ex. **??**, in METAL its signature may be defined starting from the signature defined in **??** adding what follows:

```
sort   stack
cn  Empty: stack
op  Put: nat stack -> stack
op  Get: stack -> stack
op  First: stack -> nat
pr  IsEmpty: stack
```

End example

Syntax
A signature is a sequence of declarations of sorts, constants, operations and predicates.

$$
\begin{array}{lll}
\mathit{SignatureDec} & ::= & \texttt{sort}\ \mathit{Sort}_1\ \ldots\ \mathit{Sort}_n \ \mid \\
 & & \texttt{cn}\ \mathit{UpIdent}:\ \ \mathit{Sort} \ \mid \\
 & & \texttt{op}\ \mathit{UpIdent}:\ \ \mathit{Sort}_1\ \ldots\ \mathit{Sort}_n\ \texttt{->}\ \mathit{Sort} \ \mid \\
 & & \texttt{pr}\ \mathit{UpIdent}:\ \ \mathit{Sort}_1\ \ldots\ \mathit{Sort}_n
\end{array}
$$

$\mathit{Sort}\ ::=\ \mathit{LowIdent}$

The sorts used in the declarations of the constants, of the operations and of the predicates must be part of the signature, and must be declared before. For example the following signature contains two errors .

```
op  X: a -> a   **  error: a has not still been declared
sort  a  b
op  Y: a -> b   **  OK
op  Z: a -> c   **  error: c is not in the signature
```

It is prohibited to declare twice the same sort. It is instead allowed to declare constants, operations and predicates with the same name, but they must have different arities (see subsection ??).

1.2 Terms and atoms over a signature

A signature Σ defines the syntax of a data type; for using this data type we need a way for representing the elements of the carries, the application of the operations and predicates to such elements. For doing all that in METAL we use appropriate constructs, which are the terms and the atoms over the signature.

The terms over Σ are used to represent the elements of the carrier and the application of the operations to such elements; the atoms over Σ are used for expressing some basic logic conditions on the elements of the carries, given either by the application of the predicates of the signature, or by the comparisons between terms.

The terms over a signature Σ are obtained by composing the constants and the operations; i.e. a term can be:

- a constant name;
- an operation name applied to some arguments, which in turn are represented by terms.

Referring to the signature of the Ex. ??, the terms `Zero`, `Succ(Zero)`, `Succ(Succ(Zero))` and so long represent elements of the carrier of `nat`. The term `Sum(Zero,Succ(Zero))` represents the application of the operation denoted by `Sum` to the elements represented by the terms `Zero` and `Succ(Zero)`.

An atom over a signature Σ can be:

- a predicate name applied to some arguments,which are represented by terms;
- a comparison between two terms of the same sort.

Referring to the signature of the Ex. ?? the following are atoms:
`IsEmpty(Empty)`, `Zero = Succ(Zero)`, `Empty = Put(Succ(Zero),Empty)`.

METAL is a typed language, in the sense that it does not allow to confuse different sorts. This feature helps to separating in a clear way distinct concepts and allows to determine some conceptual errors due to the confusion.

For example, always referring to the signature of Ex. ??, the terms `Succ(Empty)`, `Put(Zero,Zero)`, `Get(Zero,Empty)` and the atoms `Zero = Empty`, `Ge(Zero,Put(Zero,Empty))`, `IsEmpty(Zero)` are wrong since the arguments do not respect the arities of the operations or of the predicates.

Syntax

$$\begin{array}{lcl} \mathit{Term} & ::= & \mathit{UpIdent} \mid \mathit{UpIdent}\texttt{(}\mathit{Term}_1\texttt{, ..., }\mathit{Term}_n\texttt{)} \\ \mathit{Atom} & ::= & \mathit{UpIdent}\texttt{(}\mathit{Term}_1\texttt{, ..., }\mathit{Term}_n\texttt{)} \mid \mathit{Term}_L \texttt{ = } \mathit{Term}_R \end{array}$$

Not all the terms and the atoms obtained by the above grammar are correct: the requirements about the arities of the operations and of the predicates are the following.

- `C` is a correct term of sort `s` if `C: s` is a constant of the signature.
- `O(t1, ...,tn)` is a correct term of sort `s` if `O: s1 ... sn -> s` is an operation of the signature and `t1`, ..., `tn` are correct terms respectively of sorts `s1`, ..., `sn`.
- `P(t1, ...,tn)` is a correct atom if `P: s1 ... sn` is a predicate of the signature and `t1`, ..., `tn` are correct terms respectively of sort `s1`, ..., `sn`.
- `tL = tR` is a correct atom if `tL` and `tR` are correct terms of the same sort.

1.2.1 Variables

"In the data type natural numbers, whatever number added to zero gives as result the number itself". For expressing properties of this type in mathematics we use the variables, writing in a way very compact and expressive $\forall\, n \in \mathbb{N},\ n+0=n$; in METAL the concept of variable is used analogously.

The variables are used to represent generic elements of the carrier of a certain sort; they are represented by lower case identifiers and may be used for building the terms, together constants and operations. For example if we add to the declarations of the Ex. ?? the declaration `var n: nat`; then `n` and `Sum(Zero,n)` are terms with variables of sort `nat`.

Example 1.5
Consider the following signature with variables:

```
sort  nat
var  n  m: nat
cn  Zero: nat
op  Succ: nat -> nat
op  Sum: nat nat -> nat
pr  Ge: nat nat

sort  pair
var  p: pair
op  Pair: nat nat -> pair
op  First: pair -> nat
op  Second: pair -> nat
pr  In: nat pair
```

The following are correct terms: `p`, `Zero`, `Pair(n,Sum(Zero,n))`, `First(p)`;
while these are wrong: `Second(n)`, `Pair(n,Pair(Zero,Zero))`, `Pair(p)`;
the following are correct atoms: `Ge(n,m)`, `Second(p) = Zero`;
while these are wrong: `In(p,p)`, `Zero = p`, `Sum(n,m) = Second(n)`. **End example**

Syntax

VarDec ::= `var` $LowIdent_1$ ... $LowIdent_n$: *Sort*
Term ::= *LowIdent*

The sorts appearing in a variable declaration must be declared before to be used.

Given a signature, `v` is a correct term of sort `s` if `v` is a variable of sort `s` and has been declared before to be used.

1.2.2 Overloading

The overloading of the names of constant, operation and predicate is a feature of some signatures. In METAL a name of constant, of operation or of predicate can be overloaded with many different meanings. For example `Sum`, that in the previous examples represented the sum of natural numbers, can be used also for representing other operations, as the sum of integer numbers, the lists concatenation, or the adding of an element to a table.

The advantage of using the overloading is that we can choose names for the operations/ predicates which are really meaningful without be too long. In this way we can write more clear and compact specification. The disadvantage is that, since a name may have various interpretations, sometime it is hard to understand which is the right one; in some cases it is the context to establish it, in other cases the context is not sufficient and so we have ambiguous terms and atoms.

However we cannot abuse of the overloading, otherwise we risk of getting specifications of difficult comprehension for two reasons: the first is that a nonambiguous term/atom containing names with several meanings can be difficult to interpret for a human reader; the second is that for eliminating the ambiguity sometime we have to rewrite terms and atoms in a complex and not natural way.

Example 1.6
Consider the following signature, for a data type putting together natural and integer numbers:

```
sort  nat int
var  n  m: nat
var  i: int

cn  Zero: nat
op  Succ: nat -> nat
op  Sum: nat nat -> nat

cn  Zero: int
op  Succ -> int -> int
op  Pred: int -> int
op  Sum: int  int -> int
op  NatInt: nat -> int
op  Abs: int -> nat
```

Notice that some operations over naturals, different from that on the integers, are denoted by the same symbols (`Zero`, `Succ`, `Sum`).

The following are ambiguous terms: `Zero`, `Succ(Zero)`, `Sum(Zero,Succ(Zero))`;
while these one are not ambiguous: `NatInt(Succ(Zero))`, `Sum(Zero,n)` and
the following are wrong terms: `NatInt(Sum(Zero,i))`, `Abs(n)`. **End example**

1.2.3 Constants, operations and predicates with mixed infix syntax

The syntax used until now for terms (and atoms) is called functional, or prefix with parenthesis, and it is based on writing the name of the operation (or of the predicate) followed by the arguments between parentheses, separated by commas. Usually in mathematics more compact and expressive notations are used, as for example the binary infixed notation used for the adding and multiplication operations between numbers. This since

`1 + Zero` is more clear of `Sum(1,Zero)`.

METAL allows to define together the names of constants, operations and predicates also the syntax for using them. The notations binary infix, unary prefix, polish prefix and polish postfix are particular cases of the general mechanism of syntactic definition supported by METAL. This mechanism is called "mixed infix notation".

In the declarations of constants, operations and predicates we give the syntax with we intend to use tehm in terms and atoms. The character underscore "_" denotes the place of the arguments, following the declaration order. There are as many "_" as the arguments (and then in the declarations of constants no _). Building terms and atoms, each "_" is replaced by a term of the corresponding sort.

Example 1.7 Pairs of naturals with mixed infix syntax
For giving an example of mixed syntax, we consider a signature for the data type "pairs of natural numbers" (another signature for the same data type, but with functional syntax, is in the Ex. **??**).

```
sort  nat pair
var  n m: nat
var  p: pair

cn  0: nat
op  Succ _ : nat -> nat
op  _  + _ : nat nat -> nat
op  _  * _ : nat nat -> nat
op  _  ! : nat -> nat
pr  _  <= _ : nat nat

op  < _   _  >: nat nat -> pair
op  _ 1: pair -> nat
op  _ 2: pair -> nat
pr  _ In _ : nat  pair
```

The following are correct and non ambiguous terms of sort `nat`: `0`, `Succ 0`, `0 + Succ n`, `p 1`, `< Succ 0 0 > 1`, `n!`; the following are correct but ambiguous terms of sort `nat`: `n + 0 + m`, `Succ n !` and those are correct terms of sort `pair`: `p`, `< n m >`.

The following are correct atoms: `0 = 0`, `Succ 0 = 0`, `0 In p`; while `n + m + 0 = n + m` is an ambiguous atom.

Notice that some of these terms and atoms are ambiguous also if there is no overloading. **End example**

1.2.4 Ambiguity of terms and atoms

The ambiguity of the terms can be caused by the overloading, by the mixed infix syntax, or by a combination of them, and sometime we can solve it using the context in which the term is inserted; for this reason ambiguous terms are allowed by the language. The ambiguity of the atoms is caused by the ambiguity of the component terms, and cannot be solved using the context; for this reason the ambiguous atoms are wrong.

The ambiguity caused by the overloading can be eliminated stating explicitly the sort of the terms. It is as if for each sort `s` there is an operation `s: s -> s`, called explicit typing operation, or casting. This operation is used for coercing a term to have sort `s`.

Example 1.8 Overloading and ambiguous terms
Consider the signature of the Ex. **??**.

ambiguous term	corresponding non ambiguous terms
`Zero`	`nat(Zero)`, `int(Zero)`
`Succ(Zero)`	`Succ(nat(Zero))`, `int(Succ(Zero))`
`Sum(Zero, Succ(Zero))`	`Sum(int(Zero),Succ(Zero))`,
	`Sum(nat(Zero),Succ(Zero))`,
	`nat(Sum(Zero,Succ(Zero)))`

End example

The ambiguity caused by the mixed infix syntax is solved putting round parentheses around the terms to whom we want to give precedence.

Example 1.9 Mixed syntax and ambiguous terms
Referring to the signature of the Ex. **??**, consider the following terms.

- `0`, `< 0 Succ 0 >`, `0 + n`, `p 1` are correct and nonambiguous.
- `0 + n + m` is ambiguous: it can be `0 + (n + m)` or `(0 + n) + m`.
- `Succ 0 + n` is ambiguous: it can be `(Succ 0) + n` or `Succ(0 + n)`.
- `Succ n !` is ambiguous: it can be `(Succ n)!` or `Succ(n!)`.
- `p In 0`, `Succ < 0 0 >`, `< < 0 Succ 0 > n >`, `n 1`, `p!`, `< < 0 0 > 0>` are wrong.

End example

1.2.5 Implicit embeddings

The unary operations of the kind `_ -> s -> s'` are called implicit embeddings; in the terms they are invisible and can cause ambiguities. Consider the following example.

Example 1.10 Sequences of natural numbers

```
sort  nat seq
var  n: nat
var  q: seq

cn  0: nat
op  Succ _ : nat -> nat

op  _ : nat -> seq
op  _   _ : nat  seq -> seq

pr  _  In _ : nat  seq
```

The following are correct terms of sort `seq` over the above signature:

`0`, `Succ 0`, `Succ 0 0`, `Succ 0 0 0`, ...

The term `0` is ambiguous since it can be of sort `nat` or of sort `seq`; also the other terms of sort `nat` are ambiguous, since due to the implicit embedding they are also terms of sort `seq`. **End example**

The ambiguities caused by the implicit embedding operations are extremely dangerous, and sometime are very difficult to solve; thus the use of these operations is limited by the following restrictions.

- No cycles of implicit embeddings. For example the three operations "`_ : a -> b`", "`_ : b -> c`", "`_ : c -> a`" cannot coexist in the same signature.
- An implicit embedding operation cannot be applied to a term having form `(t)`, when `t` is a term. In the case they are applied to `t` before to put the parentheses.
- An implicit embedding operation cannot be applied to the result of an operation of explicit typing, i.e. to a term having form `s(t)`, where `s` is a sort and `t` is a term.

Syntax

SignatureDec ::= `cn` *MixfixRepr*: *Sort* |
  `op` *MixfixRepr*: $Sort_1$ ... $Sort_n$ `->` *Sort* |
  `pr` *MixfixRepr*: $Sort_1$ ... $Sort_n$
MixfixRepr ::= $MixfixReprElem_1$ ... $MixfixReprElem_n$
MixfixReprElem ::= *Symbol* | *UpIdent* | _
Term ::= *Mixfix* | *Sort*`(`*Term*`)`
Atom ::= *Mixfix*
Mixfix ::= $MixfixElem_1$ ... $MixfixElem_n$
MixfixElem ::= *UpIdent* | *Symbol* | *Term* | `(`*Mixfix*`)`

The number of the "_" in an declaration of an operation or of a predicate must be equal to the number of the arguments; and the "_" cannot appear in a constant declarations.

A term is obtained from a constant name, or from an operation name with mixed infix syntax by replacing the _ with terms of sorts corresponding to the arguments; or by enclosing between parentheses another term; or by coercing the sort of another term.

An atom is obtained by the name of a predicate with infix mixed syntax by replacing the "_" with terms of sorts corresponding to the arguments; or by enclosing between parentheses a correct atom; moreover `tL = tR` is an atom if `tL` and `tR` are terms of the same sort, and no one of the two is obtained with the application to the outer level of an operation of implicit embedding. The ambiguous atoms are wrong.

1.3 Properties and axioms

The signatures are used to define the syntax of the data types, but not to expressing the semantic properties of the interpretations of the symbols of the signature. In particular, the signature cannot express:

- which are really the elements of the carries;
- which is the result of the application of an operation to some arguments;
- if a predicate applied to some arguments is true or false.

For specifying these aspects of the data types we give the properties of its components (constants, operations and predicates). Such properties are expressed in METAL with particular constructs called axioms.

Example 1.11 Axioms of the sum of naturals
Referring to the Ex. **??** of the pairs of natural numbers, we want to specify with some axioms the meaning of the sum, i.e. which is the result of the application of the function associated to the operation `_ + _` to two generic natural numbers.

This can be done in several ways. One of that consists in requiring the following two properties:

- for each natural number `n`, `n + 0 = n` holds;
- for each two natural numbers `n` and `m`, `n + (m + 1) = (n + m) + 1` holds.

These two properties characterizes completely the sum, in the sense that completely define the result of the sum of two arbitrary numbers.

Consider the signature with variables of the Ex. **??**: in METAL the above properties can be expressed by means of the two axioms:

```
ax  n + 0  =  n
ax  n + Succ m  =  Succ(n + m)
```

End example

When we use variables in an axiom, we means that the axiom holds whatever values assume the variables. For example the axiom `ax n + 0 = n` is read "for each possible value of `n` in the carrier of `nat`, the equality `n + 0 = n` holds".

The atoms are the simpler kind of axiom; more complex axioms can be given by composing the atoms using the logic combinators and by quantifying the variables.

1.3.1 Logic combinators

For giving the properties of the predicate `<=` of the Ex. **??** the atoms are not sufficient; we need more complex axioms, which are obtained by composing the atoms using the so called logic combinators.

Let `A1`, `A2` be two axioms. Then the following are axioms:
`not A1`, `A1 and A2`, `A1 or A2`, `A1 if A2`, `if A1 then A2`, `A1 iff A2`.

- `not` is the logic negation. `not A1` is true when `A1` is false and vice versa.
- `and` is the logic conjunction. `A1 and A2` is true when `A1` and `A2` are both true.
- `or` is the logic union. `A1 or A2` is true when either `A1` is true, or `A2` is true, or both.

– `if then` is the logic implication, i.e. the conditional form. `if A1 then A2` is true when either `A2` is true, or `A1` is false. Notice that if `A1` and `A2` are both false, the implication is true.

  The alternative form `A2 if A1` is equivalent to `if A1 then A2` and depending on the cases may be clearer.

– `iff` means "if and only if". `A1 iff A2` is true when `A1` and `A2` are either both true, or both false.

The meaning of the logic combinators is synthesized by the following tables:

A	not A
false	true
true	false

A1	A2	A1 and A2	A1 or A2	if A2 then A1	A1 iff A2
false	false	false	false	true	true
false	true	false	true	false	false
true	false	false	true	true	false
true	true	true	true	true	true

The axiom `A1 if A2` is equivalent to `if A2 then A1`, but it has the arguments swapped.

By a syntactic point of view the logic combinators have a mixed infix syntax, so axioms may be ambiguous; in this case some precedence between the logic combinators are given for establishing how to interpret an axiom in case of ambiguity; the rule is that the combinators which are more at left in the previous table have bigger precedence and have to be applied first: for example "`A1 or A2 and A3`" has to be interpreted "`A1 or (A2 and A3)`". The `not` has precedence over all the binary combinators.

The binary combinators associate on the left: for example "`A1 if A2 if A3`" has to be interpreted "`(A1 if A2) if A3`". In the case of `and` and `or` this is irrelevant since the result does not change.

If we want an interpretation different from that established by the precedence rules, we can use the parentheses for enclosing the axioms to be evaluated first. For example we can write "`(A1 or A2) and A3`".

Example 1.12
For showing the use of the logic combinators, we give some axioms on the signature of the Ex. **??**:

```
ax  n + 0  =  n
ax  n + Succ m  = Succ(n + m)
ax  n * 0  =  0
ax  n * (Succ m) = (n * m) + n
ax  0! =  1
ax  (Succ n)! = (Succ n) * (n!)
ax  0 <= n
ax  Succ n <= Succ m  if  n <= m
ax  < n  m >  1  =  n
ax  < n  m >  2  =  m
ax  n In p  if  n =  p  1 or n =  p  2
```

End example

Syntax

AxiomDec ::= `ax` *Axiom*
Axiom ::= *Atom* | (*Axiom*) | `not` *Axiom* |
Axiom$_1$ `and` *Axiom*$_2$ | *Axiom*$_1$ `or` *Axiom*$_2$ |
`if` *Axiom*$_1$ `then` *Axiom*$_2$ | *Axiom*$_1$ `if` *Axiom*$_2$ | *Axiom*$_1$ `iff` *Axiom*$_2$

We require that each atom appearing in an axiom must have one and only one interpretation, i.e. it must be correct and not ambiguous.

1.3.2 Quantifiers

The variable quantifiers are `forall` and `exists` and are applied to build some types of axiom. For example for expressing

"*there does not exists a natural number that added with itself gives an odd result*"

we can write

`ax not exists n: not exists m: n + n = (m + m) + Succ 0`

The quantifier `forall` used to the outer level is useless, since the variables are already considered "for each value', instead used with `exists` allows to express interesting properties.

Example 1.13
In this axiom, the value of `m` can be chosen depending on the value given to `n`:

```
ax  exists m: Sum(n,m) =  n
**  or equivalently
ax  forall n: exists m: Sum(n,m) =  n
```

Instead this axiom is stronger since says that there is a value for `m` which works independently from the value given to `n`:

`ax exists m: forall n: Sum(n,m) = n` **End example**

Syntax

Axiom ::= `forall` *LowIdent*$_1$ ... *LowIdent*$_n$: *Axiom* |
`exists` *LowIdent*$_1$ ... *LowIdent*$_n$: *Axiom*

LowIdent is a variable identifier, and must be declared before to be used in the axioms.

1.4 Specification of data types

The specification of a data type consists of a signature, a set of variables and a set of axioms. The signature includes declarations of sorts, constants, operations and predicates. The axioms define the properties that the interpretation of the components of the signature (constants, operations and predicates) must have.

1.4.1 Specification and data types.

Consider the relationship between a specification and the data type that it is denoting. Given a specification *Spec* consisting of the signature Σ and of the axioms Ax, we say that DT is a model of *Spec* if:

- to each sort `s` of Σ corresponds a carrier DT_s of DT;
- to each symbol of constant `C: s` of Σ corresponds an element $DT_C \in DT_s$;
- to each symbol of operation `O: s1 ... sn -> s` of Σ corresponds a function DT_O: $DT_{s1} \times \ldots \times DT_{sn}$ `->` DT_s of DT;
- to each symbol of predicate `P: s1 ... sn` of Σ corresponds a predicate DT_P: $DT_{s1} \times \ldots \times DT_{sn}$ of DT;
- the axioms in Ax hold in DT.

  This means that taken an axiom `A`$\in Ax$, for each assignment of values to variables of `A` the interpretation of `A` in DT is true; moreover in DT also all the axioms which are logic consequences of that in Ax hold. For example, if `A1`, `A2`$\in Ax$ then also the axiom `A1 and A2` holds in DT.

  Notice that in general in DT also other formulae, not derivate by the axioms, may hold.

In general a specification can have either several models, or no one. If a specification has many different models, all of them have the structure imposed by the signature, and satisfy the axioms. A specification has not models if its axioms are contradictory, and so there does not exist a data type satisfying all of them.

Sometime there is a particular model that it is interesting; other times instead there are different models equally interesting.

Consider the following two examples, that show two completely different ways of considering the meaning of the specification of a data type.

Example 1.14 Naturals with ordering relationship $\leq$
The following declarations specify the natural numbers with the ordering relationship $\leq$, denoted by the predicate `<=`.

```
sort  nat
var  n  m: nat
cn  0: nat
op  Succ _ : nat -> nat
pr  _  <= _ : nat nat
ax  0 <= m
ax  if  n <= m  then Succ n <= Succ m
```

This specification defines a precise data type, consisting of a set isomorphic to the set of the natural numbers, with the constant zero, the operation that taken a number gives the successor and the predicate corresponding to the increasing order of the naturals.

In this case, we want that the data type denoted by the specification is uniquely determined (up to isomorphism); and so the axioms give all and only the properties of constants, operations and predicates. **End example**

Example 1.15 Sets with an order relation
The following declarations specify the general properties that whatever total order relation must satisfy.

```
sort  toset
var  x y z: toset
pr  _  <= _ : toset toset
ax  x <= x   **  reflexivity
ax  if  x <= y and y <= z  then  x <= z  **  transitivity
ax  if  x <= y and y <= x  then  x  =  y   **  antisimmetry
ax  x <= y or y <= x  **  totality
```

This specification intends to define a class of data types, i.e. not a unique data type, but several ones. Indeed all sets with a total ordering relation are models of this specification; for example the natural numbers, the real numbers, the english words with the lexicography ordering and so long. **End example**

These are two completely different approaches to the problem of giving a meaning to a specification. Precisely there are two kinds of specifications.

- Design specification, as the specification of the natural numbers of the Ex. **??**, which denote a unique data type.
- Requirement specification, as the specification of the ordered sets of the Ex. **??**, which denote a very large class of data types.

The crucial difference between these two approaches is in the way the axioms are considered: in the first case the axioms completely characterizes all the properties of the data type; in the second case the axioms give the properties which surely must hold, leaving open the possibility that other properties hold.

1.4.2 Order of the declarations

In METAL the declarations composing a signature can be mixed with the declarations of the variables and of the axioms, whenever each symbol is declared before to be used. This freedom allows to structure the specification in a proper way, and must be used carefully; in general we recommend of following one of the two schemas:

- Separation between signature and axioms: first we declare the sorts; followed by the constants, the operations and the predicates; then the variables and at last the axioms. In this way the signature of the data type is put in evidence.

- Integration between signature and axioms: first we declare the sort, followed by the variables, and after the constants, the operations and the predicates together with the relative axioms. In this way the link between axioms and relative operations is in evidence; there is the risk of loosing the view of the signature, which is spread out in the specification.

1.5 Design specifications

A design specification of a data type includes as usually a signature and some axioms; we want that denotes a data type with *exactly* the sorts, the constants, the operations and the predicates declared in the signature, and that *satisfies all and only the properties* expressed by the axioms and their consequences.

If we consider the specification of the Ex. ??, we do not want that in the denoted type the property `Succ Succ 0 = 0`, i.e. `2 = 0` hold. This property is not a logic consequence of the axioms and thus it must not hold in the model in which we are interested.

Moreover we want that the carrier of the sort `nat` does not contain elements not expressible with the constants and the operations: the data type of the real numbers is not appropriate since the irrational numbers cannot be expressed using only `0` and `Succ`.

The data type DT denoted by a design specification SP with signature Σ and axioms Ax is characterized by the following properties.

- DT is a model of SP.
- For expressing all the elements of the carries of DT the constants and the operations of Σ are sufficient; in other word, DT is generated by Σ.
- Let `A` be an axiom over Σ; `A` holds in DT if and only if `A` is a logic consequence of some axioms Ax.

In general a specification has many models, but there only one (up to isomorphism) satisfying these three properties; such data type is said the initial model of the specification.

In METAL we have a design specification enclosing the signature declarations and axioms between the key words `design` and `end`.

Example 1.16 Models of the specification of the naturals
Consider the following specification:

```
design
sort  nat
var x  y: nat
cn  0: nat
op  Succ _  : nat -> nat
op  _  + _  : nat nat -> nat
ax  x + 0  =  x
ax  x + Succ y  =  Succ(x + y)
end
```

This specification denotes the data type of the natural numbers (which is its initial model), since the declarations of signature and axioms are enclosed between `design` and `end`.

But the same set of declarations has many models; in the following we give two examples of data types different from the initial model.

In the trivial model the carrier of `nat` contains a unique element $\bullet$ and all the operations have as result $\bullet$. Obviously the equation `Succ 0 = 0` holds in this model, also if it is not a consequence of the axioms: thus this model is not initial.

In the model of the real numbers the carrier of `nat` is *Real* and includes positive and negative real numbers, with fractions, algebraic roots and so long. This model satisfies the first and the third requirement, but not the second: indeed there exists a number $n \in \mathit{Real}$ which cannot be represented with `0`, `Succ` and `+`, e.g. π, and thus this model does not satisfies the second requirement. **End example**

1.5.1 Existence of the initial model

Depending on the form of the axioms of a specification, it can happen that the initial model determined by a design specification, i.e. the data type satisfying the three requirements given previously, does not exist. However it can be shown that if the axioms have a certain form, called positive conditional, then the initial model exists. In other words if we use only positive conditional axioms, we are guaranteed that the initial model does exist; for this reason in the design specification we can only use positive conditional axioms. An axiom is positive conditional if has one of the following forms:

- *atom*
- *atom* `if` $atom_1$ `and` ... `and` $atom_n$;
- `if` $atom_1$ `and` ... `and` $atom_n$ `then` *atom*.

Some kinds of axioms which are not positive conditional can be reformulated in an equivalent way as a set of positive conditional axioms. For giving some examples, let `A`, `B` and `C` be some atoms.

The axiom "`if A or B then C`" is equivalent to the pair of positive conditional axioms "`if A then C`", "`if B then C`".

The axiom "`A and B`" is equivalent to the pair of positive conditional axioms "`A`", "`B`"; since `A and B` means that both `A` and `B` must hold.

Analogously the axiom "`if A then B and C`" is equivalent to the pair of positive conditional axioms "`if A then B`", "`if A then C`".

Instead the following axioms in general are not equivalent to other positive conditional axioms: "`A or B`", "`(A or B) if C`", "`not A`", "`exists x: A`".

Syntax

BasicDec ::= *SignatureDec* | *VarDec* | *AxiomDec*
Spec ::= `design` $\mathit{BasicDec}_1$ ... $\mathit{BasicDec}_n$ `end`

All the axioms appearing in a design specification must be positive conditional. The sorts, the constants, the operations, the predicates and the variables must be declared before to be used.

1.6 Requirement specification

In some cases it is desirable to specify only very general properties of a class of data types. A specification of this kind denotes the class of all its models, i.e. the class of the data types DT such that DT has the sorts, the constants, the operations and the predicates of the signature of the specification, and satisfies all its axioms.

The other constraints for the initial model in this case are not required. In particular it is not necessary that the carries of DT are generated by constants and operations of the specification: elements non represented by terms can be in the carriers; clearly the axioms and their logic consequences must hold in DT; however it is possible that other properties, which are are not consequences of the axioms hold.

In METAL we have a requirement specification[1] i.e. interpreted in the above way by enclosing the declarations of signature and the axioms between the keywords `requirement` and `end`.

Example 1.17 Group requirements
In mathematics a group is a set with a binary operation, called sum, which has a neutral element and is associative, and for whom each element has an inverse. The real numbers w.r.t. the product, the integers w.r.t. the sum are examples of groups.

Whatever model of the following specification is a group, and whatever group can be seen as a model of the following specification.

```
requirement   **  properties of the groups
sort  group
var  x  y  z: group

cn  Id: group  **  identity
op  _  + _ : group group -> group  **  sum

ax  x + (y + z) =  (x + y) + z  **  associativity
**  Id is the identity}
ax  x + Id  =  x
ax  Id + x  =  x
ax  forall x: exists y: x + y  =  Id  **  existence of the inverse
end
```

End example

In the requirement approach the axioms may have whatever form and so we can use freely the `not`, the `or` combinators and the quantifiers.

[1]In the scientific literature this way of interpreting a specification is known as loose approach.

Syntax

Spec ::= `requirement` $BasicDec_1$ ... $BasicDec_n$ `end`

The sorts, the constants, the operations, the predicates and the variables must be declared before to be used.

1.7 Two final examples

This paragraph contains two final examples including all the features of METAL presented in the section. The first is a design specification, the second a requirement specification.

Example 1.18 Data type "priority queues of natural numbers"

```
design  **  natural numbers
sort  nat
var  n  m: nat

cn  0: nat
op  Succ _ : nat -> nat
op  _  + _ : nat nat -> nat
ax  0 + n =  n
ax  (Succ m) + n  =  Succ(m + n)

pr  _  <= _ : nat nat
ax  0 <= n
ax  if n <= m  then  Succ n <= Succ m

op  Max: nat nat -> nat
ax  if n <= m  then  Max(n,m) =  m
ax  if m <= n  then  Max(n,m) =  n

pr  _  =/= _ : nat nat
ax  if  x <= y  then  x =/= Succ y
ax  x =/= y  if  y =/= x

sort  queue  **  priority queues
var  q: queue

cn  <> : queue
op  Add : nat  queue -> queue

pr  _  Is Empty: queue
ax  <> Is Empty
```

```
op  Max _  : queue -> nat  **  computes the maximum
ax  Max <>  =  0
ax  Max Add(n,q) =  Max(n,Max q)

op  Rest _ : queue -> queue   **  gets the maximum
ax  Rest <>  =  <>
ax  Rest Add(n,q) =  q  if n  =  Max Add(n,q)
ax  Rest Add(n,q) =  Rest q  if n =/= Max Add(n,q)
end
```

Notice that the last axiom is positive conditional, due to the introduction of the auxiliary predicate `=/=` on naturals. **End example**

Example 1.19 Class of the data types with a bijective operation
Consider two sets, say *DOM* and *CODOM*, linked by a function f which taken an element of *DOM* returns an element of *CODOM*, such that:

- (injectivity) whatever two elements a and b of *DOM* we take, if $a \neq b$ then $f(a) \neq f(b)$;
- (surjectivity) whatever element c of *CODOM* we take, there exists an element a of *DOM* such that $f(a) = c$.

A function f having this two properties is said bijectivity. This situation can be specified as follows:

```
requirement
sort  dom codom   **  domain and codomain
op  F: dom -> codom   **  bijective operation

var  x  y: dom
var  n: codom

ax  if not x  =  y  then  not F(x) =  F(y)  **  injectivity
ax  exists x: F(x) =  n}   **  surjectivity
end
```

End example

2 Specifications structured in modules

In METAL it is possible to split a specification in several separated modules; each module corresponds to an isolated aspect of the problem, and its reduced dimension makes its content immediately understandable. Each module is connected to the other in a clear and ordered way; the division of a specification in modules allows to reuse the same components in various different situations, with a consistent save of time and of work.

In METAL we give a name to each module by means of an association with an identifier called "bind"; in this way we define an environment associating modules with identifiers.

Developing a specification of a system, we usually start from an informal description of the minimal requirements that the system must satisfy. This description is formalized with a requirement specification, leaving space to different interpretations. The specification is then refined in several steps until we get a detailed design specification of the project. During this refinement it is possible that some modules are terminated before other, and thus in the intermediate steps we have requirement specifications with some subcomponents already at the design level.

Example 2.1 Specification of the stacks of naturals
Consider the following modular specification of the data type "stacks of natural numbers"; first we specify the natural numbers, then we use the naturals for specifying the stacks. We associate the specification of the data type "natural numbers" with the identifier `NAT` by the following bind.

```
NAT =
design
sort  nat
cn  0: nat
op  Succ _: nat -> nat
```

Then we associate the specification of the data type "stacks of natural numbers" with the identifier `STACK_OF_NATURAL` by writing

```
STACK_OF_NATURAL =
design
use  NAT
var  n: nat
sort  stack
var  s: stack

cn  Empty: stack
op  Put: nat  stack -> stack

op  Get: nat  stack -> stack
ax  Get(Put(n,s)) = s
ax  Get(Empty) = Empty

op  First: nat stack -> nat
ax  First(Put(n,s)) = n
**  be careful: First(Empty) is different from all naturals of NAT

pr  _ Is Empty: stack
ax  Empty  Is Empty
```

Now the data type denoted by `STACK_OF_NATURAL` includes also the naturals denoted by `NAT`, and it is in some sense an extension of it. **End example**

The principal constructs of METAL about the structuring in modules of the specifications are the following.

- "`use`" allows to define a specification using other specifications defined previously as subparts. If `A` uses `B`, the elements of the signature of `B` are available in `A`, with their semantic properties, and thus there is no need to redeclare them.
- "`rename`" allows to change name to some sorts, constants, operations and predicates of a module. In this way we can reuse a module, by changing some of its symbols, for adapting it to the new needs.
- The parameterization is an important tool for reusing the specifications; it allows to specify generic data types, that can be instantiated afterwards.

  For example we can define the generic the data type stack, without to say which are the objects of the stack. After, this data type can be instantiated by creating stacks of naturals, stacks of streams of naturals, stacks of stacks of trees of pairs of naturals ....
- "`export`" and "`hide`" allow to control which symbols of the signature of the specification defined in a module become available outside and which not.

These operations among specifications can be applied to the basic specifications both requirement and design, for getting new specifications; in this way after having written some basic modules (and using those built-in) we can build more complex specifications of data types.

Together with requirement and design METAL has two other kinds of specifications.

- The draft specification are modules without any semantic, neither initial nor loose, but which simply denote a signature and some axioms. They are used for solving some problems of modular decomposition.
- The param specifications are used within the parametric specifications, for expressing the requirements that their parameters must satisfy. They have loose semantic and differ by the requirement specifications only in the use.

2.1 Binds and environments

A bind is the association of an identifier with a specification. The Ex. **??** contains two binds, which associate two specifications with the two identifiers `NAT` and `STACK_OF_NATURAL`.

A sequence of binds in which all the identifiers on the left of the equalities are distinct is an environment.

For giving the semantic to a specification we need to know the binds to whom it refers; for example `STACK_OF_NATURAL` refers to `NAT`, and for giving a meaning to `STACK_OF_NATURAL` first we have to give a meaning to `NAT`. Among the binds of an environment there is thus a dependence relationship, without cycles; moreover within a bind for an identifier it is not possible to refer to the same identifier.

Syntax

BindSequence ::= $Bind_1$ ... $Bind_n$
Bind ::= *UpUpIdent* `=` *Spec*

The identifiers on the left must be all distinct and each identifier must be declared before to be used.

2.2 Use of modules

Let `A = SA` and `B = SB` be two binds of an environment.

If the declaration `use B` appears in `SA`, from that point we can use the sorts, the constants, the operations and the predicates defined in `B`. The variables cannot be used, since they are local to the modules. In such case we say that `A` uses `B`.

Depending on the semantic of `SA` and of `SB` there are some restrictions to `A` using `B`.

- `SA` design, `SB` design: it is desirable that `SA` does not change the properties of `SB`. For this reason we put two restrictions, called *no junk* and *no confusion*, presented in the subsection **??**.
- `SA` design, `SB` draft: it is allowed, if all the axioms of `SB` are positive conditional.
- `SA` design, requirement or draft, `SB` param: error; a param specification can be used only within other param modules.
- `SA` requirement or param, `SB` design: in this case the data type denoted by `SB` w.r.t. the initial semantic is inserted in the data type denoted by `SA`, which has a loose semantic.

  Notice that whatever model of `SA` restricted to the signature of `SB` must coincide with the initial model of `SB`. That means that the restrictions *no junk* and *no confusion*, expressed in subsection **??**, hold. This is the situation during the refinement of the specification of a system when while we specify the requirements of the whole system some submodules are already developed (i.e. given by design specifications).
- `SA` requirement or param, `SB` requirement or draft: no problems.
- `SA` param, `SB` param, design, requirement or draft: no problems.
- `SA` draft, `SB` draft, design or requirement: it is possible, but we cannot guarantee any properties of consistency, since `SA` has not a semantic in terms of data types.

Syntax

Spec ::= *UpUpIdent*
BasicDec ::= `use` $Spec_1$, ... , $Spec_n$

The restrictions on the importing of modules are expressed by the table below.

A uses B	SA design	SA requirement	SA draft	SA param
SB design	no junk, no confusion	no junk, no confusion	OK	no junk, no confusion
SB requirement	Error	OK	OK	OK
SB draft	SB with only positive conditional axioms	OK	OK	OK
SB param	Error	Error	Error	OK

2.2.1 No junk, no confusion

Let `A = SA` and `B = SB` be two binds, with SB design (with initial semantic), such that A uses B. B denotes uniquely a data type, and it is important that the carries and the properties of the data type denoted by B are not modified by the new declarations of A, but they are preserved within the data type denoted by A. This requirement is formalized by the two constraints:

- *no junk*: A does not add elements to the carries of the sorts of B;
- *no confusion*: A does not contain axioms which make true properties not holding in B. An atom without variables built using the symbols of the signature of B holds in B iff it holds in A.

The idea of that is that we use the specification determined by B as it is, without modifying the data type which defines.

The first constraints (no junk) cannot be verified automatically. Indeed in general it is allowed that in SA we declare constants and operations of a sort defined in SB, and so we cannot guarantee that the terms obtained with these new constants and operations are equivalent to some old terms.

Ex. ?? does not satisfy the constraint no junk: indeed the term `First(Empty)` of sort `nat`, which corresponds to read the first element of an empty stack, denotes an element completely new for such sort, different from all naturals defined in NAT; thus the carrier of `nat` has been modified. For fixing this problem we have to add new axioms identifying these new term with another already existent. For example we may add:

```
ax  First(Empty) =  0
```

Also the second constraint (no confusion) cannot be verified automatically, since the confusion may be caused by axioms apparently about other sorts.

Example 2.2
Consider the environment defined by the binds NAT and STACK_OF_NATURAL of the Ex. ?? and by the following ones.

```
STACK_OF_NATURAL' =
design  **  adding one axiom
use  STACK_OF_NATURAL
var  s s': stack
ax  s = s' if  First(s) = First(s')
```

The added axiom say that two stacks are equal if they have equal the first element. This axiom produces confusion in the sort `nat`. Indeed consider `s0 = Put(0,Put(0,Empty))` and `s1 = Put(0,Put(Succ 0,Empty))`.

From the last axiom, we get $s_0 = s_1$, since `First(s0) = First(s1) = 0`. On the other hand if `s0 = s1` then `First(Get(s0)) = First(Get(s1))` and thus `0 = Succ 0`. That means that one is equal to zero! Proceeding in this way we can deduce that all the elements of sort `nat` are equal.

The result is that a wrong axiom non only ruins the stacks, but also the naturals, which apparently were not involved. Errors of this kind can be very difficult to find and thus we need to be very careful. **End example**

2.3 Rename

The rename is a construct which allows to reuse modules written previously simply by changing the name to some of the components of the signature (sorts, constants, operations and predicates).

By renaming a specification `S` we can change the symbols directly defined in `S`, and also that imported in `S` from other specification. The kind of the specification instead remains the same, and so we can rename design, requirement, param and draft specifications getting specifications of the same kind.

Example 2.3 Partial and total orders
A partial order structure (POSet, Partially Ordered Set) consists of a set, and of a partial order (i.e. two elements may be noncomparable).

```
POSET =
requirement
sort  poset
var  x  y  z: poset
pr  _ <= _: poset  poset
ax  x <= x    **  reflexivity
ax  if x <= y and y <= x  then  x = y  **  antisimmetry
ax  if x <= y and y <= z  then  x <= z  **  transitivity
```

For getting the specification of the totally ordered sets (`TOSET`) we add the axiom
`x <= y or y <= x`;
at this point it is reasonable to change the name to sort, to avoid confusion (see **??**).

```
TOSET =
requirement
use  rename sort poset to toset in POSET
var  x  y: toset
ax  x <= y or y <= x
```

End example

Syntax

$Spec$ ::= `rename` Map_1 ... Map_n `in` $Spec$
Map ::= `sort` $Sort_1$ `to` $Sort_2$ | $CnOpPrExpr$ `to` $MixfixRepr$
$CnOpPrExpr$ ::= `cn` $MixfixRepr$ | `cn` $MixfixRepr$: $Sort$ |
`op` $MixfixRepr$ | `op` $MixfixRepr$: $Sort_1$ ... $Sort_n$ `->` $Sort$ |
`pr` $MixfixRepr$ | `pr` $MixfixRepr$: $Sort_1$ ... $Sort_n$

The sorts, the constants, the operations and the predicates on the left of the `to` must be part of the signature of the renamed specification.

The arities of the symbols on the left of the `to` can be omitted, if there is no ambiguity. The arities of the symbols on the right are deducible by the renaming of the arities of the symbols on the left, and thus there is no need to declare them explicitly.

Renaming a sort corresponds really to declare a new sort; it is the same thing for the names of sorts of the subsection ??.

2.4 Parametric specifications

METAL supports a powerful tool for the reuse of the modules: indeed it allows to parameterize the modules over other modules. The parameterization is a construct of high level which allows to reuse specifications written before, in a controlled and safe way.

The METAL approach to parametric modules it is based on the following points.

- The formal parameters are identifiers representing specifications.
- A specification (of kind param) is associated with each formal parameter, it represents the requirements that the corresponding actual parameter must satisfy for making the correct instantiations. These requirements are both about the structure of the signature and the properties of the interpretation of the constants, of the operations and of the predicates.
- The instantiations are obtained by associating with each formal parameter a specification, called actual parameter, which must satisfy the relative requirements.
- The association of an actual parameter with a formal one it is not trivial, since we have to define how the actual parameter satisfies the requirements of the formal parameter. This it is done by associating with each of the components (sorts, constants, operations and predicates) of the specification defined in the requirement part of the formal parameter a corresponding element of the actual parameter.
- The sorts defined in a parametric specification can have a syntax parameterized on the sorts appearing in the requirement part of the formal parameters; for example we can define the parametric lists, having sort `list(elem)`, and then instantiate them with the natural numbers as elements, getting the sort `list(nat)`.

  In this way the sorts defined in different instantiations of the same parametric specification have automatically assigned different names; this is very useful to avoid the collisions among the names of the sorts.

Example 2.4 Parametric lists
In this example we specify the data type lists parameterized on the elements. This parametric specification can be instantiated producing lists of natural numbers, lists of streams, lists of lists of naturals, or lists of whatever other data type (which has at least a sort).

```
LIST =
generic ELEM : param  sort elem end in
requirement
use  ELEM
sort  list(elem)
var  e: elem
cn  <>: list(elem)
op  _ & _: elem  list(elem) -> list(elem)
op  _: elem -> list(elem)
ax  list(elem)(e) = e & <>
end
```

Now we want to instantiate `LIST`, binding the formal parameter `ELEM` with the actual parameter `NAT`. We have to define a correspondence between the signature of the requirements of `ELEM` and that of `NAT`, by associating the sort `elem` with the sort `nat`.

```
LIST_NAT =
LIST(NAT sort elem to nat)
```

Now the specification `LIST_NAT` has the sorts `nat` and `list(nat)`. Other examples of instantiations are:

```
LL_NAT =
LIST(LIST(NAT sort elem to nat) sort elem to list(nat))

LLL_NAT =
LIST(LL_NAT  sort elem to list(list(nat)))
```

End example

The requirements on actual parameters allow to prove general properties of the parametric module, holding on all instantiations, whenever the actual parameter satisfies the requirements. The same specification can be used as actual parameter from different points of view, depending on the association among the signatures.

Syntax

Spec ::= `generic` $UpUpIdent_1$: $Spec_1$ ... $UpUpIdent_n$: $Spec_n$ $Spec_{n+1}$ |
$UpUpIdent(Instant_n,\ \ldots,\ Instant_n)$ |
`param` $BasicDec_1$ ... $BasicDec_n$ `end`
Instant ::= *Spec* Map_1 ... Map_n | *Spec*
Sort ::= $LowIdent(Sort_1,\ \ldots,\ Sort_n)$

2.4.1 Instantiation of the parameters

To instantiate a parametric specification we associate with each formal parameter an actual parameter. Each actual parameter must satisfy the requirements put in the declaration of the corresponding formal parameter. These requirements are about signature and axioms. The signature of the requirement of the formal parameter put some structural constraints on the actual parameter, while the axioms put semantic constraints. Let `X: P` be a formal parameter and `A` the corresponding actual parameter.

Structural constraints. They are satisfied by defining a correspondence between the symbols of the signature of `P` and the symbols of the signature of `A`. Taken a symbol of the signature of `P` we gives the corresponding element of the signature of `A`.

The symbols which have to be associated are that visible in the signature of `P`, originated by a specification of kind param. Such symbols are usually declared directly in `P`, but they can be also imported from another specification of type param. Notice that if `P` imports the symbols from a specification of different kind, these symbols must be such and that in the actual parameter. More precisely:

- Each sort, constant, operation and predicate of `P`, originated in a specification not of kind param must be present in `A` with the same arity.

- A sort $\mathtt{s}^A$ of `A` must correspond with each sort $\mathtt{s}^P$ of the signature of `P` originated in a specification of kind param.

- A constant $\mathtt{Cn}^A\mathtt{:}\ \mathtt{s}^A$ of `A`, where $\mathtt{s}^A$ corresponds to $\mathtt{s}^P$, must correspond to each constant $\mathtt{Cn}^P\ \mathtt{:}\ \mathtt{s}^P$ of the signature of `P` originated in a specification of kind param.

- An operation $\mathtt{Op}^A\ \mathtt{:s1}^A \ldots \mathtt{sn}^A\ \mathtt{s}^A$ of `A`, where the sorts $\mathtt{s1}^A, \ldots, \mathtt{sn}^A, \mathtt{s}^A$ correspond respectively to the sorts $\mathtt{s1}^P, \ldots, \mathtt{sn}^P, \mathtt{s}^P$, must correspond to each operation $\mathtt{Op}^P\ \mathtt{:s1}^P \ldots \mathtt{sn}^P\ \mathtt{s}^P$ of the signature of `P` originated by a specification of kind param.

- A predicate $\mathtt{Pr}^A\ \mathtt{:s1}^A \ldots \mathtt{sn}^A$ of `A`, where the sorts $\mathtt{s1}^A, \ldots, \mathtt{sn}^A$ correspond respectively to the sorts $\mathtt{s1}^P, \ldots, \mathtt{sn}^P$, must correspond to each predicate $\mathtt{Pr}^P\ \mathtt{:s1}^P \ldots \mathtt{sn}^P$ of the signature of `P` originated in a specification of kind param.

- If we omit the association for a symbol of the signature of `P`, and there exists a symbol in the signature of `A` with the same name and corresponding arity, these symbols are associated by default.

The variables are local and so we do not need to associate them.

Semantic constraints. The associations defined between the signature of `P` and that of `A` allows to translate the axioms of `P` into axioms of `A`. The semantic constraints are satisfied if the axioms of `P` translated in the signature of `A` hold also in `A`.

It is not possible to verify automatically the semantic constraints.

Example 2.5 Ordered lists
In general a specification can be used as actual parameter in several different ways. For example the naturals can be considered an ordered set both w.r.t. the relation "$\leq$" and the relation "$\geq$".

The following parametric specification gives the requirements of the parametric lists with an ordering operation. For defining such operation we need a total order predicate on the list elements.

```
TOSET =
param
sort  toset
var  x  u  z: toset
pr  _ <= _: toset  toset
ax  x <= x  **  reflexivity
ax  if  x <= u and y <= x  then  x = y  **  antisimmetry
ax  if x <= y and y <= z  then  x <= z  **  transitivity
ax  x <= y or y <= x  **  totality

ORDERED_LIST =
generic ELEM: TOSET
requirement
use  LIST(ELEM sort toset to elem)
var  h1 h2: toset
var  t1 t2  l: list(toset)

**  list permutations
pr  _ Permute _: list(toset) list(toset)
ax  (h1 h2 t) Permute (h2  h1  t)
ax  (h  t1) Permute (h  t2) if  t1 Permute t2
ax  <> Permute <>

**  checks if a list is ordered
pr  _ Is Ordered: list(toset)
ax  h1 h2 t1  Is Ordered if  h1 <= h2 and h2 t1 Is Ordered}
ax  h1 <>  Is Ordered
ax  <> Is Ordered

**  ordering operation
op  Order: list(toset)}{list(toset)
ax  l Permute Order(l) and Order(l) Is Ordered
end
```

We want to instantiate the parameter `ELEM` with the natural numbers: they can be considered an ordered set both in increasing and decreasing order.

```
NAT =
design
sort  nat
var  x  y: nat
cn  0: nat
op  Succ _: nat -> nat
pr  _ <= _: nat  nat
pr  _ => _: nat  nat
ax  0 <= x
ax  x <= Succ x
ax  x => y if y <= x
end

INCREASING_LIST_OF_NATURAL =
ORDERED_LIST(NAT sort toset to nat)
**  we can omit  ``pr _ <= _: toset toset to _ <= _''
**  since it is taken by default

DECREASING_LIST_OF_NATURAL =
ORDERED_LIST(NAT sort toset to nat pr _ <= _ to _ => _)
**  we can omit the arity of ``_ <= _'' since there is no overloading
```

End example

Example 2.6 Lists with a measure on the elements
If on the elements there is defined an operation returning a natural number (the measure of the elements), we can enrich the lists with an operation for computing the sum of such numbers.

```
MEASURE =
param
use  NAT
**  NAT is not of type param and thus we do not need to associate its symbols
**  with symbols of the actual parameter; they must appear in the actual parameter,
**  otherwise how we can have the corresponding of Measure?
sort  elem
op  Measure: elem -> nat

LIST_WITH_MEASURE =
generic ELEM: MEASURE
design
use  LIST(ELEM)
**  we can omit ``sort elem to elem'' since such mapping is assumed by default
var  h: elem
var  t: list(elem)
op  Measure: list(elem) -> nat
```

```
ax  Measure(h &  t) = Measure(h) + Measure(t)
ax  Measure(<>) = 0
end

L_M_NAT =
LIST_WITH_MEASURE(NAT sort elem to nat op Measure to Succ _)

LL_M_NAT =
LIST_WITH_MEASURE(L_M_NAT sort elem to list(nat))
```

End example

2.5 Controlling the exported symbols

The need of hiding some of the symbols of a specification usually arose in two cases.

- Specifying a data type, we can use auxiliary operations (or sorts or constants or predicates), for being able to write the axioms, which are not meaningful from the global point of view. These elements of the signature have to be hidden, so that cannot be visible outside.
- Importing a specification, we can have the intention of using only some of the symbols of the signature.

METAL allows to put a filter for hiding the symbols which must not be visible outside. The filter can be positive (export) or negative (hide).

Example 2.7 Export filter
Consider the following specification of the parametric data type "binary trees" with balancing test.

```
TREE =
generic ELEM: param sort elem end
export
sort elem
sort tree(elem)}
cn  <>
op  Node
pr  _ Is Balanced
**  the arities are omitted since there is no ambiguity
from
design
use  ELEM
var  k: elem
sort  tree(elem)
var  t1  t2: tree(elem)
```

```
cn  <>: tree(elem)
op  Node: elem  tree(elem)  tree(elem) -> tree(elem)

use  NAT
op  Height: tree(elem) -> nat
ax  Height(<>) = 0
ax  Height(Node(k,t1,t2)) = Max(Height(t1),Height(t2)) + 1

pr  _ Is Balanced: tree(elem)}
ax  <> Is Balanced
ax  Node(k,t1,t2) Is Balanced if
        t1 Is Balanced and t2 Is Balanced and
        (Height(t1) = Height(t2) or
         Height(t1) = Succ Height(t2) or
         Succ Height(t1) = Height(t2))
end
```

The specification `TREE` defines parametric binary trees, with a predicate `Is Balanced` checking if a tree is balanced; for defining this predicate we have to use an auxiliary function `Height` computing the height of a tree, and which uses the naturals, and thus we have to import also the naturals, which apparently have no relationship with the trees. But both the natural numbers and the auxiliary function `Height` are not visible outside, since the export filter blocks them. **End example**

Example 2.8 Hide filter
Consider the specification `ORDERED_LIST` of the Ex. ??: for defining the properties of the operation `Order` we need to introduce the binary predicates on lists `Permute` and `<=`. We hide these predicates by writing

```
ORDERED_LIST' =
hide
pr  _ Permute _
pr  _ <= _ -> list(elem)  list(elem)
in
ORDERED_LIST
```

Notice that the predicate `_ <= _ : list(elem) list(elem)` has been denoted with the whole arity, otherwise it could be confused with `_ <= _ : elem elem`. **End example**

Syntax

Spec ::= `hide` *Filter*$_1$ ... *Filter*$_n$ `in` *Spec* | `export` *Filter*$_1$ ... *Filter*$_n$ `from` *Spec*
Filter ::= `sort` *Sort* | *CnOpPrExpr*

2.6 Draft specifications

The semantics of the specifications of kind draft does not associate any data type with them. Sometime these specifications are useful for splitting a large specification in smaller pieces, which by themselves have not interesting models. Typically they are used for solving problems as the collision of the names of sorts, or for anticipating some declarations useful for defining recursive data types in a modular way.

In METAL it is allowed to introduce in physically distinct declarations sorts with the same name, and constants, operations and predicates with the same name and the same arity, but these symbols cannot be put in the same signature, since the may cause ambiguity not solvable in any way. The only exceptions are the objects declared in the draft modules, which are identified with any other object of the same type identified by the same name, without giving problems.

Syntax

Spec ::= `draft` $BasicDec_1$ ... $BasicDec_n$ `end`

2.6.1 Collision of the names of the sorts

To declare in different specifications two sorts with the same name but with different meanings makes confusion and makes the specification unreadable. In general it is correct to proceed as follows:

- for denoting the same carrier in two different specifications we declare once the sort in a third specification, which will be used by the first two ones;
- for denoting different carries we use sorts with different names.

METAL allows of declaring in two different places two different sorts with the same name, but these two sorts never have to be shared by the same signature. The only exceptions are the sorts declared in the draft specifications, which are identified with whichever sort having the same name.

Example 2.9 Collision of sorts
In the following example we has a collision among sorts:

```
A1 = design sort  a  . . .  end

A2 = design sort  a   . . .   end  **  until here Ok

B =
design
use  A1, A2
**  now there is a collision among distinct sorts with the same name
cn  C: a
end
```

This problem can be solved in several ways. If we want that the two sorts being the same, i.e. corresponding to the same carrier we can write

```
A = draft sort  a  end

A1 = design use A . . . end

A2 = design use A  . . . end

B =
design
use  A1, A2
cn  C: a
end
```

In this case the sort `a` has been declared once in the specification `A`, and thus the specification `B` is correct. Alternatively we can write

```
A1 = draft sort a  . . . end

A2 = draft sort  a  . . . end   **  until here Ok

B =
design
use  A1, A2
**  now there is not  a collision since being A1 draft the sort a of A1 is
**  identified with the sort a of A2
cn  C: a
end
```

If instead we want that the two sorts are distinct, i.e. which correspond to distinct carries we must write for example:

```
A1 = design sort a1 end

A2 = design sort a2 end

B =
design
use  A1, A2
cn  C: a1
cn  C: a2
end
```

End example

The situation is the same for constants, operations and predicates. Two of these symbols, having exactly the same name and the same arity, but coming from distinct declarations, cannot stay in the same signature, except that at least one of the two is declared in a draft specification.

2.6.2 Recursive data types

There exist and they are often used data types which are defined in terms of themselves; to give a specification split in modules of these data types can be a problem, due to the impossibility of having cycles in the dependence among the modules. The point can be solved predeclaring some symbols in a draft specification.

Example 2.10 Multilists of naturals
We want to model the data type of the lists containing elements which in turns can be lists, or natural numbers. We want to specify such a data type as follows:

```
M =
design
sort  m
use  NAT
op  _ : nat -> m
use  LIST(M sort elem to m)    **  error
op  [ _ ]: list(m) -> m
```

Unfortunately this cannot be done, since it is in contrast with the rule for which the dependencies among modules cannot have loops and cycles. This point can be solved by using an auxiliary draft specification.

```
M' = draft sort m end

M =
\design
use  M', NAT
op  _ : nat -> m
use  LIST(M' sort elem to m)
op  [ _ ]: list(m) -> m
end
```

End example

Example 2.11 Trees
We consider the parametric trees, with labelled nodes and arcs. Each node can have whatever number of sons, which in turn are trees. A set of trees is a forest, and thus the set of the sons of each node can be seen as a forest.

```
LABEL_NODES =  **  requirements on the labels of the nodes
param
```

```
sort  e_nodes
end

LABEL_ARCS =  **  requirements on the labels of the arcs
param
sort  e_arcs
end
```

We predeclare the sort tree in a draft specification.

```
TREE =
generic E_NODES: LABEL_NODES
        E_ARCS: LABEL_ARCS
draft
use  E_NODES, E_ARCS
sort  tree(e_nodes,e_arcs)
end

FOREST =
generic E_NODES: LABEL_NODES
        E_ARCS: LABEL_ARCS
draft
use  TREE(E_NODES, E_ARCS)
sort  forest
var  x  y  z: forest
cn  <>: forest
op  _ : tree(e_nodes,e_arcs) -> forest
op  _ + _: forest  forest -> forest
ax  x + (y + z) = (x + y) + z
ax  x + y = y + x
ax  x + <> = x
end

TREE =
generic E_NODES: LABEL_NODES
        E_ARCS: LABEL_ARCS
design
use  FOREST(E_NODES, E_ARCS)
**  finally we can give an initial semantic to the carrier of the sort tree,
**  redeclaring it in a design specification
sort  tree(e_nodes,e_arcs)
op  Tree: e_nodes  forest -> tree(e_nodes,e_arcs)
end
```

The draft specification `TREE` is used to predeclare the sort `tree(e_nodes,e_arcs)` for defining the forests, and thus the trees. **End example**

3 Dynamic specifications

3.1 Dynamic signatures

Syntax

A dynamic signature is a sequence of declarations of sorts, dynamic sorts, constants, operations and predicates.

SignatureDec ::= `dsort` *Sort*: *MixfixRepr*
Sort ::= `lab_`*Sort*

To a dynamic sort declaration of the form `dsort` *Sort*: *MixfixRepr* are implicitly added the following declarations:

`sort lab_`*Sort* and

`pr` *MixfixRepr*: *Sort* `lab_`*Sort* *Sort*.

It is prohibited to declare twice the same dynamic sort; instead it is possible to redeclare dynamic a sort already declared as usual static sort; moreover if the sort `lab_`*Sort* or the predicate *MixfixRepr* have been already declared no error is signalled and the new ones coincide with the old ones.

3.2 Design and requirement dynamic specifications

Design dynamic specification are just as the usual METAL design specifications; but now the axioms may involve the transition predicate and thus define the activity of the dynamic elements.

Example 3.1 Specification of a buffer
We give the specifiation of a simple buffer containing natural numbers.

```
BUFFER =
design
use  NAT  **  predefined data type specification (see the appendix)
dsorts  buffer: _ -- _ --> _
var  b: buffer
var  n: nat
cn  Empty: buffer
op  C: nat -> buffer
**  the sort lab_buffer and the predicate _ -- _ --> _ are implicitly
**  declared
cn  INT: lab_buffer op  SEND: nat -> lab_buffer
op  REC: nat -> lab_buffer

ax  Empty -- REC(n) --> C(n)
ax  C(n) -- SEND(n) --> Empty
ax  b -- INT --> b
end
```

End example

The axioms of requirement dynamic specifications are extended with combinators of the temporal logic for expressing requirements on the activity of the dynamic elements.

Syntax

Axiom ::= *Term* `in each case` *PathAxiom* | *Term* `in one case` *PathAxiom*
PathAxiom ::= (*PathAxiom*) |
`not` *PathAxiom* |
$PathAxiom_1$ `and` $PathAxiom_2$ |
$PathAxiom_1$ `or` $PathAxiom_2$ |
`if` $PathAxiom_1$ `then` $PathAxiom_2$ |
$PathAxiom_1$ `if` $PathAxiom_2$ |
$PathAxiom_1$ `iff` $PathAxiom_2$ |
$PathAxiom_1$ `until` $PathAxiom_2$ |
$PathAxiom_1$ `wuntil` $PathAxiom_2$ |
`always` *PathAxiom* |
`now and always` *PathAxiom* |
`eventually` *PathAxiom* |
`now or eventually` *PathAxiom* |
`after` *PathAxiom* |
`[` *LowIdent* . *Axiom*`]` |
`[` *Axiom* `]` |
`[` *Term* `]` |
`<` *LowIdent* . *Axiom* `>` |
`<` *Axiom* `>` |
`<` *Term* `>`

A Predefined data type specifications

The specification of some data types used very frequently are predefined in METAL; some of them are basic data types, as the natural numbers and the integers; while others are parametric data types, as lists and sets. Some of these specifications are expressed by using the standard features of METAL, other use particular syntactic features.

Truth values The predefined specification `BOOL` models the data type "truth values".

```
BOOL =
design
sort  bool
var  b: bool
cn  True: bool
cn  False: bool
```

```
op  Not _ : bool -> bool
ax  Not True  =  False
ax  Not False  =  True

op  _ And _ : bool   bool -> bool
ax  True And b  =  b
ax  False And b  =  False

op  _ Or _ : bool  bool -> bool
ax  True Or b  =  True
ax  False Or b  =  b
end
```

Natural numbers The predefined specification `NAT` models the data type of the natural numbers, with carrier $\mathbb{N}$, and the operations of sum, product and the comparison predicates.

The constants `0`, `1`, `2`, ..., `1967`, ... of sort `nat` are predefined built-in.

```
NAT =
design
use  BOOL

sort  nat
var  n  m: nat

**  built-in
**  cn 0 : nat
**  cn 1 : nat
**  . . .
op  Succ: nat -> nat
**  built-in
**  ax Succ(0) =  1
**  ax Succ(1) =  2
**  . . .

op  _  +  _: nat  nat -> nat
ax  0 +  n  =  n
ax  Succ(n) +  m  =  Succ(n  +  m)

op  _  *  _: nat  nat -> nat
ax  0 *  n  =  0
ax  Succ(n) *  m  =  (n  *  m) +  m

pr  _ <= _: nat  nat
ax  n <= n
```

```
ax  if  n <= m  then  n <= Succ(m)

pr  _  <  _ -> nat  nat
ax  n  <  m if Succ(n) <= m

pr  _  =>  _: nat  nat
ax  n => m  if  m <= n

pr  _ > _: nat  nat
ax  n > m  if  m  <  n

pr  _ =/= _ : nat  nat
ax  0 =/= Succ(m)
ax  Succ(n) =/= 0
ax  if  n =/= m  then  Succ(n) =/= Succ(m)
end
```

Integers numbers The specification `INT` models the data type "integer numbers", with carrier $\mathbb{Z}$, and various operations and predicates.

The integers use the naturals, taking advantage of the built-in representation due to an operation of implicit embedding. This implicit operation is not understood by the parser if it is applied to the outer level of one of the two terms of an equality. For this reason it is advisable to use the unary operation `+` when writing positive integers constants.

```
INT =
design
use  NAT
var  n: nat

sort  int
var  i  j: int

op  _ : nat -> int

op  Succ: int -> int
ax  Succ(int(n)) =  int(Succ(n))

op  Pred: int -> int
ax  Succ(Pred(i)) =  i
ax  Pred(Succ(i)) =  i

op  + _ : int -> int
ax  + i  =  i

op  - _ : int -> int
```

```
ax  -0 =  + 0
ax  - Succ(i) =  Pred(- i)
ax  - Pred(i) =  Succ(- i)

op  _  +  _ : int   int -> int
ax  0 + i  =  i
ax  Succ(i) + j  =  Succ(i + j)
ax  Pred(i) + j  =  Pred(i + j)

op  _ - _ : int  int -> int
ax  i - j  =  i + (- j)

op  _  *  _ : int  int -> int
ax  0 * i  =   +  0
ax  Succ(i) * j  =  (i * j) + j
ax  Pred(i) * j  =  (i * j)- j

pr  _ <=  _ : int  int
ax  i <= i
ax  if  i <= j  then  i <= Succ(j)

pr  _  <   _ : int  int
ax  i  <  j  if  Succ(i) <= j

pr  _  =>  _ : int  int
ax  i  =>  j  if  j <= i

pr  _ > _ : int  int
ax  i > j  if  j  <  i

pr  _ =/= _ : int  int
ax  if  i > j  then  i =/= j
ax  if  i  <  j  then  i =/= j
end
```

Identifiers The specification IDENT gives a convenient wayfor handling identifiers, as atomic unit.

An infinite number of constants are built-in, that we write putting between single primes whatever non empty sequence of alphabetic lower case and upper case letters.

```
IDENT =
design
sort  ident
**  cn '*': ident
**  where * is a nonempty sequence of upper case or lower case letters
```

```
end
```

Parameters requirements Some common requirements for the specification parameters are predefined.

```
ELEMENT = param sort elem  end

ERROR_ELEMENT =
param
use  ELEMENT
cn  ErrorElement -> elem
end

DIFFERENT_ELEMENT =
param
use  ELEMENT
pr  _ =/= _: elem  elem
var  e1  e2: elem
ax  e1 =/= e2 iff not e1 =  e2
end

POSET =
param
use  ELEMENT
pr  _ <= _: elem  elem
var  x  y  z: elem
%
ax  x <= x
ax  if  x <= y and y <= x  then  x  =  y
ax  if  x <= y and y <= z  then  x <= z
end

TOSET =
param
use  POSET
var  x  y: elem
ax  x <= y or y <= x
end
```

Lists The following parametric specification `LIST` is predefined. It can be instantiated for modelling the set of the finite sequences of elements of any data type. We require that such data type should have a special value, to be returned by the operation `Head` when applied to the empty list.

```
LIST =
```

```
generic ELEM: ERROR_ELEMENT
draft
use  NAT, ELEM
var  h: elem

sort  list(elem)
var  t  l: list(elem)

cn  Empty: list(elem)
op  _  ^   _ : elem  list(elem) -> list(elem)

op  _ : elem -> list(elem)
ax  h  =  h ^ Empty

op  _ &  _ : list(elem)  list(elem) -> list(elem)
ax  Empty &  l  =  l
ax  (h ^ t) &  l  =  h ^ (t & l)

op  Head: list(elem) -> elem
ax  Head(Empty) =  ErrorElement
ax  Head(h ^ t) =  h

op  Tail: list(elem) -> elem
ax  Tail(Empty) =  Empty
ax  Tail(h ^ t) =  t

pr  _ Is Empty: list(elem)
ax  Empty Is Empty

pr  _ Is Not Empty: list(elem)
ax  h ^ t Is Not Empty

op  Length: list(elem) -> nat
ax  Length(Empty) =  0
ax  Length(h ^ t) =  Succ(Length(t))
end
```

Finite sets The following parametric specification `SET` is predefined. It can be instantiated for modelling the data type finite sets of elements of another data type.

The operations _ ; _ and _ ; _ ; _ are useful for simply expressing sets with 2 or 3 elements.

```
SET =
generic ELEM: ELEMENT
draft
```

```
use  ELEM
sort  set(elem)
var  x  y  z: elem
var  s  s'  s'': set(elem)

cn  {}: set(elem)
op  { _ }: elem -> set(elem)

op  _ U _: set(elem)  set(elem) -> set(elem)
ax  s U {}  =  s
ax  s U (s' U s'') =  (s U s') U s''
ax  s U s' =  s' U s
ax  s U s  =  s

pr  _ In _: elem  set(elem)
ax  x In  { x } U s

op  {_ ; _}: elem  elem -> set(elem)
ax  {x;y} =  {x} U {y}

op  {_ ; _ ; _}: elem   elem  elem -> set(elem)
ax  {x;y;z} =  {x} U {y;z}
end
```

Record The data type record (or cartesian product of various components enriched by operations for selecting and modifying the components) are very common and frequently used; thus in METAL there is a special construct, with ad hoc syntax, for specifying record types, putting in evidence:

- the sort of the record type;
- the fields (the components of the product), each one with the relative name, and the specification defining the relative values; clearly, since in general a specification has different sorts we have to indicate also which sort of such specification defines the values of the component.

This construct declares automatically the constructor of the record as tuple of elements, and the operations of access and modification of the fields. The specification expression

```
record rec  **  sort of the record
F1: s1 of S1
. . .
Fn: sn of Sn
**  fields, with name, sort and specification of the relative values
end
```

is equivalent to

```
design
use  S1, S2, . . . , Sn
sort  rec
var  v1  v1': s1
. . .
var  vn  vn': sn
op  < _ ; _ ;  . . .  ; _ >: s1  . . .   sn -> rec
**  constructor of the record

**  operations for field access
op  F1: rec -> s1
ax  F1(< v1 ;  . . . ; vn >) =  v1
. . .
**  operations for field modification
op  _ [_ / F1 ]: rec  s1 -> rec
ax  < v1 ;  . . .  ; vn > [v1'/ F1]  =  < v1' ;  . . . ; vn >
. . .
end
```

Syntax

Spec ::= `record` *Sort* $Field_1$ ... $Field_n$ `end`
Field ::= *UpIdent*: *Sort* `of` *Spec*

Given the field `F: s of S` the sort `s` must be in the signature of the specification `S`.

Enumeration types Frequently we have to specify a data type with only one carrier, a finite number of constants and nothing other; these are the enumeration types. METAL gives a built-in construct for rapidly defining the enumeration types. The specification expression

`enum s : Id1 Id2 . . . Idn end`

is equivalent to

```
design
sort  s
cn  Id1: s
 . . .
cn  Idn : s
pr  _ =/= _ : s   s
ax  Idi =/= Idj   for i, j =1,  . . . , n, i =/= j
end
end
```

Syntax

Spec ::= `enum` *LowIdent* : *UpIdent*$_1$ ... *UpIdent*$_n$ `end`

B The syntax of METAL

B.1 Lessical level

In this section we defines the set of the terminal symbols used in the pattern rules, which are particular sequences of basic characters.

Basic characters: the set *Character* includes all the characters used in METAL:

```
1 2 3 4 5 6 7 8 9 0
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
~  ` ! @ # $ %  ^ & *  ( ) - _ +  =  |  / \ { }
[ ]   :   ;
'  "  <   >  .  ?
```

There are also the space, the tab character and the end of line.

Separators. The space character, the tab character and the new line character, are considered separators. The separators are not part of the terminals; they are used to separate the terminal symbols among them, when they are needed, and for giving a better form to the code.

Another separator is the comment. The comment starts with `**` in each context and ends with the end of line character.

Terminal symbols: the set *Terminal* is a subset of *Character**.

Upper case letters: the set *UpCase* consists of the characters

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Lower case letters: the set *LowCase* consists of the characters

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Digit: the set *Digit* consists of the characters

```
1 2 3 4 5 6 7 8 9 0
```

Special characters: the set *SpecialCharacter* consists of the characters

```
~ ‘ ! @ # $ % ^ & * - +  = \ /
|  { }  [ ]   :   ;  "  ’  <   > . ?
```

notice that the characters “_”, “(”, “)” and “,” have been excluded.

Identifier tails: whatever sequences (also empty) of upper and lower case letters, digits, primes and underscores can be the tail of an identifier.
$IdentTail = (\,UpCase \cup LowCase \cup Digit \cup \{\ \_,\ ‘\ ,\ ’,\ "\ \})^*$.

Lower case identifiers: they either start with a lower case letter, followed by an *IdentTail*. The key words *KeyWord* are excluded.
$LowIdent = (\{lt \mid l \in LowCase \ \text{ and } \ t \in IdentTail\} - KeyWord)$

Upper case identifiers: they either starts with a upper case letter, followed by an *IdentTail*, or are sequences of decimal digits.
$UpIdent = \{ut \mid u \in UpCase \ \text{ and } \ t \in IdentTail\} \cup Digit^*$

Completely upper case identifiers: they are the subset of upper case identifiers which do not include lower case letters.
$UpUpIdent = \{u \in UpIdent \mid u \text{ does not include lower case letters}\}$

Symbols: they are sequences of the special characters *SpecialCharacter*; the reserved symbols *KeySymbol* are excluded.
$Symbol = SpecialCharacter^* \setminus KeySymbol$.

Reserved symbols: the set *KeySymbol* includes the terminal symbols
“(”, “)”, “ : ”, “ = ”, “,”, “->” and “**”.

Reserved words: the set *KeyWord* includes the terminal symbols

```
and   ax   cn
design  draft  enum  end  exists  export forall
from generic  hide if  iff in not of op or
param pr  record rename requirement  sort then to  use var
```

Terminal: $Terminal = UpIdent \cup LowIdent \cup KeyWord \cup KeySymbol \cup Symbol$

Note.

- The identifiers available to the users are distinguished in three categories: upper case, lower case and symbols. The upper case identifiers have in turn a subclass, that of the completely upper case identifiers.
- The upper case identifiers include the numbers.
- The characters `_`, `(`, `)` and `,` are not part of the symbols, and do not need to be preceded and followed by an explicit separator. Thus for example `_+_` is equivalent to `_ + _` and `F(x,y)` is equivalent to `F ( x , y )`. Instead `AA_BB` is different from `AA _ BB` since the underscore can be part of the identifiers.

B.2 Concrete syntax

In this section we gives the grammar by means of pattern rules. The pattern rules are a variant of the well-known formalism BNF commonly used for defining context free grammars.

The nonterminal symbols *UpIdent*, *UpUpIdent*, *LowIdent* and *Symbol* are defined in the section **??**.

The initial symbol of the grammar is *BindSequence*, which derives in sequences of associations name-specification.

$BindSequence ::= Bind_1 \ \dots \ Bind_n$
$Bind ::= UpUpIdent \ \texttt{=} \ Spec$

Specification expressions. They include the base specifications, the instantiations of the generic specifications, the export/hide filters and the rename of specifications, and further derived constructs.

$Spec ::= \texttt{design} \ BasicDec_1 \ \dots \ BasicDec_n \ \texttt{end} \ |$
$\quad \texttt{requirement} \ BasicDec_1 \ \dots \ BasicDec_n \ \texttt{end} \ |$
$\quad UpUpIdent \ |$
$\quad \texttt{rename} \ Map_1 \ \dots \ Map_n \ \texttt{in} \ Spec \ |$
$\quad \texttt{generic} \ UpUpIdent_1\texttt{:} \ Spec_1 \ \dots \ UpUpIdent_n\texttt{:} \ Spec_n \ Spec_{n+1} \ |$
$\quad \texttt{param} \ BasicDec_1 \ \dots \ BasicDec_n \ \texttt{end} \ |$
$\quad UpUpIdent\texttt{(}Instant_n\texttt{,} \ \dots\texttt{,} \ Instant_n\texttt{)} \ |$
$\quad \texttt{hide} \ Filter_1 \ \dots \ Filter_n \ \texttt{in} \ Spec \ |$
$\quad \texttt{export} \ Filter_1 \ \dots \ Filter_n \ \texttt{from} \ Spec \ |$
$\quad \texttt{draft} \ BasicDec_1 \ \dots \ BasicDec_n \ \texttt{end} \ |$
$\quad \texttt{record} \ Sort \ Field_1 \ \dots \ Field_n \ \texttt{end} \ |$
$\quad \texttt{enum} \ LowIdent\texttt{:} \ UpIdent_1 \ \dots \ UpIdent_n \ \texttt{end}$
$Map ::= \texttt{sort} \ Sort \ \texttt{to} \ Sort \ | \ CnOpPrExpr \ \texttt{to} \ MixfixRepr$
$CnOpPrExpr ::= \texttt{cn} \ MixfixRepr \ | \ \texttt{cn} \ MixfixRepr\texttt{:} \ Sort \ |$
$\quad \texttt{op} \ MixfixRepr \ | \ \texttt{op} \ MixfixRepr\texttt{:} \ Sort_1 \ \dots \ Sort_n \ \texttt{->} \ Sort \ |$

$\quad\quad$ `pr` *MixfixRepr* | `pr` *MixfixRepr* : *Sort*$_1$ ... *Sort*$_n$
Instant ::= *Spec* *Map*$_1$ ... *Map*$_n$ | *Spec*
Filter ::= `sort` *Sort* | *CnOpPrExpr*
Field ::= *UpIdent* : *Sort* `of` *Spec*

Basic declarations.

BasicDec ::= *SignatureDec* | *VarDec* | *AxiomDec*
VarDec ::= `var` *LowIdent*$_1$ ... *LowIdent*$_n$: *Sort*

Signature declarations.

SignatureDec ::= `sort` *Sort*$_1$ ... *Sort*$_n$ |
`cn` *MixfixRepr* : *Sort* |
`op` *MixfixRepr* : *Sort*$_1$ ... *Sort*$_n$ `->` *Sort* |
`pr` *MixfixRepr* : *Sort*$_1$ ... *Sort*$_n$
Sort ::= *LowIdent* | *LowIdent*`(`*Sort*$_1$`,` ... `,` *Sort*$_n$`)`
MixfixRepr ::= *MixfixReprElem*$_1$ ... *MixfixReprElem*$_n$
MixfixReprElem ::= *Symbol* | *UpIdent* | _

Axioms: they are obtained by the atoms (equalities or built by predicates) by applying the logic combinators and quantifiers.

For giving a more readable grammar, the precedences among the combinators are given a part, as contextual constraints.

AxiomDec ::= `ax` *Axiom*
Axiom ::= *Atom* | `(`*Axiom*`)` |
`not` *Axiom* |
Axiom$_1$`and` *Axiom*$_2$ | *Axiom*$_1$`or` *Axiom*$_2$ |
`if` *Axiom*$_1$`then` *Axiom*$_2$ | *Axiom*$_1$`if` *Axiom*$_2$ |
Axiom$_1$`iff` *Axiom*$_2$
Atom ::= *UpIdent*`(`*Term*$_1$`,` ... `,` *Term*$_n$`)` | *Term*$_L$ `=` *Term*$_R$ | *Mixfix*
Mixfix ::= *MixfixElem*$_1$ ... *MixfixElem*$_n$
MixfixElem ::= *UpIdent* | *Symbol* | *Term* | `(`*Mixfix*`)`

Terms

Term ::= *UpIdent* |
UpIdent`(`*Term*$_1$`,` ... `,` *Term*$_n$`)` |
LowIdent |
Mixfix |
Sort`(`*Term*`)`

B.3 Pattern rule

Here we introduce the pattern rules, which are a variant of the well-known formalism B.N.F. commonly used for defining the syntax of programming languages.

The pattern rule are more concise, and allow to easily express context constraints.

The terminal symbols are identifiers written in typewritten font or sequences of special characters.

For example `design`, `:` and `->` are terminal symbols.

The nonterminal symbols are written with the italic style and starts with upper case letters.

For example *Bind*, *BasicDec*, *Sort* are nonterminals.

Writings as

$NonTerm$ ::= $RightBb$ | $RightAa$

denote that from the nonterminal *NonTerm* we can derive *RightBb* or *RightAa*.

Subscripts and superscripts on the nonterminal are used for enumerating distinct occurrences of the same nonterminal within the same rule.

For example, consider

$Atom$ ::= $Term_L$ `=` $Term_R$

here the sequences derived from $Term_L$ and $Term_R$ can be different.

Lists and sequences are conveniently expressed with suspensive dots.

For example

$Spec$ ::= `design` $BasicDec_1$ ... $BasicDec_n$ `end`

means that from *Spec* we can derive a sequence of a certain number $n \geq 1$ of *BasicDec*, when n is not specified.

Contents