OBSCURE: A specification environment for abstract data types

by

Thomas Lehmann  Jacques Loeckx

A 87/06

Saarbrücken, August 1987
OBSCURE: A specification environment for abstract data types

Thomas Lehmann       Jacques Loeckx

July 31, 1987

Fachbereich Informatik
Universität des Saarlandes
D - 6600 Saarbrücken (West Germany)
loekx%sbsvax.uucp@germany.csnet
thomas%sbsvax.uucp@germany.csnet

Table of Contents

1. Introduction
2. Basic notions
3. The command language of OBSCURE
4. A protocol
5. Some final comments
References
Figures
Appendix
1 Introduction

The idea that abstract data types may support the development of correct programs is now well-accepted. Meanwhile several methods have been proposed for the specification of abstract data types: operational specifications ([Ho 72], [Sh 81], [Li 81], [NY 83], [LG 86]), algebraic specifications ([GTW 78], [GHM 78], [TWW 82], [BW 82], [Eh 82], [EM 85]) and constructive specifications ([Ca 80], [Ki 84], [Lo 87]). While operational specifications are embedded in an imperative language, algebraic specifications are more abstract in that they make use of first-order formulas, usually equalities or Horn-clauses. Algorithmic specifications offer a similar degree of abstraction but differ by their constructive nature.

The design of non-trivial specifications is practicable only if it is performed modularly. To this end specifications are embedded in a specification language. Essentially, such a language allows the construction of specifications out of more elementary ones. In the case of operational specifications the specification language is foreordained to be the embedding imperative language. For algebraic specifications several specification languages have recently been proposed: CLEAR ([Sa 84],[BG 80]), ACT-ONE ([EM 85]), OBJ2 ([FGJM 85]), PLUSS ([Gd 84],[BGM 87]), ASL ([Wi 86]), ASF ([BHK 87]).

Even with the use of a specification language the design of non-trivial specifications with pencil and paper is tedious and error-prone. A solution consists in embedding the specification language into an adequate environment. Such an environment supports the interactive design of specifications as well as the (interactive or automatic) verification of their properties. Some more or less elaborate environments have been described or announced in the literature: OBJ2 ([FGJM 85]), an environment for a subset of the specification language PLUSS called ASSPEGIQUE ([BCV 85]), the environment RAP ([Hu 87]), an environment for the specification language ACT-ONE called the ACT-System.

The specification tool to be presented in this paper is called OBSCURE. It consists of a specification language together with an environment for it. The specification language is a simple language similar to Bergstra's term language ([BHK 86]). The environment is a program consisting of a design unit and a verification unit. The design unit allows the interactive design of specifications. More precisely, with the help of a command language the user induces the design unit to stepwise generate specifications. The verification unit allows to prove properties of these specifications. The main features by which OBSCURE differs from the specification languages and environments described in the literature are now briefly discussed.

First, the specification language of OBSCURE has been designed as a language to be used in an environment, not as a language to be used with pencil and paper. As a result the specification language has a very simple syntax and semantics at the expense of more elaborate context conditions.
These context conditions put no burden on the user as they are checked automatically and on-line (i.e. at each command) by the design unit. Second, the specification language is independent from the specification method used. It even allows the use of different (algebraic and/or constructive) specification methods within the same specification. This is possible because OBSCURE distinguishes between the constructs inherent to the specification method — such as "data constraints" in CLEAR — and those inherent to putting specifications together. Third, OBSCURE distinguishes between the specification language and the command language of the design unit. This is reflected by the fact that procedure mechanisms (i.e. "parameterized specifications") and user-friendly macros are part of the command language, not of the specification language. Hence parameterized specifications are not specifications but rather constitute a tool to construct specifications. Next, apart from the classical operations OBSCURE provides means to explicitly construct subalgebras and quotient algebras. Finally, OBSCURE directly ties the design of a specification to its verification. In particular, the design unit automatically generates formulas expressing, for instance, certain persistency conditions and transmits them to verification to the verification unit.

The goal of the present paper is to give a very brief and informal overview of the command language of the design unit and to illustrate its use. A precise and formal description of the specification language may be found in [LL 87].

Section 2 briefly recalls some basic notions. Section 3 describes the command language of the design unit. Section 4 presents a commented protocol. Section 5 contains final comments.

2 Basic notions

2.1 Signatures

Sorts, operations and signatures are defined as usual.

Formally, a sort is an identifier. An operation is a \((k + 1)\)-tuple, \(k \geq 0\),

\[
\pi : s_1 \times \ldots \times s_k \rightarrow s_{k+1}
\]

where \(s_1, \ldots, s_{k+1}\) are sorts. It is called \(S\)-sorted if \(S\) is a set of sorts with \(s_1, \ldots, s_{k+1}\) among its elements. A list of sorts and operations is a \((k + 1)\)-tuple

\[
(s_1, \ldots, s_k; o_1, \ldots, o_l)
\]

with \(s_1, \ldots, s_k\) sorts and \(o_1, \ldots, o_l\) operations, \(k \geq 0, l \geq 0\).

An example is

\[
(set, integer; e : set, Insert : set \times integer \rightarrow set)
\]
An \textbf{(algebra) signature} is a pair $(S, \Omega)$ where $S$ is a set of sorts and $\Omega$ a set of operations such that each operation of $\Omega$ is $S$-sorted.

For a given signature, say $\Sigma$, one may introduce the notions of a $(\Sigma-)$ \textit{term} and of a $(\Sigma-)$ \textit{formula} in the classical way. For instance, for the signature

\[
\{\text{set, integer}, \{\varepsilon \rightarrow \text{set}, \text{Insert : set} \times \text{integer} \rightarrow \text{set}\}\}
\]

$\text{Insert}(\varepsilon, 0)$ is a term and $\neg \text{Insert}(\varepsilon, 0) = \varepsilon$ is a (first-order) formula. Precise definitions are in [LL 87].

In what follows the signatures considered satisfy the following condition: for any two different operations with the same operation name $n$

\[
\begin{align*}
n : s_1 \times \ldots \times s_k & \rightarrow s_{k+1}, \quad k \geq 0 \\
\text{and} \quad n : t_1 \times \ldots \times t_l & \rightarrow t_{l+1}, \quad l \geq 0
\end{align*}
\]

either $k \neq l$ or there exists $i, 1 \leq i \leq k$, such that $s_i \neq t_i$. This condition allows \textit{operation overloading}, i.e. in terms and formulas any operation, say $n : s_1 \times \ldots \times s_{k+1}$, may be replaced by its name $n$ without creating ambiguities. (Actually, we already did so in the examples of a term and a formula given above). Hence a signature may contain the operations

\[
\text{Insert : set} \times \text{integer} \rightarrow \text{set}
\]

and

\[
\text{Insert : list} \times \text{integer} \rightarrow \text{list}
\]

but not

\[
\varepsilon : \rightarrow \text{set}
\]

and

\[
\varepsilon : \rightarrow \text{list}
\]

\subsection{2.2 Algebras}

Let $\Sigma = (S, \Omega)$ be a signature. A $(\Sigma-)$ \textit{algebra} is a function that maps

\begin{enumerate}
\item [\text{i}] each sort $s$ of $S$ into a set $A(s)$ called the \textit{carrier set} of sort $s$;
\item [\text{ii}] each operation $n : s_1 \times \ldots \times s_{k+1}, k \geq 0$, of $\Omega$ into a (possibly partial) function
\end{enumerate}

\[
A(n : s_1 \times \ldots \times s_k \rightarrow s_{k+1}) : A(s_1) \times \ldots \times A(s_k) \rightarrow A(s_{k+1}).
\]

Again, for a given $\Sigma$-algebra $A$ the \textit{value} of a $\Sigma$-term and the \textit{validity} of a $\Sigma$-formula are defined as usual, viz. as an extension of the function $A$ (see [LL 87]). For instance, in the "standard" interpretation the value of the term $\text{Insert}(\varepsilon, 0)$ is the set consisting of the number zero, and the formula $\neg \text{Insert}(\varepsilon, 0) = \varepsilon$ is valid.

The class of all $\Sigma$-algebras is denoted $\text{Alg}_\Sigma$. 3
2.3 Modules

Modules constitute the semantics of specifications. They produce “exported” algebras by extending “imported” ones (cf. [BHK 86], [EW 85], [DP 87], [EFPP 86], [EW 86]).

More formally, a module signature is a pair \((\Sigma_i, \Sigma_e)\) of signatures called the imported and exported signature respectively. A sort or operation that occurs in both the exported and imported signature is called an inherited one. Figure 1 shows a graphical representation of a module signature that will be used in Section 3.

A module for the module signature \((\Sigma_i, \Sigma_e)\) is a (possibly partial) function

\[ m : \text{Alg}_{\Sigma_i} \rightarrow \text{Alg}_{\Sigma_e} \]

satisfying the following persistency condition:

for every algebra \(A\) of the domain of \(m\):
for every inherited sort or operation \(r\):

\[ m(A)(r) = A(r). \]

Informally, the persistency condition expresses that the meaning of the inherited sorts and operations remains unchanged.

This definition of a module allows to cope with specifications that define a single model. In order to be able to also handle loose specifications it is sufficient to define a module as a relation \(m \subseteq \text{Alg}_{\Sigma_i} \times \text{Alg}_{\Sigma_e}\) rather than a function \(m : \text{Alg}_{\Sigma_i} \rightarrow \text{Alg}_{\Sigma_e}\) (see [LL 87]).

2.4 Atomic specifications

Essentially, OBSCUR E allows to construct specifications out of atomic specifications (cf. [BHK 86]). An atomic specification is drawn up according to one of the numerous specification methods. The description of its syntax and semantics is outside the realm of OBSCUR E.

3 The command language of OBSCUR E

As indicated above a specification in the specification language of OBSCUR E has the form of a term and is interpreted as a module.

The command language of (the design unit of) OBSCUR E is essentially a postfix version of this specification language together with macros and a procedure mechanism. Its goal is to generate specifications in the specification language. To this end the design unit makes use of a stack (for transforming postfix into infix) and of a library of procedures. At each command the design unit automatically checks the context conditions. Moreover it generates
formulas, the validity of which guarantees the semantic consistency of the specification. In this way the design unit makes sure that the specifications generated are syntactically and semantically correct.

A non-exhaustive list of the commands is in Figure 2. The semantics of these commands are graphically illustrated in Figure 3 and are now shortly commented. Most of these comments are illustrated in the protocol of Section 4. Hence the reader may very well skip the present Section and return to it if required. For a definition of the syntax, the context conditions and the semantics of the command language the reader is referred to [LL 87]. This paper also contains a proof that the context conditions together with the formulas generated by the specification unit suffice to guarantee the consistency of the definition of the semantics.

The design unit is started with an empty stack. The library is empty or may contain "given" procedures.

The command create am endcreate writes the atomic specification am on the stack.

The commands add and compose act like binary postfix operators: they replace the top two elements of the stack by the result. The commands forget iso through quot s by w act like unary operators.

The commands add and compose put specifications together (see Figure 3). The command add constructs the "union" of the specifications m₁ and m₂. The use of compose corresponds to a top-down design: being contained by the top element of the stack the "refinement" m₁ has been designed after m₂. Figures 3(a) and 3(b) suggest that both commands are subject to severe context conditions. For instance, the exported signatures of the operands m₁ and m₂ of add may only have inherited sorts and operations in common. Being too stringent for most "practical" cases these context conditions are relaxed in the macro-command refine to be discussed below.

The command forget (Figure 3(c)) drops exported sorts and operations. It allows to get rid of auxiliary ("hidden") sorts and operations. More importantly it allows to eliminate those operations which would fail to satisfy the semantic constraints induced by subsequent sub or quot commands (see below).

The command

\[ \text{e-rename } iso1 \text{ into } iso2 \]

(Figure 3(d)) renames exported sorts and operations. More precisely, the (exported) sorts and operations of the list iso2 are simultaneously substituted for those of iso1. The imported sorts and operations remain unchanged. Note that the renaming of a sort implies the renaming of its occurrences in the operations. For instance, the renaming of the sort el into integer entails the substitution of the operation Insert : set × el → set by Insert : set × integer →
The command may be used to avoid name clashes, i.e. to comply with the context conditions of, for instance, a subsequent add command.

The command

\[ \text{i-renam} \text{e } \text{iso1 } \text{into } \text{iso2} \]

is similar but applies to the imported signature. As a fundamental difference the renaming extends to the exported sorts and operations that are inherited: in Figure 3(c), for instance, both the imported and the exported occurrence of \( a \) is renamed into \( f \). Moreover, the command may identify names by giving them the same name: in Figure 3(c) both names \( b \) and \( c \) are renamed into \( g \). The utility of this command will become clear in the discussion of the parameter passing mechanism.

In the command

\[ \text{i-axiom } w \]

the formula \( w \) is a formula of the imported signature. It expresses a semantic constraint on the domain of the module. More precisely, the module defined by the specification \( m \) yielded by the command is identical with the (module defined by the) specification \( m_1 \) to which the command is applied, except that its domain is restricted to those algebras \( A \) which satisfy the formula \( w \). From a user’s point of view the command requires the verification of a semantic constraint, i.e. a proof that the “intended” imported algebra belongs to the domain of the module \( m \). To this end the design unit transmits the formula \( w \) to the verification unit (see [LL 87] for more precision). The main use of the command is to express semantic constraints on the (formal) parameters of a procedure, — as will become clear below.

In the command

\[ \text{e-axiom } w \]

the formula \( w \) is a formula of the exported signature. The domain of the module \( m \) is now restricted to those algebras \( A \) for which the algebra \( m(A) \) satisfies the formula \( w \). The command allows in particular to express that the specification satisfies a given property. For instance, having specified the sort set the user may want to check that

\[ \neg(x \in \text{Delete}(s,x)) = \text{true} \]

i.e. that an element no longer belongs to a set from which it has been deleted.

In the command

\[ \text{sub } s \text{ by } w \]
the formula \( w \) is a formula of the exported signature and contains free occurrences of a single variable, namely a variable of sort \( s \). The exported sort \( s \) may not be inherited. The module \( m \) yielded by the command differs from the module \( m_1 \) to which it is applied in the following way: let \( A \) be an arbitrary algebra of the domain of the module \( m \); the carrier set \( m(A)(s) \), i.e. the carrier set of sort \( s \) of the algebra \( m(A) \), is a subset of the carrier set \( m_1(A)(s) \), namely the subset whose elements satisfy the formula \( w \). In other words, \( m(A) \) is the subalgebra of the algebra \( m_1(A) \) induced by the formula \( w \) ([EM 85], [Lo 87], [LL 87]). For instance, if the carrier set \( m_1(A)(s) \) consists of multisets, and if

\[
Nodup : s \rightarrow \text{bool}
\]

is an operation expressing that a multiset contains no duplicates then the command

\[
\text{sub } s \text{ by } (Nodup(x) = \text{true})
\]

applied to the module \( m_1 \) yields a module \( m \) such that the carrier set \( m(A)(s) \) consists of sets (see Section 4 for a detailed example). Clearly, this construction of a subalgebra is well-defined only if some closure conditions are satisfied ([EM 85], [Lo 87], [LL 87]). In the case of the example these closure conditions express that the operations of the algebra \( m_1(A) \) map duplicate-free arguments into duplicate-free values. These closure conditions constitute implicit semantic constraints on the module \( m_1 \). The design unit automatically generates formulas expressing these constraints and transmits them to the verification unit.

The command

\[
\text{quot } s \text{ by } w
\]

is similar but the formula \( w \) now has free occurrences of two variables of the sort \( s \). Hence \( w \) defines a relation rather than a subset. It is assumed that this relation is an equivalence relation (again a semantic constraint to be verified!). Let \( m, m_1 \) and \( A \) be as above. The carrier set \( m(A)(s) \) now consists of the equivalence classes induced by the equivalence relation in the carrier set \( m_1(A)(s) \). In other words, \( m(A) \) is the quotient algebra induced by the formula \( w \) ([EM 85], [Lo 87], [LL 87]). For instance, if the carrier set \( m_1(A)(s) \) consists of lists, and if

\[
Eq : s \times s \rightarrow \text{bool}
\]

is an operation expressing that two lists are identical except for the order of occurrence of their elements, then the command

\[
\text{quot } s \text{ by } (Eq(u,v) = \text{true})
\]
applied to the module $m_1$ yields a module $m$ such that the carrier set $m(A)(s)$ consists of multisets. Again, the construction of a quotient algebra is well-defined only if the equivalence relation is a congruence relation, i.e. if each operation of the algebra $m_1(A)$ maps equivalent arguments into equivalent values. Again, the design unit automatically generates formulas expressing the different semantic constraints.

The command `is proc n(lo)` constitutes a procedure declaration. The name of the procedure is $n$, the procedure body is the specification at the top of the stack and the sorts and operations of $lo$ are the formal parameters. The effect of the command is to add the declaration to the library and to pop the stack. The sorts and operations constituting the formal parameters have to be imported ones. For instance, if the specification contained by the top element of the stack specifies "sets of elements", $set$ being an exported sort and $element$ an imported one, a possible procedure declaration is

```
is proc SET (element).
```

From the user's point of view a procedure body is designed as if it were a "normal" specification; when completed this specification is turned into a procedure by the `is proc n(lo)` command.

The command `call n(lo)` constitutes a procedure call: $n$ is the name of a procedure from the library and the sorts and operations of $lo$ are the actual parameters. The effect of the command is to write (a copy of) the procedure body on the stack. The parameter passing is performed by an implicit `i-rename` command. For instance, the command

```
call SET (integer)
```

yields a specification specifying sets of integers. This specification is obtained by applying the command

```
i-rename el into integer
```

to the body of the procedure SET. The sort `integer` is an imported sort of this specification — as was the sort `element` in the specification constituted by the procedure body. Note that one of the context conditions requires that a procedure name is declared before it is called. This excludes, in particular, recursive calls.

The command identity $lo$ creates an "empty" specification: the module it defines is the identity function for the imported signature (Figure 3(f)).

The command `refine` combines the effects of the commands `add` and `compose` (Figure 3(g)). It automatically "supplies" the missing sorts and operations. As a result the context conditions are less stringent than for `add` and `compose."
The command `extend first` permutes the two top elements of the stack and then acts like `refine`. While `compose` and `refine` are intended for top-down design of specifications, `extend` is useful for bottom-up design.

In the command

`copy iso1 as iso2`

the sorts and operations are from the exported signature (Figure 3(a)). A possible use of the command is the following. Consider the notation of Figure 3(a) with $d$ denoting a sort. Apply the command `sub d by w` on the specification $m$. According to this command the carrier set of sort $d$ becomes a subset of the original one. But this original carrier set is not lost: thanks to the `copy` command it is still available under the name $f$.

The command `is proc n(iso1')(iso2')` is similar to a usual procedure declaration but contains in addition the parameters $iso2'$, called `result parameters`. The sorts and operations of $iso2'$ have to be exported ones. At the command `call n(iso1')(iso2')` the parameter passing of the result parameters is performed by an implicit `e-rename` command. For instance, a procedure declaration could be

`is proc SET (element)(set)`

and a corresponding procedure call

`call SET (integer)(setofint).`

The specification yielded by this call again has the sort `integer` among its imported sorts but the sort `setofint` (instead of `set`) among its exported ones. As they automatically rename exported sorts and operations, result parameters may be used to avoid name clashes when a procedure is called more than once.

The design unit which is being implemented in our laboratory offers several additional facilities. The user is, for instance, allowed to designate at any moment an operation by its name only. If — due to overloading — this name does not univocally determine the arity of the operation the design unit will ask the user for it. A further facility allows to delete the actual parameters from a procedure call whenever they are identical with the formal ones. Hence the command `call n` writes the body of the procedure `n` on the stack without any renaming.

4 A protocol

The goal of this Section is to illustrate the use of `OBSOURE` as a tool for the design of specifications. The three examples illustrate in particular the use of procedures, the construction of quotient algebras and the introduction of
semantic constraints on parameters. Together the three examples constitute
the protocol of a single session with the design unit of OBSCURE.

When presenting concrete examples one has to fix the specification method
and the logic. We have chosen the initial algebra specification method and
first-order logic.

The reader of this Section may experience that it is tedious to keep track
of the current module signature. He should remember that the design unit
automatically updates the signatures and checks the context conditions at
each command.

The protocol is in the Appendix. Note that “n?.” introduces text written
by the user, “*****” introduces text displayed by the design unit and “$$$”
introduces comments that are not part of the protocol.

5 Some final comments

OBSCURE allows top-down, bottom-up or “mixed” development of specifi-
cations. It does not require to start specifying “from scratch” because the
initial contents of the library of procedures has not to be empty.

The commands subset and quotient are essential in the algorithmic speci-
cification method (see [Lo 87]). While they may be dispensed with in an al-
gebraic specification method they may be useful to make specifications more
modular and explicit.

OBSCURE is more than a specification language in that it also allows to
express properties of the specifications. In particular, the command e-axiom
is similar to a “Hoare-like” assertion in an imperative programming language.

A prototype of an environment for OBSCURE is being developed in our
laboratory. The design unit is nearing completion. It offers several facilities
beyond those mentioned in this paper. A manual of the editor is available
([FH 87]).
References


[Sh 81] Shaw, M., "ALPHARD, Form and Content", *Springer-Verlag* (1981)


FIGURE 1: A graphical representation of (the signature of) an algebra module. The symbols $a, b, c, d$ stand for sorts and/or operations: $a$ and $b$ are imported, $a, c$ and $d$ are exported, $a$ is inherited.
Syntactic categories

<table>
<thead>
<tr>
<th>am</th>
<th>atomic specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>sort</td>
</tr>
<tr>
<td>iso</td>
<td>list of sorts and operations</td>
</tr>
<tr>
<td>w</td>
<td>formula</td>
</tr>
<tr>
<td>n</td>
<td>(procedure) name</td>
</tr>
</tbody>
</table>

List of commands

(i) Elementary commands

<table>
<thead>
<tr>
<th>create am</th>
<th>endcreate</th>
<th>atomic specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td></td>
<td>horizontal composition</td>
</tr>
<tr>
<td>compose</td>
<td></td>
<td>vertical composition</td>
</tr>
<tr>
<td>forget iso</td>
<td></td>
<td>dropping sorts and operations</td>
</tr>
<tr>
<td>e-rename iso1 into iso2</td>
<td>renaming exported sorts and operations</td>
<td></td>
</tr>
<tr>
<td>i-rename iso1 into iso2</td>
<td>renaming imported sorts and operations</td>
<td></td>
</tr>
<tr>
<td>i-axioms w</td>
<td></td>
<td>semantic constraint on import</td>
</tr>
<tr>
<td>e-axioms w</td>
<td></td>
<td>semantic constraint on export</td>
</tr>
<tr>
<td>sub s by w</td>
<td></td>
<td>subalgebra</td>
</tr>
<tr>
<td>quotient s by w</td>
<td></td>
<td>quotient algebra</td>
</tr>
</tbody>
</table>

(ii) Procedure commands

<table>
<thead>
<tr>
<th>is proc n (iso)</th>
<th>procedure declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>call n (iso)</td>
<td>procedure call</td>
</tr>
</tbody>
</table>

(iii) Macro-commands

<table>
<thead>
<tr>
<th>identity iso</th>
<th>empty specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>refine</td>
<td>refinement step</td>
</tr>
<tr>
<td>extend</td>
<td>extension step</td>
</tr>
<tr>
<td>copy iso1 as iso2</td>
<td>copy with new names</td>
</tr>
<tr>
<td>is proc n (iso1) (iso2)</td>
<td>procedure declaration with result parameters</td>
</tr>
<tr>
<td>call n (iso1) (iso2)</td>
<td>procedure call with result parameters</td>
</tr>
</tbody>
</table>

FIGURE 2: The commands of the design unit of OBSCURE. The left columns of the tables contain the commands, the right columns contain comments. The list of macro-commands is not exhaustive.
(a) add

(b) compose

(c) forget (a, c)

(d) s-rename (a, c) into (c, f)
FIGURE 3 Graphical illustration of the semantics of the main commands. In this illustration $m_1$ and $m_2$ are operands (i.e. the specifications contained by the top element and the second element of the stack respectively), $m$ is the result (i.e. the specification yielded by the command). Each of the symbols $a, b, c, \ldots$ stands for a sort or an operation.
Appendix

First example

1. create

\[
\begin{align*}
\text{produces} & \quad \text{sorts} & \quad \text{list} \\
\text{operations} & \quad \epsilon & \rightarrow \text{list} \\
& \quad \ldots : \text{list} \times \text{el} & \rightarrow \text{list} \\
& \quad \in_\ldots : \text{el} \times \text{list} & \rightarrow \text{bool} \\
\text{equations} & \quad (\epsilon \in \epsilon) = \text{false} \\
& \quad (\epsilon \in (\ldots, \epsilon')) = \text{if } (\epsilon = \epsilon') \text{ then true else } (\epsilon \in \ldots) \text{ I1} \\
\text{needs} & \quad \text{sorts} & \quad \text{el, bool} \\
& \quad \text{operations} & \quad \ldots = \ldots : \text{el} \times \text{el} & \rightarrow \text{bool} \\
\end{align*}
\]

endcreate

*** \( \Sigma_i = (\epsilon, \text{bool}; \ldots = \ldots : \text{el} \times \text{el} \rightarrow \text{bool}) \)

\( \Sigma_e = \Sigma_i \cup (\text{list}; \epsilon \rightarrow \text{list}, \ldots : \text{list} \times \text{el} \rightarrow \text{list}, \ldots \in \ldots : \text{el} \times \text{list} \rightarrow \text{bool}) \)

$$$ This atomic specification introduces "list of elements" and is drawn up according to the initial algebra specification method. Its imported signature is described under the heading needs. Its exported signature additionally contains the sorts and operations listed under the heading produces. Hence all sorts and operations of the imported signature are inherited. According to the initial algebra specification method an atomic specification defines a free extension of the imported algebra. The operation "." expresses "appending", "\in\" expresses "element of".

$$$ To be precise we should have added the constants true, false and the ternary operation if-then-else to the imported signature. We have omitted them to simplify the presentation.

$$$ The stack of the design unit now contains the atomic specification as its single element.

2. is proc LIST(\epsilon; \ldots = \ldots : \text{el} \times \text{el} \rightarrow \text{bool}) (\text{list}; \epsilon \rightarrow \text{list})

$$$ The specification in the stack is turned into a procedure with name LIST. It contains two (formal) argument parameters, viz. \epsilon and \ldots = \ldots : \text{el} \times \text{el} \rightarrow \text{bool}, and two (formal) result parameters, viz. \text{list} and \epsilon \rightarrow \text{list}. This procedure is added to the library.

$$$ The stack of the design unit is now empty again.

3. call LIST (int; \ldots = \ldots : \text{int} \times \text{int} \rightarrow \text{bool}) (\text{ilist}; i\epsilon \rightarrow \text{ilist})

*** \( \Sigma_i = (\text{int, bool}; \ldots = \ldots : \text{int} \times \text{int} \rightarrow \text{bool}) \)
\[ \Sigma_e = \Sigma_i \cup (i\text{-}e :\rightarrow \text{ilist}, \ldots : \text{ilist} \times \text{int} \rightarrow \text{ilist}, \_ \in \_ : \text{int} \times \text{ilist} \rightarrow \text{bool}) \]

$$$ The specification resulting from the call is pushed onto the stack. It is obtained by applying two renamings on (a copy of) the procedure body: an \(i\)-rename for the argument parameters (i.e. the parameters contained by the first pair of parentheses) and an \(e\)-rename for the result parameters (i.e. the parameters contained by the second pair of parentheses).

4 ?- call \text{LIST} \ (\text{string}; \_ = \_ : \text{string} \times \text{string} \rightarrow \text{bool}) \ (\text{slist}; s\text{-}e :\rightarrow \text{slist})

*** \[ \Sigma_e = (\text{string}, \text{bool}; \_ = \_ : \text{string} \times \text{string} \rightarrow \text{bool}) \]

\[ \Sigma_e = \Sigma_i \cup (\text{slist}; s\text{-}e :\rightarrow \text{slist}, \ldots : \text{slist} \times \text{string} \rightarrow \text{slist}, \_ \in \_ : \text{string} \times \text{slist} \rightarrow \text{bool}) \]

$$$ Yet another call of the procedure \text{LIST}. The stack now contains two specifications.

5 ?- \text{add}

*** \[ \Sigma_i = (\text{int}, \text{string}, \text{bool}; \_ = \_ : \text{int} \times \text{int} \rightarrow \text{bool}, \_ = \_ : \text{string} \times \text{string} \rightarrow \text{bool}) \]

\[ \Sigma_e = \Sigma_i \cup (\text{ilist}, \text{slist}; i\text{-}e :\rightarrow \text{ilist}, \ldots : \text{ilist} \times \text{int} \rightarrow \text{ilist}, \_ \in \_ : \text{int} \times \text{ilist} \rightarrow \text{bool}, s\text{-}e :\rightarrow \text{slist}, \ldots : \text{slist} \times \text{string} \rightarrow \text{slist}, \_ \in \_ : \text{string} \times \text{slist} \rightarrow \text{bool}) \]

$$$ Note the overloading of the operation names "." and "\(\in\)". Note also that without the renaming performed by the result parameters of the procedure calls overloading such as that of the operation name "\(\in\)" would fail to satisfy the overloading condition of Section 2.1.

6 ?- \text{is proc LISTS-OF-INT-AND-LISTS-OF-STRINGS}

$$$ The specification is stored for later use as a parameterless procedure. The stack is empty again.

Second example

$$$ The goal is to deduce a specification of multisets from the specification \text{LIST}. To this end the procedure \text{LIST} is enriched. According to the top-down philosophy this enrichment is "created" and then "refined" by (the procedure body of) \text{LIST}.
7 ?- create

produces operations $Eq : list \times list \rightarrow bool$
$Subset : list \times list \rightarrow bool$
$Delete : list \times el \rightarrow list$

equations $Eq(l_1, l_2) = \begin{cases} \text{if } Subset(l_1, l_2) \text{ then } Subset(l_2, l_1) \\ \text{else false fi} \end{cases}$
$Subset((l_1, e), l_2) = \begin{cases} \text{true} & \text{if } (e \in l_2) \text{ then } Subset(l_1, Delete(l_2, e)) \\ \text{else false fi} \end{cases}$
$Delete(e, l) = e$
$Delete((l, e'), e) = \begin{cases} e = e' \text{ then } l \\ \text{else Delete}(l, e).e' \text{ fi} \end{cases}$

needs sorts $list, el, bool$
operations $e : \rightarrow list$
$\_ : list \times el \rightarrow list$
$\_ \_ : el \times list \rightarrow bool$

endcreate

*** $\Sigma_i = (list, el, bool; e : \rightarrow list, \_ : list \times el \rightarrow list, \_ \_ : el \times list \rightarrow bool)$

$\Sigma_e = \Sigma_i \cup (Eq : list \times list \rightarrow bool, Subset : list \times list \rightarrow bool, Delete : list \times el \rightarrow list)$

$$Delete(l, e) \text{ deletes the rightmost occurrence of the element } e \text{ in the list } l.$$
$Eq(l_1, l_2) \text{ checks whether the lists } l_1 \text{ and } l_2 \text{ contain the same number of occurrences of each element.}$

8 ?- forget (Subset, Delete)

*** $\Sigma_i$ is unchanged

$\Sigma_e = \Sigma_i \cup (Eq : list \times list \rightarrow bool)$

$$\text{The user drops these operations because he feels he no longer needs them. Hence these operations act like "hidden functions".}$$

9 ?- call LIST

*** $\Sigma_i = (el, bool; \_ \_ : el \times el \rightarrow bool)$

$\Sigma_e = \Sigma_i \cup (list; e : \rightarrow list, \_ : list \times el \rightarrow list, \_ \_ : el \times list \rightarrow bool)$

$$\text{A copy of the procedure body of the procedure LIST is pushed onto the stack without any renaming.}$$

10 ?- refine

*** $\Sigma_i = (el, bool; \_ \_ : el \times el \rightarrow bool)$
\[\Sigma_e = (\text{list, el, bool}; \ e :\rightarrow \text{list}, \ \ldots : \text{list} \times \text{el} \rightarrow \text{list}, \ \_ : \_ : \text{el} \times \text{list} \rightarrow \text{bool}, \ Eq : \text{list} \times \text{list} \rightarrow \text{bool})\]

$$\$$ The enrichment was performed in a top-down way. The user could also have performed it bottom-up by first calling LIST, then “creating” the new operations and finally putting both specifications together with the command extend.

11 ?- quotient list by Eq(l1, l2) = true

*** \(\Sigma_i, \Sigma_e\) are unchanged.

$$\$$ \(l_1, l_2\) are variables of sort list.

$$\$$ The design unit automatically generates formulas expressing the semantic constraints implied by the construction of the quotient algebra and transmits them for verification to the verification unit.

$$\$$ The original meaning of list, \(\epsilon\), ";" and "\(\in\)" is now "overwritten". While this original meaning is no longer available on the stack it is of course still attached to the procedure LIST contained in the library.

12 ?- e-rename (list; Eq : \text{list} \times \text{list} \rightarrow \text{bool} ) \text{ into (multiset; } = : \text{multiset} \times \text{multiset} \rightarrow \text{bool})

*** \(\Sigma_i = (\text{el, bool}; \ = : \_ : \text{el} \times \text{el} \rightarrow \text{bool})\)

\[\Sigma_e = (\text{multiset, el, bool}; \ e :\rightarrow \text{multiset}, \ \ldots : \text{multiset} \times \text{el} \rightarrow \text{multiset}, \ \_ : \_ : \text{el} \times \text{multiset} \rightarrow \text{bool}, \ = : \_ : \text{multiset} \times \text{multiset} \rightarrow \text{bool})\]

$$\$$ The new names are more suggestive. Note that Eq : \text{list} \times \text{list} \rightarrow \text{bool} or after renaming \(= : \text{multiset} \times \text{multiset} \rightarrow \text{bool}\) expresses the standard equality between multisets — as directly results from the definition of the quotient operation.

13 ?- is proc MULTISET (el; \ = : \_ : \text{el} \times \text{el} \rightarrow \text{bool})

Third example

$$\$$ The goal is to deduce a specification of ordered lists from the specification LIST. First "parameterized axioms" are introduced.

14 ?- identity (el, bool; \ \leq \ : \_ : \text{el} \times \text{el} \rightarrow \text{bool})

*** \(\Sigma_i = \Sigma_e = (el, bool; \ \leq \ : \_ : \text{el} \times \text{el} \rightarrow \text{bool})\)

$$\$$ The command creates an "empty specification", i.e. a specification defining the identity module.
15 ?- i-axioms

variables u : el, v : el, w : el
(u ≤ u) = true
(u ≤ v) = true ∧ (v ≤ u) = true ∨ u = v
(u ≤ v) = true ∧ (v ≤ w) = true ∨ (u ≤ w) = true
(u ≤ v) = true ∨ (v ≤ u) = true

endaxioms

*** Σi, Σe are unchanged.

$$$ The axioms express that “≤” is a total order. It is implicitly assumed that each axiom is universally quantified and that the different (universally quantified) axioms are connected by “∧” to yield a single first-order formula.

16 ?- is proc TOTAL-ORDER (el; - ≤ - : el × el → bool)

$$$ The procedure TOTAL-ORDER constitutes a “parameterized axiom”.

$$$ Now the procedure LIST is extended.

17 ?- create

produces operations Isord : list → bool
- ⊙ - : list × el → bool

equations Isord(e) = true
Isord(e, e) = true
Isord((l, e), e') = if e ≤ e' then Isord(l, e)
else false fi

ε ⊙ e = e.e
(l, e) ⊙ e' = if e ≤ e' then ((l, e), e')
else ((l, ε') . e) fi

needs sorts list, el, bool
operations ε := list
- .. : list × el → list
- ≤ - : el × el → bool

endcreate

*** Σl = (list, el, bool; ε := list, .. : list × el → list, - ≤ - : el × el → bool)

Σe = Σl ∪ (Isord : list → bool, - ⊙ - : list × el → list)

$$$ “⊙” expresses “ordered appending”.

18 ?- e-axioms

variables l : list, ε : el
Isord(l, ε) = false ∨ ¬(l = ε)

endaxioms
*** $\Sigma_i, \Sigma_e$ are unchanged.

$$\text{For one reason or another the user wants to check that his specification satisfies the property expressed by this axiom. The design unit transmits the formula for verification to the verification unit.}$$

19 ?- call LIST

*** $\Sigma_i = (el, bool; _ = _ : el \times el \rightarrow bool)$

$\Sigma_e = \Sigma_i \cup \{list; e : \rightarrow list, \ldots : list \times el \rightarrow list, _ \in _ : el \times list \rightarrow bool\}$

20 ?- call TOTAL-ORDER

*** $\Sigma_i = \Sigma_e = (el, bool; _ \leq _ : el \times el \rightarrow bool)$

21 ?- add

*** $\Sigma_i = (el, bool; _ = _ : el \times el \rightarrow bool, _ \leq _ : el \times el \rightarrow bool)$

$\Sigma_e = \Sigma_i \cup \{list; e : \rightarrow list, \ldots : list \times el \rightarrow list, _ \in _ : el \times list \rightarrow bool\}$

$$\text{The axioms of TOTAL-ORDER are added as semantic constraints to (the procedure body of) LIST.}$$

22 ?- refine

*** $\Sigma_i = (el, bool; _ = _ : el \times el \rightarrow bool, _ \leq _ : el \times el \rightarrow bool)$

*** $\Sigma_e = \{list, el, bool; _ = _ : el \times el \rightarrow bool, _ \leq _ : el \times el \rightarrow bool, e : \rightarrow list, \ldots : list \times el \rightarrow list, _ \in _ : el \times list \rightarrow bool, Isord : list \rightarrow bool, \ldots \in \times el \rightarrow list\}$

$$\text{The macro-command refine "automatically" adds the missing operations before "adding" the specifications obtained in the steps "21 ?-" and "18 ?-".}$$

23 ?- forget (_ = _, _ \leq _, ...) 

*** $\Sigma_i$ is unchanged.

$\Sigma_e = \{list, el, bool; e : \rightarrow list, Isord : list \rightarrow bool, _ \in _ : el \times list \rightarrow bool\}$

$$\text{The first two operations are dropped in order to simplify the example. Dropping the third operation is essential: the operation would fail to satisfy the closure condition of the sub command applied next.}$$

24 ?- sub list by (Isord(i) = true)

23
*** $\Sigma_i$, $\Sigma_e$ are unchanged.

All unordered lists are removed from the carrier set. Again, the
design unit generates formulas expressing the corresponding semantic con-
straints, viz. the closure conditions.

25 ?- forget (Isord)

*** $\Sigma_i$ is unchanged.

$\Sigma_e = (\text{list}, el, \text{bool}; \_ \rightarrow \text{list}, \_ \odot \_ : \text{list} \times \_ \rightarrow \text{list}, \_ \in \_ : \_ \times \text{list} \rightarrow \text{bool})$

This operation is superfluous because it now has the constant value true.

26 ?- e-rename (list; $\_ \rightarrow \text{list}$) into (olist; $\alpha$-$\varepsilon$ $\rightarrow$ olist)

*** $\Sigma_i$ is unchanged.

$\Sigma_e = (\text{olist}, el, \text{bool}; \alpha$-$\varepsilon$ $\rightarrow$ olist, $\_ \odot \_ : \text{olist} \times \_ \rightarrow$ olist, $\_ \in \_ : \_ \times$ olist $\rightarrow$ bool)

The user wants more suggestive names.

27 ?- ins proc ORDERED-LIST (el; $\_ = \_ : \_ \times \_ \rightarrow$ bool, $\_ \leq \_ : \_ \times$ el $\rightarrow$ bool) (olist; $\alpha$-$\varepsilon$ $\rightarrow$ olist)

The user has reached his goal. A possible call is:

28 ?- call ORDERED-LIST(int; $\_ = \_ : \_ \times \_ \rightarrow$ bool, $\_ \leq \_ : \_ \times$ int $\rightarrow$ bool) (olist; $\alpha$-$\varepsilon$ $\rightarrow$ olist)

*** $\Sigma_i = (\text{int}, \text{bool}; \_ = \_ : \_ \times \_ \rightarrow$ bool, $\_ \leq \_ : \_ \times \_ \rightarrow$ bool)

$\Sigma_e = (\text{olist}, \text{int}, \text{bool}; \alpha$-$\varepsilon$ $\rightarrow$ olist, $\_ \odot \_ : \text{olist} \times \_ \rightarrow$ olist, $\_ \in \_ : \_ \times$ olist $\rightarrow$ bool)

The renamings resulting from the call "automatically" translate the
axioms expressing that $\_ \leq \_ : \_ \times \_ \rightarrow$ bool is a total order into axioms
expressing that $\_ \leq \_ : \_ \times \_ \rightarrow$ bool is (see [LL 87] for more details).

There is no difficulty in specifying, for instance, ordered lists of or-
dered lists of integers. In principle one more call of the procedure
ORDERED-LIST suffices but it is of course necessary to first enrich
the sort olist with operations $\_ = \_ : \text{olist} \times \text{olist} \rightarrow$ bool and $\_ \leq \_ : \text{olist} \times \text{olist} \rightarrow$ bool.