Example 10.6

initial mspec **sorts** *set*, **import** *el*
opns $\emptyset : \to set$
$Insert : set \times el \to set$
vars $e, e_1, e_2 : el$
axioms $Insert(Insert(s,e),e) = Insert(s,e)$
$Insert(Insert(s,e_1),e_2) = Insert(Insert(s,e_2),e_1)$
endmspec

The specification is similar to that of Example 7.6 but the sort *el* is interpreted "loosely". □

As indicated above, a loose module specification $((\Sigma_i, \Sigma_e), \Phi)$ may be viewed as the loose specification (Σ_e, Φ). This does not hold for initial module specifications as illustrated by the above example: in contrast with the initial specification (Σ_e, Φ), the initial module specification $((\Sigma_i, \Sigma_e), \Phi)$ does not "collapse" in the presence of sorts such as *el*, the term language $T_{\Sigma_e, el}$ of which is empty.

The definition of constructive module specifications is left as Exercise 10.2-2. As an important advantage the modules they define are guaranteed to be both persistent and consistent. In fact, by a generalization of Theorem 8.11 a constructive module specification also constitutes a loose module specification and an initial module specification; as a result it is persistent and consistent. Alternatively, while a constructive specification may be viewed as a (functional) program, a constructive module specification behaves like a procedure; clearly, a procedure is "persistent" and "consistent".

10.3 A module specification language

To illustrate the concepts of a module specification language an elementary language called *MSL* is introduced. It contains the specification language *SL* introduced in Chapter 9 as a sublanguage when one views a specification as a module specification with an empty import signature.

Again, the following definition associates with each module specification *msp* a module signature $\mathcal{S}(msp)$. The constructs are straightforward generalizations of those of the specification language *SL* except for the additional construct "$\circ$".

Definition 10.7 (*Abstract syntax of the module specification language MSL)* The set of *module specifications msp* of the language *MSL* and their module signatures $\mathcal{S}(msp)$ are defined inductively:

(i) any atomic module specification *atm* is a module specification; $\mathcal{S}(atm)$ is the module signature of this atomic module specification;

(ii) if msp_1 and msp_2 are module specifications with $\mathcal{S}(msp_1) = (\Sigma_{1i}, \Sigma_{1e})$, $\mathcal{S}(msp_2) = (\Sigma_{2i}, \Sigma_{2e})$ and if:

- each sort and each operation of $\Sigma_{1e} \cap \Sigma_{2i}$ is inherited in $\mathcal{S}(msp_1)$,

- each sort and each operation of $\Sigma_{2e} \cap \Sigma_{1i}$ is inherited in $\mathcal{S}(msp_2)$,

then:

$$(msp_1 + msp_2)$$

is a module specification with $\mathcal{S}(msp_1 + msp_2) = (\Sigma_{1i} \cup \Sigma_{2i}, \Sigma_{1e} \cup \Sigma_{2e})$ (cf. Figure 10.2(a));

(iii) if msp_1 and msp_2 are module specifications with $\mathcal{S}(msp_1) = (\Sigma_i, \Sigma)$, $\mathcal{S}(msp_2) = (\Sigma, \Sigma_e)$, then:

$$(msp_2 \circ msp_1)$$

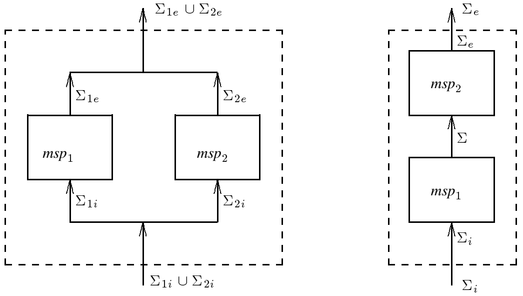
is a module specification with $\mathcal{S}(msp_2 \circ msp_1) = (\Sigma_i, \Sigma_e)$ (cf. Figure 10.2(b));

(iv) if *msp* is a module specification with $\mathcal{S}(msp) = (\Sigma_i, \Sigma_e)$, if $\mu : \Sigma_e \longrightarrow \Sigma'$ is a renaming such that $\mu(so) \notin \Sigma_i$ for each sort or operation *so* with $\mu(so) \neq so$, then:

$$(\mathbf{rename}\ msp\ \mathbf{by}\ \mu)$$

is a module specification with $\mathcal{S}(\mathbf{rename}\ msp\ \mathbf{by}\ \mu) = (\Sigma_i, \mu(\Sigma_e))$;

(v) to (x) the constructs **forget**, **extend**, **model**, **generated**, **freely generated**, **freely extend** and **quotient** are defined similarly to (iv) (cf. Definition 9.1(iv) to (vii) and Definition 9.20(viii) and (ix)). □



(a) Illustration of $msp_1 + msp_2$ (b) Illustration of $msp_2 \circ msp_1$

Figure 10.2. Illustration of the syntax of the constructs "+" and "$\circ$" of the module specification language *MSL*

Informally, the constructs of (iv) to (x) are those of the specification language *SL* being understood that they "act" on the export signature only. The syntactical constraints of the constructs "$+$" and **rename** in this definition avoid "name clashes" that would turn sorts or operations into inherited ones and thus harm the persistency. More precisely, the constraints of "$+$" express the fact that any inherited sort or operation of $\mathcal{S}(msp_1 + msp_2)$ was already inherited in $\mathcal{S}(msp_1)$ or in $\mathcal{S}(msp_2)$. The constraints of **rename** express the fact that the "new" exported sorts and operations are different from the imported ones.

As announced in Section 10.2 the language constructs **rename** and **forget** allow one to obtain module specifications, the import signature of which is not a subsignature of the export signature. In fact, it is sufficient to rename or forget inherited sorts or operations.

A concrete syntax may be defined as usual.

The semantics of the different constructs of *MSL* is similar to those of *SL* except, of course, for the new construct "$\circ$". This construct puts together module specifications msp_1 and msp_2, where msp_1 constitutes a "refinement" (in the sense of Example 1.7) of msp_2. This construct allows one to eventually obtain a module specification with an empty import signature, i.e. a genuine specification.

Definition 10.8 (*Semantics of the module specification language MSL)*
The meaning $\mathcal{M}(msp)$ of a module specification *msp* is a module (in the sense of Definition 10.2) and is defined inductively according to Definition 10.7:

(i) the meaning $\mathcal{M}(atm)$ of an atomic module specification *atm* is the module defined by this atomic module specification according to Section 10.2;

(ii) if $\mathcal{S}(msp_1) = (\Sigma_{1i}, \Sigma_{1e})$ and $\mathcal{S}(msp_2) = (\Sigma_{2i}, \Sigma_{2e})$, then:

$$\begin{aligned}&\mathcal{M}(msp_1 + msp_2)(A)\\ &\quad = \{\, B \in Alg(\Sigma_{1e} \cup \Sigma_{2e}) \mid (B \mid \Sigma_{1e}) \in \mathcal{M}(msp_1)(A \mid \Sigma_{1i}),\\ &\qquad\qquad (B \mid \Sigma_{2e}) \in \mathcal{M}(msp_2)(A \mid \Sigma_{2i}) \,\}\end{aligned}$$

for all $A \in Alg(\Sigma_{1i} \cup \Sigma_{2i})$;

(iii) if $\mathcal{S}(msp_1) = (\Sigma_i, \Sigma)$ and $\mathcal{S}(msp_2) = (\Sigma, \Sigma_e)$, then:

$$\mathcal{M}(msp_2 \circ msp_1)(A) = \bigcup_{B \in \mathcal{M}(msp_1)(A)} \mathcal{M}(msp_2)(B)$$

for all $A \in Alg(\Sigma_i)$;

(iv) if $\mathcal{S}(msp) = (\Sigma_i, \Sigma_e)$, then:

$$\begin{aligned}&\mathcal{M}(\mathbf{rename}\ msp\ \mathbf{by}\ \mu)(A)\\ &\quad = \{\, B \in Alg(\mu(\Sigma_e)) \mid (B \mid \mu) \in \mathcal{M}(msp)(A) \,\}\end{aligned}$$

for all $A \in Alg(\Sigma_i)$ (remember that $\mu : \Sigma_e \longrightarrow \Sigma'$ is bijective);

(v) to (x) similar to (iv) (cf. Definition 9.4(iv) to (vii) and Definition 9.20(viii) and (ix)). □

Fact 10.9 For any module specification *msp* with $\mathcal{S}(msp) = (\Sigma_i, \Sigma_e)$:

(i) $\mathcal{M}(msp)$ maps Σ_i-algebras into classes of Σ_e-algebras;

(ii) for each $A \in Alg(\Sigma_i)$ the value $\mathcal{M}(msp)(A)$ is an abstract data type.

Proof: Left as Exercise 10.3-1. □

Fact 10.10

(i) Each construct of the specification language *MSL* except **freely extend** and **quotient** preserves persistency.

(ii) The constructs $\circ$, **rename**, **forget**, **extend**, **freely extend** and **quotient** preserve consistency; the construct $+$ preserves consistency when applied to persistent specifications.

Proof: Left as Exercise 10.3-2. □

Note that the proof of Fact 10.10(i) draws heavily on the syntax of *MSL*.

Adding an environment to the language *MSL* may be performed along the lines of Section 9.10. The resulting language is called *e-MSL* (see Exercise 10.3-3).

Example 10.11 The following example is a module specification of the specification language *e-MSL*. In this example *msp* stands for the atomic module specification of Example 10.4.

LIST **is** *msp*;
BOOL **is loose mspec sorts freely generated** *bool*
  opns constr *True* $: \to$ *bool*
    constr *False* $: \to$ *bool*
  endmspec;
EL **is loose mspec sorts** *el* **endmspec**;
LIST $\circ$ (*BOOL* $+$ *EL*)

The import signature of this module specification is empty. Hence it may be viewed as a specification with the signature:

$(\{bool, el, list\}, \{True, False, [\,], Add, \ldots\})$

defining lists of "elements". □

To simplify the writing of module specifications it is possible to introduce "sugared notation" and/or "macros". As an example, let msp_1 and msp_2 be module specifications with $\mathcal{S}(msp_1) = (\Sigma_{1i}, \Sigma_{1e})$, $\mathcal{S}(msp_2) = (\Sigma_{2i}, \Sigma_{2e})$. The syntax of *MSL* allows one to write $msp_2 \circ msp_1$ only if $\Sigma_{1e} = \Sigma_{2i}$. This notation is generalized for the case $\Sigma_{1e} \subseteq \Sigma_{2i}$ by viewing:

$msp_2 \circ msp_1$

as a shorthand for:

$$msp_2 \circ (1_{\Sigma_{2i} - \Sigma_{1e}} + msp_1).$$

In this expression 1_Σ denotes the module specification defined by:

$$\begin{array}{rcl} \mathcal{S}(1_\Sigma) & = & (\Sigma, \Sigma), \\ \mathcal{M}(1_\Sigma)(A) & = & \{ B \in Alg(\Sigma) \mid A \simeq B \} \text{ for each } A \in Alg(\Sigma) \end{array}$$

for any signature Σ.

Further comments on module specification languages may be found in Sections 10.5 and 10.6*.

10.4 A parameterized specification language

The notion of parameterization to be introduced is an elementary one. Alternative notions will be briefly discussed in Section 10.6*. It will be indicated how each of them may be simulated by the elementary notion introduced here.

Informally, a parameter is a distinguished sort or operation of the import signature of a module specification. In the module specification of Example 10.4 the imported sort *el* is predestined as a parameter in contrast with the imported sort *bool*. The reason is that the sort *el* has to do with reusability while the sort *bool* has to do with modular design. More precisely, the intended meaning of the sort *bool* is fixed while the meaning of the sort *el* is intentionally left pending. In fact, it makes sense to use the module specification with different meanings for *el*, for instance natural numbers, strings or lists of lists of natural numbers, but it is not sensible to use the module with a meaning for *bool* other than the intended one. By the way, the difference between imported sorts or operations that are parameters and those that are not is similar to that between parameters and global variables in the procedure body of an imperative language.

As a result, imported sorts and operations that are parameters and those that are not do not differ by their semantics but merely differ by their intended use. Providing a module specification language with a parameter mechanism may therefore be reduced to the introduction of two additional language constructs called **import rename** and **import model** respectively. The first of these constructs allows parameter passing by renaming the (imported sorts and operations that constitute the) formal parameters into their actual values. In the case of Example 10.4 it allows one to rename the sort *el* into, for instance, *nat* or *string*. The second construct allows one to put semantic constraints on the parameters. For instance, a module specification of "ordered lists" that is parameterized in the sort of its elements requires that the carriers of this sort satisfy the axioms of a partial order — as will be illustrated in Example 10.16.

The following definition introduces a parameterized specification language called *PSL*. This language is identical with the module specification language *MSL* except for the two additional language constructs mentioned above.

The definition of the construct **import rename** is slightly more complex than the above comments may suggest. While the construct introduces "new" names for the sorts and operations of the import signature it generally modifies the export signature too. In fact, the inherited sorts and operations in the export signature have to be renamed accordingly. The

same holds for the inherited sorts occurring in the arities of the exported operations. For this reason the signature morphism is defined as a signature morphism $\mu : \Sigma_i \cup \Sigma_e \longrightarrow \Sigma'$ rather than $\mu : \Sigma_i \longrightarrow \Sigma'$.

Definition 10.12 *(Abstract syntax of the parameterized specification language PSL)* The set of *parameterized specifications psp* of the language *PSL* and their module signatures $\mathcal{S}(psp)$ are defined inductively:

(i) to (x) as Definition 10.7(i) to (x) but with "parameterized specification" instead of "module specification";

(xi) if *psp* is a parameterized specification with: $\mathcal{S}(psp) = (\Sigma_i, \Sigma_e)$ and if $\mu : \Sigma_i \cup \Sigma_e \longrightarrow \Sigma'$ is a surjective signature morphism satisfying the following four conditions:

  (a) for each sort s from $\Sigma_e \backslash \Sigma_i$: $\mu(s) = s$,

  (b) for each operation ω from $\Sigma_e \backslash \Sigma_i$: $\mu(\omega)$ and ω have the same operation name,

  (c) for any two different sorts or operations so_{1e} and so_{2e} from Σ_e: $\mu(so_{1e}) = \mu(so_{2e})$ implies that both so_{1e} and so_{2e} are inherited,

  (d) for any sort or operation so_i from Σ_i and so_e from Σ_e: $\mu(so_i) = \mu(so_e)$ implies that so_e is inherited,

then:

$$(\textbf{import rename } psp \textbf{ by } \mu)$$

is a parameterized specification with:

$$\mathcal{S}(\textbf{import rename } psp \textbf{ by } \mu) = (\mu(\Sigma_i), \mu(\Sigma_e));$$

(xii) if *psp* is a parameterized specification with $\mathcal{S}(psp) = (\Sigma_i, \Sigma_e)$ and if $\Phi \subseteq L(\Sigma_i)$ is a set of formulas for some logic L, then:

$$(psp \textbf{ import model } \Phi)$$

is a parameterized specification with:

$$\mathcal{S}(psp \textbf{ import model } \Phi) = \mathcal{S}(psp).$$

□

Informally, the conditions (xi)(a) and (xi)(b) express the fact that μ constitutes a renaming of the import signature. More precisely, the condition (xi)(a) expresses the fact that μ renames sorts from Σ_e only if they are inherited. The condition (xi)(b) expresses the same property for operations or, at least, for their names; the condition does not extend to their arities: if a non-inherited operation ω from Σ_e contains imported sorts in its arity, ω and $\mu(\omega)$ may differ from each other in their arities. The conditions (xi)(c) and (xi)(d) avoid "name clashes". More precisely, the condition (xi)(c) expresses the fact that μ is injective on the non-inherited sorts and operations from Σ_e. The condition (xi)(d) expresses the fact that μ may identify a sort or operation from Σ_i and a sort or operation from Σ_e only if the latter is inherited. Note that the signature morphism μ is not necessarily bijective and hence may fail to constitute a renaming in the sense of Definition 4.1. This is sensible because it must be possible that two different formal parameters get the same actual value (see Example 10.15(ii)).

Example 10.13 The example illustrates respectively the conditions (a), (c) and (d) of Definition 10.12(xi). Let $\Sigma_i = (\{a, b\}, \emptyset)$, $\Sigma_e = (\{a, c\}, \emptyset)$. The signature morphism μ is inappropriate if one of the following conditions holds:

(i) $\mu(c) = d$;

(ii) $\mu(a) = \mu(c)$;

(iii) $\mu(b) = \mu(c)$. □

A concrete syntax for *PSL* may be chosen to be identical with that for *MSL* but with **mspec** replaced by **pspec**. When using the concrete syntax of the construct **import rename** one has of course to check that the conditions (xi)(a) to (xi)(d) of Definition 10.12 hold.

In the following definition $\mu_{|\Sigma_i}$ and $\mu_{|\Sigma_e}$ denote the restriction of the function μ to the signatures Σ_i and Σ_e respectively (according to Section 2.1.3). This notation should not be confused with the notation $B \mid \ldots$ for reducts.

Definition 10.14 (*Semantics of the parameterized specification language PSL*) The meaning $\mathcal{M}(psp)$ of a parameterized specification *psp* is a module (in the sense of Definition 10.2) and is inductively defined according to Definition 10.12:

(i) to (x) as in Definition 10.8 but with "parameterized specification" instead of "module specification";

(xi) if $\mathcal{S}(psp) = (\Sigma_i, \Sigma_e)$, then:

$$\begin{aligned} &\mathcal{M}(\textbf{import rename } psp \textbf{ by } \mu)(A) \\ &\quad = \ \{ B \in Alg(\mu(\Sigma_e)) \mid (B \mid (\mu_{|\Sigma_e})) \in \mathcal{M}(psp)(A \mid (\mu_{|\Sigma_i})) \} \end{aligned}$$

for each $A \in Alg(\mu(\Sigma_i))$;

(xii) if $\mathcal{S}(psp) = (\Sigma_i, \Sigma_e)$, then:

$$\mathcal{M}(psp \textbf{ import model } \Phi)(A) = \begin{cases} \mathcal{M}(psp)(A) & \text{if } A \models \Phi, \\ \emptyset & \text{otherwise} \end{cases}$$

for each $A \in Alg(\Sigma_i)$. □

The meaning of the construct **import rename** is "as expected". In fact, first the renaming is "undone" by building the $(\mu_{|\Sigma_i})$-reduct of A, next $\mathcal{M}(psp)$ is applied and finally the result is "renamed" according to the signature morphism $\mu_{|\Sigma_e}$. The effect of the construct **import model** is to "eliminate" arguments that are not a model of Φ.

Again, for any parameterized specification *psp* the module signature of $\mathcal{M}(psp)$ is $\mathcal{S}(psp)$. Moreover, $\mathcal{M}(psp)$ maps algebras into abstract data types (see Exercise 10.4-1). The construct **import rename** preserves persistency; it also preserves consistency when applied to a persistent specification. The construct **import model** preserves persistency but not consistency (see Exercise 10.4-2).

As the parameterized specification language *PSL* is the module specification language *MSL* augmented by two language constructs, any module specification also constitutes a parameterized specification. As, moreover, a specification may be viewed as a module specification with an empty import signature, the notion of a parameterized specification encompasses the notion of a specification too.

Again, the addition of an environment leading to a language *e-PSL* presents no problem. From now on all examples are written in *e-PSL*.

Example 10.15

(i) An example of a declaration of a parameterized specification is:

PAIR **is loose pspec sorts** **freely generated** *pair*, **import** el_1,
import el_2
opns **constr** $[\_\,,\ \_] : el_1 \times el_2 \to pair$
$First : pair \to el_1$
$Second : pair \to el_2$
vars $e_1 : el_1,\ e_2 : el_2$
axioms $First([e_1, e_2]) = e_1$
$Second([e_1, e_2]) = e_2$
endpspec

(ii) An "instantiation" of this parameterized specification is, for instance:

import rename *PAIR* **by sorts** el_1, el_2 **as sorts** *nat*, *nat*.

More precisely, the parameterized specification:

PAIR **is loose pspec sorts** … **endpspec**;
import rename *PAIR* **by sorts** el_1, el_2 **as sorts** *nat*, *nat*

defines a (Σ_i, Σ_e)-module with:

$$\begin{aligned} \Sigma_i &= (\{nat\}, \emptyset), \\ \Sigma_e &= (\{nat, pair\}, \{[\_\,,\ \_] : nat \times nat \to pair, First : pair \to nat, \\ &\qquad Second : pair \to nat\}). \end{aligned}$$

The example shows that import renaming does not have to be injective.

(iii) Similarly, the parameterized specification:

PAIR **is loose pspec sorts** … **endpspec**;
NAT **is loose pspec sorts freely generated** *nat*
opns constr $0 : \to nat$
constr *Succ* $: nat \to nat$
endpspec;
(**import rename** *PAIR* **by sorts** el_1, el_2 **as sorts** *nat*, *nat*) ∘ *NAT*

defines a module with empty import signature, i.e. a specification (note that "∘" is the generalized construct introduced at the end of Section 10.3). This specification constitutes a specification of pairs of natural numbers. Its signature is (S, Ω) with:

$$S = \{nat, pair\}$$
$$\Omega = \{0 : \to nat, Succ : nat \to nat, [\_ , \_] : nat \times nat \to pair, First : pair \to nat, \ Second : pair \to nat\}.$$

□

The notation for an instantiation of a parameterized specification is clumsy, in particular because it requires repetition of the "formal parameters". This drawback may be avoided by adopting the following "sugared" notation copied from that for procedures of programming languages. According to this notation the parameters are written between brackets after the name of the parameterized specification; this rule applies to the formal parameters at the declaration as well as to the actual parameters at each instantiation. This notation has the additional advantage that a declaration explicitly distinguishes between imported sorts and operations that are parameters and those that are not. A precise definition of this notation is dispensed with. Instead it is illustrated by means of two examples.

Example 10.16 The specification of Example 10.15(iii) may now be written:

PAIR(**sorts** el_1, el_2) **is** … **endpspec**;
NAT **is** … **endpspec**;
PAIR(**sorts** *nat*, *nat*)∘*NAT*

□

The following example illustrates the use of the construct **import model** to express parameter constraints. Informally, *ORDERED-LISTS* constitutes a parameterized specification of ordered lists of elements. A relation "⊑" on the elements is provided as a formal parameter. The axioms of the construct **import model** express the fact that this relation is a partial order. The example contains an instantiation of this parameterized specification yielding a specification of ordered lists of natural numbers.

Example 10.17

ORDERED-LISTS (**sorts** *el*, **opns** $\_ \sqsubseteq \_ : el \times el \to bool$) **is**
((**loose pspec sorts freely generated** *list*, **import** *bool*, **import** *el*
opns import *True* $: \to bool$
import *False* $: \to bool$
import $\_ \sqsubseteq \_ : el \times el \to bool$

constr $[\,] : \to list$
constr *Add* $: el \times list \to list$
Is-ordered $: list \to bool$
vars $e, e_1, e_2 : el, l : list$
axioms $\mathit{Is\text{-}ordered}([\,]) = \mathit{True}$
$\mathit{Is\text{-}ordered}(\mathit{Add}(e, [\,])) = \mathit{True}$
$(e_1 \sqsubseteq e_2) = \mathit{True} \supset$
$\mathit{Is\text{-}ordered}(\mathit{Add}(e_1, \mathit{Add}(e_2, l))) =$
$\mathit{Is\text{-}ordered}(\mathit{Add}(e_2, l))$
$(e_1 \sqsubseteq e_2) = \mathit{False} \supset$
$\mathit{Is\text{-}ordered}(\mathit{Add}(e_1, \mathit{Add}(e_2, l))) = \mathit{False}$
endpspec)
import model
vars $e, e_1, e_2, e_3 : el$
axioms $(e \sqsubseteq e) = \mathit{True}$
$(e_1 \sqsubseteq e_2) = \mathit{True} \wedge (e_2 \sqsubseteq e_3) = \mathit{True} \supset$
$(e_1 \sqsubseteq e_3) = \mathit{True}$
$(e_1 \sqsubseteq e_2) = \mathit{True} \wedge (e_2 \sqsubseteq e_1) = \mathit{True} \supset e_1 = e_2$);

NATBOOL **is** (**loose pspec sorts freely generated** *bool*, **freely generated** *nat*
opns constr *True* $: \to bool$
constr *False* $: \to bool$
constr $0 : \to nat$
constr *Succ* $: nat \to nat$
$\_ \leq \_ : nat \times nat \to bool$
vars $m, n : nat$
axioms $(0 \leq n) = \mathit{True}$
$(\mathit{Succ}(m) \leq 0) = \mathit{False}$
$(\mathit{Succ}(m) \leq \mathit{Succ}(n)) = (m \leq n)$
endpspec);
ORDERED-LISTS(**sorts** *nat*, **opns** $\leq : nat \times nat \to bool$)∘ *NATBOOL*

Clearly, the module defined by *ORDERED-LISTS* is persistent and monomorphic but not consistent. Hence the specification *ORDERED-LISTS*(…) ∘ $NATBOOL$ makes sense only if the module defined by *ORDERED-LISTS* is consistent for each algebra of the (monomorphic) abstract data type defined by *NATBOOL*. This is the case because "≤" satisfies the axioms of "⊑". A proof of this property therefore constitutes an important check of the adequacy of the specification. □

10.5 Comments

The remarks on flattening, properties and proofs of Sections 9.5 and 9.6 carry over to the specification language *PSL*. Clearly, rapid prototyping is not possible as long as the meaning of the imported sorts and operations and the "actual values" of the parameters are not fixed.

Two particular properties a user may be interested in are the persistency and consistency of the module defined by a parameterized specification. As already indicated, atomic loose specifications are persistent but not necessarily consistent; atomic initial specifications are consistent but not necessarily persistent; finally, atomic constructive specifications are both persistent and consistent. The language constructs introduced that may alter persistency are **freely extend** and **quotient**; those that may alter consistency are $+$, **model**, **generated**, **freely generated**, **import rename** and **import model**. Hence these constructs have to be used with caution. From this point of view constructive specifications have a definitive advantage over initial and loose specifications. To check the persistency of a specification built from constructive specifications it is sufficient to prove that each use of the constructs **freely extend** and **quotient** preserves the persistency. A similar remark holds for consistency being understood that the constructs $+$ and *import rename* have to be checked only when they are applied to specifications that are not persistent.

From the simple examples given above it may be clear that the design of "real-life" specifications with a parameterized specification language such as *e-PSL* is tedious. On the one hand it is time-consuming to check the different syntactic constraints of the language constructs and it may be difficult to keep track of the signatures. On the other hand it is often necessary to prove persistency or consistency, or it is desirable to prove some other properties. In practice the use of a computer system supporting the design of non-trivial specifications is therefore indispensable; it should perform syntactic checks and update signatures; moreover it should contain an automatic or interactive theorem prover.

10.6* Alternative parameterization mechanisms

The parameter mechanism of *PSL* is said to belong to the *renaming approach* because its parameter passing is based on renaming. Other parameter mechanisms of the same approach have been described in the literature. Most of them differ from the parameter mechanism of *PSL* by renaming complete signatures rather than single sorts and operations.

Being based on renaming the renaming approach has a syntactic flavour. Two other approaches are now roughly sketched. Both have a semantic rather than a syntactic flavour. Nevertheless, all three approaches are essentially equivalent to each other — at least in the framework of specification presented in this book.

In the λ*-calculus approach* a parameter consists of a complete specification. More precisely, a parameterized specification is of the form:

$$\lambda X\!:\!\mathit{par}.\mathit{sp} \tag{10.1}$$

where *par* and *sp* are specifications of a specification language with environment — such as the specification language *e-PSL* — and where X is a name. It is understood that X may occur in the specification *sp*. Informally, the notation "$X\!:\!\mathit{par}$" indicates that actual parameters have to be of "type" *par*, i.e. have to belong to the abstract data type defined by *par*. Somewhat more precisely, (10.1) is equivalent to the following parameterized specification of *e-PSL*:

- the environment is extended by the declaration

  $$X \textbf{ is } \ldots$$

  where $\ldots$ stands for the loose atomic module specification $((\mathcal{S}(\mathit{par}), \mathcal{S}(\mathit{par})), Th(\mathcal{M}(\mathit{par})))$; informally, this module specification leaves the signature $\mathcal{S}(par)$ unchanged but adds the formulas of the theory of the abstract data type defined by *par*;

- *sp* is turned into a parameterized specification of *e-PSL* by defining the sorts and operations of $\mathcal{S}(par)$ as imported sorts and operations or, more precisely, as parameters.

Alternatively, one may define $\ldots$ to stand for the loose atomic module specification $((\mathcal{S}(\mathit{par}), \mathcal{S}(\mathit{par})), \emptyset)$ and add $Th(\mathcal{M}(\mathit{par}))$ to the parameterized specification obtained from *sp* with the help of the construct **import model**.

The following approach is called the *pushout approach* because the parameter passing mechanism is defined as a pushout in an appropriate category. The approach was first described in [EL78]. It makes use of the notion of a specification morphism (introduced in Exercise 9.1-9). To simplify the description of the approach it is assumed that all specifications are atomic ones or, alternatively, have been turned into atomic ones by the meaning function $\mathcal{N}$ of Section 9.9. A parameterized specification is now a specification morphism:

$$\pi : \mathit{par} \hookrightarrow \mathit{sp}(\mathit{par})$$

that is an inclusion. The specification $\mathit{par} = (\Sigma_{par}, \Phi_{par})$ constitutes the formal parameter; the specification $\mathit{sp}(\mathit{par}) = (\Sigma_{sp(par)}, \Phi_{sp(par)})$ constitutes the parameterized specification proper. The notation *sp*(*par*) is intended to suggest that *par* is "part of" *sp*(*par*) according to the inclusion π. Note that $\Sigma_{par} \subseteq \Sigma_{sp(par)}$ and $\Phi^*_{par} \subseteq \Phi^*_{sp(par)}$ because π is a specification morphism. Parameter passing is defined as a specification morphism:

$$\mu : \mathit{par} \longrightarrow \mathit{act}$$

where $\mathit{act} = (\Sigma_{act}, \Phi_{act})$ is the specification constituting the actual parameter. Note again that $(\mu(\Phi_{par}))^* \subseteq \Phi^*_{act}$ because μ is a specification morphism; this enforces the actual parameter *act* to "respect" the requirements Φ_{par} imposed on *par*. Finally, the effect of parameter passing is a specification *sp*(*act*) that is characterized as the pushout object of π and μ in the category of specifications and specification morphisms (see Figure 10.3). This pushout object is unique up to isomorphism and its existence is granted in the usual category of specifications and specification morphisms. Roughly speaking, the pushout of π and μ constitutes the "minimal completion" to a commutative square. It yields the specification morphism $\mu' : \mathit{sp}(\mathit{par}) \longrightarrow \mathit{sp}(\mathit{act})$ describing the relationship between the specifications before and after replacing *par* by *act*, and the specification morphism $\pi' : \mathit{act} \hookrightarrow \mathit{sp}(\mathit{act})$ describing the embedding of the actual parameter into the resulting specification.

The semantic counterpart of the pushout construction is given by the amalgamation lemma ([EM85], p. 217). The class $\mathcal{M}(\mathit{sp}(\mathit{act}))$ consists of the $\Sigma_{sp(act)}$-algebras C for which $(C \mid \pi') = A$ and $(C \mid \mu') = B$ with $A \in \mathcal{M}(\mathit{act})$, $B \in \mathcal{M}(\mathit{sp}(\mathit{par}))$ and $(A \mid \mu) = (B \mid \pi)$. Informally, $(A \mid \mu) = (B \mid \pi)$ requires that the parameter parts of A and B coincide; C is the algebra B with the *par* part of B replaced by A. A similar property holds for the $\Sigma_{sp(act)}$-homomorphisms.

It is not difficult to simulate the pushout approach in the specification language *e-PSL* when the parameter passing μ is surjective. The renaming effect of parameter passing may

$$\begin{tikzcd}
par \arrow[r, hook, "\pi"] \arrow[d, "\mu"'] & sp(par) \arrow[d, "\mu'"] \\
act \arrow[r, hook, "\pi'"] & sp(act)
\end{tikzcd}$$

Figure 10.3. The specification $sp(act)$ is the pushout object of the specification morphisms π and μ

then be simulated by the **import rename** construct together with the (surjective) signature morphism:

$$\mu' : \Sigma_{sp(par)} \longrightarrow \Sigma_{sp(act)}.$$

Essentially, this signature morphism performs the renaming of the import signature Σ_{par} into Σ_{act} and of the inherited sorts and operations in the export signature $\Sigma_{sp(par)}$. The condition that the different morphisms are specification morphisms may be taken care of by the construct **import model** — as in the case of the (simulation of the) λ-approach. When the parameter passing μ is not surjective the sorts and operations of $\Sigma_{act} - \mu(\Sigma_{par})$ have to be added to the result of the **import rename** construct as inherited sorts and operations. This may require a preliminary renaming to avoid "name clashes". Clearly, one of the advantages of the pushout approach with respect to its simulation in *e-PSL* is that all renamings are performed implicitly.

Exercises

10.2-1: (i) Let $msp = ((\Sigma_i, \Sigma_e), \Phi)$ be a loose module specification. Let $A_1, A_2 \in Alg(\Sigma_i)$. Show that $\mathcal{M}(msp)(A_1) = \mathcal{M}(msp)(A_2)$, if $A_1 \simeq A_2$.

(ii) As (i) for an initial module specification. (*Hint*: Exercise 9.2-1.)

10.2-2: (i) Define the notion of a constructive module specification and illustrate it by an example.

(ii) Show that — in contrast with loose module specifications — constructive module specifications cannot be viewed as constructive specifications (see the comment following Example 10.4).

10.3-1: Prove Fact 10.9.

10.3-2: Prove Fact 10.10.

10.3-3: Add an environment to the specification language *MSL*. (*Hint*: Follow closely the definitions of Section 9.10.)

10.3-4: (i) The generalized notation $msp_2 \circ msp_1$ introduced at the end of Section 10.3 was defined as a shorthand. Define this notation "directly" by rephrasing case (iii) of Definitions 10.7 and 10.8.

(ii) Show that the construct "$\circ$" introduced in (i) preserves persistency.

10.4-1: Prove that Fact 10.9 also holds for parameterized specifications.

10.4-2: (i) Prove that the construct **import rename** of the parameterized specification language *PSL* preserves persistency; prove that it preserves consistency when it is applied to a persistent specification.

(ii) Prove that the construct **import model** preserves persistency but not consistency.

10.4-3: Design a parameterized specification for lists of lists of elements containing an operation that determines the maximal element occurring in such a list of lists of elements. (*Hint*: Instantiate twice a parameterized specification of lists and perform a renaming to avoid name clashes between the results of these instantiations.)

10.4-4: (i) Design a parameterized specification of lists of elements with a sort *el* and a binary operation $\_ \circ \_ : el \times el \to el$ as parameters. The parameterized specification has to specify an operation $g : list \longrightarrow el$ with $g((el_1, \ldots, el_k)) = el_1 \circ \ldots \circ el_k$ for any $k \geq 1$.

(ii) Use the parameterized specification of (i) to obtain an operation that computes the sum of the elements of a list of natural numbers.

(iii) As (ii) but with an operation that computes the sum of all natural numbers occurring in a list of lists of natural numbers.

10.4-5: (i) Design a parameterized specification with parameters $f : el_1 \to el_2$ and $g : el_2 \to el_3$ that specifies an operation $h : el_1 \to el_3$ with $h = g \circ f$.

(ii) The parameterized specification designed in (i) shows that it is possible to "simulate" higher-order functions such as the composition of two functions. What are the limits of such "simulations"?