Specifying in the Large

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at

A Specification Language

A language for building "large" specifications from "small" ones.

- Abstract Syntax: set SL of specifications sp with signatures $\mathcal{S}(sp)$.
  - Atomic: If sp is "atomic" (a specification as previously defined), then
    $sp \in SL$
    with $\mathcal{S}(sp)$ as previously defined.
  - Union: If $sp_1 \in SL$ and $sp_2 \in SL$, then
    $(sp_1 + sp_2) \in SL$
    with $\mathcal{S}(sp_1 + sp_2) = \mathcal{S}(sp_1) \cup \mathcal{S}(sp_2)$.
  - Renaming: If $sp \in SL$ and $\mu : \mathcal{S}(sp) \to \Sigma'$ is a renaming, then
    $(\textbf{rename } sp \textbf{ by } \mu) \in SL$
    with $\mathcal{S}(\textbf{rename } sp \textbf{ by } \mu) = \mu(\mathcal{S}(sp))$.
  - Forgetting: If $sp \in SL$, S is a set of sorts and Ω is a set of operations such that $(S, \Omega) \subseteq \mathcal{S}(sp)$ and $\mathcal{S}(sp) \backslash (S, \Omega)$ is a signature, then
    $(sp \textbf{ forget } (S, \Omega)) \in SL$
    with $\mathcal{S}(sp \textbf{ forget } (S, \Omega)) = \mathcal{S}(sp) \backslash (S, \Omega)$.
  - ...

A Specification Language (Contd)

- Abstract Syntax: set SL of specifications sp with signatures $\mathcal{S}(sp)$.
  - ...
  - Extension:If $sp \in SL$, S is a set of sorts and Ω is a set of operations such that $\mathcal{S}(sp) \cup (S, \Omega)$ is a signature, then
    $(sp \textbf{ extend } (S, \Omega)) \in SL$
    with $\mathcal{S}(sp \textbf{ extend } (S, \Omega)) = \mathcal{S}(sp) \cup (S, \Omega)$.
  - Modelling: if $sp \in SL$ and $\Phi \subseteq L(\mathcal{S}(sp))$ for some logic L, then
    $(sp \textbf{ model } \Phi) \in SL$
    with $\mathcal{S}(sp \textbf{ model } \Phi) = S(sp)$.
  - Restricting: if $sp \in SL$ with $\mathcal{S}(sp) = (S, \Omega)$, if $S_c \subseteq S$ is a set of sorts and if $\Omega_c \subseteq \Omega$ is a set of operations with target sorts in S_c, then
    $(sp \textbf{ generated in } S_c \textbf{ by } \Omega_c) \in SL$ and
    $(sp \textbf{ freely generated in } S_c \textbf{ by } \Omega_c) \in SL$
    with $\mathcal{S}(sp \textbf{ generated in } S_c \textbf{ by } \Omega_c) = \mathcal{S}(sp)$
    and $\mathcal{S}(sp \textbf{ freely generated in } S_c \textbf{ by } \Omega_c) = \mathcal{S}(sp)$.

$\mathcal{S}(sp)$ is a signature for any specification $sp \in SL$.

Concrete Syntax

- (S,Ω) :
  sorts *sorts*
  opns *operations*
- $\mu : \Sigma \to \Sigma'$
  sorts $s_1, \ldots, s_k$ **opns** $\omega_1, \ldots, \omega_l$ **as**
  sorts $s'_1, \ldots, s'_k$ **opns** $\omega'_1, \ldots, \omega'_l$
- Example: $\mathcal{S}(sp) = (\{s,t\}, \{m : s \times t \to s, n : t \times s \to t, n :\to s\})$.
  (**rename** *sp*
  by sorts *s* **opns** $n : t \times s \to t$
  as sorts *u* **opns** $q : t \times u \to t$)
  means (**rename** *sp* **by** μ) with $\mu : \Sigma \to \Sigma'$ defined as
  $\Sigma = \mathcal{S}(sp), \Sigma' = \mu(\Sigma)$
  $\mu(s) = u, \mu(t) = t$
  $\mu(m : s \times t \to s) = (m : u \times t \to u)$
  $\mu(n : t \times s \to t) = (q : t \times u \to t)$
  $\mu(n :\to s) = (n :\to u)$

Semantics

- Semantics: $\mathcal{M}(sp)$ is inductively defined:
  - $\mathcal{M}(sp)$ of an atomic specification *sp* is as previously defined;
  - $\mathcal{M}(sp_1 + sp_2) = \{A \in Alg(\mathcal{S}(sp_1 + sp_2)) \mid$
    $(A|\mathcal{S}(sp_1)) \in \mathcal{M}(sp_1), (A|\mathcal{S}(sp_2)) \in \mathcal{M}(sp_2)\}$;
    $A|\Sigma \ldots \Sigma$-reduct of A
    - Hide sorts and operations that do not occur in signature Σ.
  - $\mathcal{M}($**rename** sp **by** $\mu) = \{A \in Alg(\mu(\mathcal{S}(sp))) \mid (A|\mu) \in \mathcal{M}(sp)\}$;
    $A|\mu \ldots \mu$-reduct of A
    - Rename sorts and operations as indicated by renaming μ.
  - $\mathcal{M}(sp$ **forget** $(S,\Omega)) = \mathcal{M}(sp) \mid (\mathcal{S}(sp) \backslash (S,\Omega))$;
  - $\mathcal{M}($**extend** sp **by** $(S,\Omega)) =$
    $\{A \in Alg(\mathcal{S}(sp) \cup (S,\Omega)) \mid (A|\mathcal{S}(sp)) \in \mathcal{M}(sp)\}$;
  - $\mathcal{M}(sp$ **model** $\Phi) = \mathcal{M}(sp) \cap Mod_{\mathcal{S}(sp)}(\Phi)$;
  - $\mathcal{M}(sp$ **generated in** S_c **by** $\Omega_c) =$
    $\{A \in \mathcal{M}(sp) \mid A \text{ is generated in } S_c \text{ by } \Omega_c\}$;
    $\mathcal{M}(sp$ **freely generated in** S_c **by** $\Omega_c) =$
    $\{A \in \mathcal{M}(sp) \mid A \text{ is freely generated in } S_c \text{ by } \Omega_c\}$.

Pragmatics

- **Operator** + builds the "union" of two specifications sp_1 and sp_2.
  - If sp_1 and sp_2 have common sorts/operations, only those algebras of $\mathcal{M}(sp_1)$ and $\mathcal{M}(sp_2)$ contribute to this union that have the same interpretation of the common parts.
- **rename** may be used to avoid "name clashes".
  - If two specifications have the same sort/operator with different meaning, rename this entity in one of them before constructing the union of both specifications.
- **forget** hides sorts and operations.
  - For auxiliary entities that are not part of the "public" specification interface.
- **extend** introduces new sorts and operations.
  - Loose semantics of new entities.
- **model** and **(freely) generated by** filter out unintended algebras.

Properties

Take specification $sp \in SL$.

- Every algebra in $\mathcal{M}(sp)$ has signature $\mathcal{S}(sp)$.
- $\mathcal{M}(sp)$ is an abstract datatype.

The semantics of the specification language is "as expected".

Example

```
(extend (
    (loose spec
       sorts freely generated bool
       opns constr True :→ bool, False :→ bool
    endspec +
    loose spec
       sorts nat
       opns 0 :→ nat, Succ : nat → nat
    endspec)
    freely generated
       in sorts nat
       by opns 0 :→ nat, Succ : nat → nat)
  by opns _ ≤ _ : nat × nat → bool)
model vars m, n : nat
  axioms
    0 ≤ n = True
    Succ(m) ≤ 0 = False
    Succ(m) ≤ Succ(n) = m ≤ n
```

A (still rather clumsy) specification of the "classical" algebra.

A Specification Language with Environments

Introduce an environment e that maps names to specifications.

- Abstract syntax: set $SL(e)$ of specs sp with signatures $\mathcal{S}(e, sp)$.
  - If n is a name such that $e(n)$ is defined, then
    $$n \in SL(e)$$
    with $\mathcal{S}(e, n) = \mathcal{S}(e, e(n))$.
  - ... (as before)
    - Using $SL(e)$ and $\mathcal{S}(e, sp)$ rather than SL and $\mathcal{S}(sp)$.
- Semantics: $\mathcal{M}(e, sp)$ is inductively defined:
  - $\mathcal{M}(e, n) = \mathcal{M}(e, e(n))$
  - ... (as before)
    - Using $\mathcal{M}(e, sp)$ and $\mathcal{S}(e, sp)$ rather than $\mathcal{M}(sp)$ and $\mathcal{S}(sp)$.

Specifications can be named.

Concrete Syntax

- Environment: defined by a declaration (sequence).
  - ϵ: the empty declaration sequence.
    - Denoting the environment that does not contain any mapping.
  - n **is** sp: a sequence with a single declaration.
    - Denoting the environment that only maps n to sp.
  - d; n **is** sp: declaration sequence d followed by a declaration.
    - Denoting the environment that maps n to sp and every other name to the same specification as the environment denoted by d does.
- Specification: d; sp
  - Declaration (sequence) d denoting an environment e.
  - $sp \in SL(e)$.
  - Special case: ϵ; sp is simply written as sp.

Specifications are defined in the context of declarations.

Example

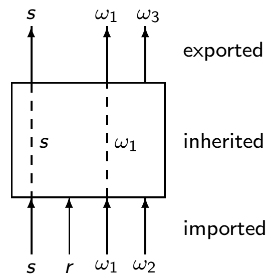
```
BOOL is
  loose spec
    sorts freely generated bool
    opns constr True :→ bool, False :→ bool
  endspec;
NAT is
  loose spec
    sorts nat
    opns 0 :→ nat, Succ : nat → nat
  endspec;
BOOLNAT is BOOL + NAT
  freely generated
    in sorts nat
    by opns 0 :→ nat, Succ : nat → nat;
extend BOOLNAT by opns _ ≤ _ : nat × nat → bool
model vars m, n : nat
  axioms
    0 ≤ n = True
    Succ(m) ≤ 0 = False
    Succ(m) ≤ Succ(n) = m ≤ n
```

A structured specification of the "classical" algebra.

Module Signatures

A module is an entity with a well-defined interface to its environment.

- Module signature: pair (Σ_i, Σ_e).
  - Import signature Σ_i.
    - A sort/operation from Σ_i is called imported.
  - Export signature Σ_e.
    - A sort/operation from Σ_e is called exported.
  - A sort/operation from $\Sigma_i \cap \Sigma_e$ is called inherited.
- Example: $\Sigma_i = (\{r, s\}, \{\omega_1, \omega_2\}), \Sigma_e = (\{s\}, \{\omega_1, \omega_3\})$.



Modularized Abstract Datatypes

Take module signature (Σ_i, Σ_e).

- A (Σ_i, Σ_e)-module (also called a "modularized abstract datatype")
  $M : Alg(\Sigma_i) \to \mathbb{P}(Alg(\Sigma_e))$
  - is a mapping from Σ_i-algebras to classes of Σ_e-algebras such that
  - for every $A \in Alg(\Sigma_i)$, $M(A) \subseteq Alg(\Sigma_e)$ is an abstract datatype.
- A (Σ_i, Σ_e)-module M is persistent for an algebra $A \in Alg(\Sigma_i)$, if
  $\forall B \in M(A) : (A|\Sigma_i \cap \Sigma_e) \simeq (B|\Sigma_i \cap \Sigma_e)$.
  - Inherited sorts/operations have the same meaning in A and in $M(A)$.
- A (Σ_i, Σ_e)-module M is consistent for an algebra $A \in Alg(\Sigma_i)$, if
  $M(A) \neq \emptyset$.
  - The mapping M is "effective".
- A (Σ_i, Σ_e)-module M is monomorphic for an algebra $A \in Alg(\Sigma_i)$, if
  $M(A)$ is monomorphic.
- M is persistent/consistent/monomorphic, if
  - it is consistent/persistent/monomorphic for every $A \in Alg(\Sigma_i)$.

Loose Module Specifications

Take logic L.

- Abstract syntax: a loose module specification is a pair
  $sp = ((\Sigma_i, \Sigma_e), \Phi)$ consisting of
  - a module signature (Σ_i, Σ_e) with $\Sigma_i \subseteq \Sigma_e$, and
  - a set of formulas $\Phi \subseteq L(\Sigma_e)$.
    - Entities of Σ_i are specified "elsewhere".
- Semantics: the meaning of a loose module specification
  $sp = ((\Sigma_i, \Sigma_e), \Phi)$ is the (Σ_i, Σ_e)-module defined as
  $$\mathcal{M}(sp)(A) = \{B \in Alg(\Sigma_e) \mid B \models \Phi \wedge B|\Sigma_i \simeq A\}$$
  for every $A \in Alg(\Sigma_i)$.

A loose module specification defines a persistent (but not necessarily consistent) module.

Concrete Syntax

$\Sigma_i = (\{bool, el\}, \{True, False\}), \Sigma_e = \Sigma_i \cup (\{list\}, \{[\,], Add, .\})$.

loose mspec
  sorts import *bool*, **import** *el*, *list*
  opns
    import $True :\to bool$
    import $False :\to bool$
    $[\,] :\to list$
    $Add : el \times list \to list$
    $\_ . \_ : list \times list \to list$
  vars $l, m : list, e : el$
  axioms
    $[\,].l = l$
    $Add(e, l).m = Add(e, l.m)$
endspec

Elements of the import signature are prefixed by the keyword **import**.

A Module Specification Language

- Abstract syntax: set MSL of specs sp with signatures $\mathcal{S}(sp)$:
  - If sp is a loose module specification, then

    $sp \in MSL$

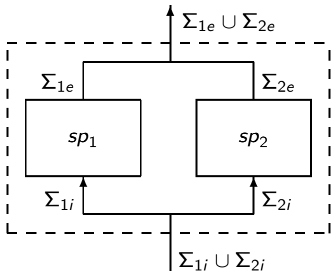
    with $\mathcal{S}(sp)$ as previously defined;
  - If $sp_1, sp_2 \in MSL$ with $\mathcal{S}(sp_1) = (\Sigma_{1i}, \Sigma_{1e})$ and $\mathcal{S}(sp_2) = (\Sigma_{2i}, \Sigma_{2e})$
    - and each sort and operation of $\Sigma_{1e} \cap \Sigma_{2i}$ is inherited in $\mathcal{S}(sp_1)$,
    - and each sort and operation of $\Sigma_{2e} \cap \Sigma_{1i}$ is inherited in $\mathcal{S}(sp_2)$,

    (no sort/operation introduced by one specification is imported by the other one)

    then

    $(sp_1 + sp_2) \in MSL$

    with $\mathcal{S}(sp_1 + sp_2) = (\Sigma_{1i} \cup \Sigma_{2i}, \Sigma_{1e} \cup \Sigma_{2e})$;
  - ...



A Module Specification Language (Contd)

- Abstract syntax: set MSL of specs sp with signatures $\mathcal{S}(sp)$:
  - ...
  - If $sp_1, sp_2 \in MSL$ with $\mathcal{S}(sp_1) = (\Sigma_i, \Sigma)$ and $\mathcal{S}(sp_2) = (\Sigma, \Sigma_e)$, then

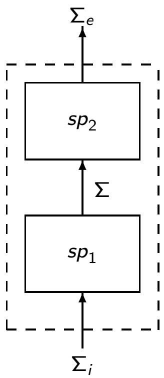
    $(sp_2 \circ sp_1) \in MSL$

    with $\mathcal{S}(sp_2 \circ sp_1) = (\Sigma_i, \Sigma_e)$.
  - If $sp \in MSL$ with $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ and $\mu : \Sigma_e \to \Sigma'$ is a renaming with $\mu(a) \notin \Sigma_i$ for each sort/operation a with $\mu(a) \neq a$, then

    (**rename** sp **by** μ) $\in MSL$

    with $\mathcal{S}$(**rename** sp **by** μ) $= (\Sigma_i, \mu(\Sigma_e))$;

    (no clash between imported sorts/operations and "new" exported sorts/operations)
  - The constructs **forget**, **extend**, **model**, and **(freely) generated** are defined similarly as before.



The language SL can be considered as a sublanguage of MSL where all module specifications have empty import signatures.

Semantics

- Semantics: $\mathcal{M}(sp)$ is inductively defined:
  - $\mathcal{M}(sp)$ of a loose module specification sp is as previously defined;
  - If $\mathcal{S}(sp_1) = (\Sigma_{1i}, \Sigma_{1e})$ and $\mathcal{S}(sp_2) = (\Sigma_{2i}, \Sigma_{2e})$, then

    $\mathcal{M}(sp_1 + sp_2)(A) = \{B \in Alg(\Sigma_{1e} \cup \Sigma_{2e}) \mid$
    $(B|\Sigma_{1e}) \in \mathcal{M}(sp_1)(A|\Sigma_{1i}) \wedge (B|\Sigma_{2e}) \in \mathcal{M}(sp_2)(A|\Sigma_{2i})\}$;
  - If $\mathcal{S}(sp_1) = (\Sigma_i, \Sigma)$ and $\mathcal{S}(sp_2) = (\Sigma, \Sigma_e)$, then

    $\mathcal{M}(sp_2 \circ sp_1)(A) = \bigcup_{B \in \mathcal{M}(sp_1)(A)} \mathcal{M}(sp_2)(B)$;
  - If$\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$, then

    $\mathcal{M}$(**rename** sp **by** μ)$(A) =$
    $\{B \in Alg(\mu(\Sigma_e)) \mid (B|\mu) \in \mathcal{M}(sp)(A)\}$;
  - The semantics of the constructs **forget**, **extend**, **model**, and **(freely) generated** is defined similarly as before.

Generalization of the semantics of a specification from an ADT to a function that takes an algebra and returns an ADT.

Example

As shown in previous section, also module specifications may be named.

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

Since the import signature of this specification is empty, it may be considered as a specification with signature $(\{bool, el, list\}, \{True, False, [\,], Add\})$.

Properties

Take specification $sp \in MSP$ with $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$.

- $\mathcal{M}(sp)$ maps Σ_i-algebras to classes of Σ_e-algebras.
- $\mathcal{M}(sp)(A)$ is an abstract datatype, for each Σ_i-algebra A.
- Each construct of the module specification language preserves persistency.
  - Thus any module specification is persistent, provided that the atomic specifications in it are.
- Each construct of the module specification language except **model**, **generated**, and **freely generated** preserves consistency.
  - Thus any module specification that does not use these constructs is consistent, provided that the atomic specifications in it are.

The semantics of the module specification language is "as expected".

Import Signatures Revisited

What is actually the purpose of a specification's import signature?

- Consider $LIST \circ (BOOL + \ldots)$
  - *LIST* uses an imported sort *bool*.
  - *BOOL* provides a specification of this sort.
  - Purpose: we want to reuse *bool* in different contexts.
    - Only a single specification *BOOL* suffices; its can then be used by import in multiple other specifications.
- Consider $LIST \circ (\ldots + EL)$
  - *LIST* uses an imported sort *el*.
  - But we actually do not expect a specification for *el* !
  - Rather *el* saves as a "placeholder" for some *other* sort.
  - Purpose: we want to instantiate *el* by different sorts.
    - Only a single specification *LIST* suffices; its sort *el* can then be instantiated by multiple concrete sorts.
  - Two additional mechanisms are needed:
    - A mapping of the specified sorts to the actual sorts.
    - A mean to express semantic constraints on the imported sorts.

Parameterized Specifications

We extend module specifications to parameterized specifications.

- Abstract Syntax: set PSL of specifications sp with signatures $\mathcal{S}(sp)$.
  - If $sp \in PSL$ with $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ and if $\mu : \Sigma_i \cup \Sigma_e \to \Sigma'$ is a signature morphism that "renames the import signature", i.e.
    - $\mu(s) = s$ for each sort $s \in \Sigma_e \backslash \Sigma_i$,
    - $\mu(\omega)$ and ω have the same operation name for each op. $\omega \in \Sigma_e \backslash \Sigma_i$,

    and that avoids "name clashes" with introduced sorts, i.e.
    - $\mu(a) = \mu(b)$ implies a and b are inherited, for all $a, b \in \Sigma_e, a \neq b$,
    - $\mu(a) = \mu(b)$ implies b is inherited for each a from Σ_i and b from Σ_e,

    then

    $(\textbf{import rename } sp \textbf{ by } \mu) \in PSP$

    with $\mathcal{S}(\textbf{import rename } sp \textbf{ by } \mu) = (\mu(\Sigma_i), \mu(\Sigma_e))$;
  - If $sp \in PSP$ with $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ and $\Phi \subseteq L(\Sigma_i)$ for logic L, then

    $(sp \textbf{ import model } \Phi) \in PSP$

    with $\mathcal{S}(sp \textbf{ import model } \Phi) = \mathcal{S}(sp)$;
  - ...(as before using PSL rather than MSL).

Example

Take $\Sigma_i = (\{a, b\}, \emptyset), \Sigma_e(\{a, c\}, \emptyset)$.

- A signature morphism μ suitable for **import rename** must *not* allow
  - $\mu(c) = d$,
    - First condition is violated.
    - μ renames an entity introduced by the specification.
  - $\mu(a) = \mu(c)$,
    - Third condition is violated.
    - μ maps exported sort a to the same name as the introduced sort c.
  - $\mu(b) = \mu(c)$.
    - Fourth condition is violated.
    - μ maps imported sort b to the same name as the introduced sort c.

The signature morphism is intended to map actual "argument" sorts to formal "parameter" sorts.

Semantics

- Semantics: $\mathcal{M}(sp)$ is inductively defined:
  - If $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$, then for each $A \in Alg(\mu(\Sigma_i))$

    $$\mathcal{M}(\textbf{import rename } sp \textbf{ by } \mu)(A) = \{B \in Alg(\mu(\Sigma_e)) \mid (B|(\mu_{|\Sigma_e})) \in \mathcal{M}(sp)(A|(\mu_{|\Sigma_i}))\};$$
    - Let $f : A \to B$ and $C \subseteq A$. The restriction $f_{|C}$ is the function

      $f_{|C} : C \to B$

      $f_{|C}(c) = f(c)$
  - If $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$, then for each $A \in Alg(\mu(\Sigma_i))$

    $$\mathcal{M}(sp \textbf{ import model } \Phi)(A) = \begin{cases} \mathcal{M}(sp)(A) & \text{if } A \models \Phi \\ \emptyset & \text{otherwise} \end{cases};$$
  - ...(as with module specifications).

Properties

Take specification $sp \in PSL$ with $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$.

- $\mathcal{M}(sp)$ maps Σ_i-algebras to classes of Σ_e-algebras.
- $\mathcal{M}(sp)(A)$ is an abstract datatype, for each Σ_i-algebra A.
- **import rename** and **import model** preserve persistency.
- Only **import rename** preserves consistency.

The semantics of the parameterized specification language is "as expected".

Example

Parameterized specification

> **loose pspec**
> **sorts import** el_1, **import** el_2, **freely generated** *pair*
> **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**

defines a (Σ_i, Σ_e)-module with

$\Sigma_i = (\{el_1, el_2\}, \emptyset)$,
$\Sigma_e = (\{el_1, el_2, pair\},$
$\{[\_,\_] : el_1 \times el_2 \to pair, First : pair \to el_1, Second : pair \to el_2\})$.

Specification of (el_1, el_2)-pairs.

Example (Contd)

Parameterized specification

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

defines a (Σ_i, Σ_e)-module with

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

Specification of *nat*-pairs.

Example (Contd'2)

Parameterized specification

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

defines a module with empty import signature and export signature

$\Sigma = \{nat, pair\},$
$\{[\_,\_] : nat \times nat \to pair, First : pair \to nat, Second : pair \to nat\})$.

Specification of pairs of natural numbers.

Example (Contd'3)

Better notation for parameterized specifications:

> *PAIR*(**sorts** el_1, el_2) **is loose pspec . . . endpspec**;
> *NAT* **is loose pspec . . . endpspec**;
> *PAIR*(**sorts** nat, nat) $\circ$ *NAT*

Similar to definition and application of parameterized procedures.

Example

OLISTS(**sorts** *el*, **opns** $\_ \sqsubseteq \_ : el \times el \to bool$) **is**
(**loose pspec**
sorts import *bool*, **import** *el*, **freely generated** *list*
opns
import $True :\to bool$
import $False :\to bool$
import $\_ \sqsubseteq \_ : el \times el \to bool$
constr $[\,] :\to list$
constr $Add : el \times list \to list$
vars $e, e_1, e_2 : el, l : list$
axioms
$ordered([\,]) = True$
$ordered(Add(e, [\,])) = True$
$(e_1 \sqsubseteq e_2) = True \Rightarrow ordered(Add(e_1, Add(e_2, l))) = ordered(Add(e_2, l))$
$(e_1 \sqsubseteq e_2) = False \Rightarrow ordered(Add(e_1, Add(e_2, l))) = False$
enspec)
import model
vars $e, e_1, e_2, e_3 : el$
axioms
$(e \sqsubseteq e) = True$
$(e_1 \sqsubseteq e_2) = True \wedge (e_2 \sqsubseteq e_3) = True \Rightarrow (e_1 \sqsubseteq e_3) = True$
$(e_1 \sqsubseteq e_2) = True \wedge (e_2 \sqsubseteq e_1) \Rightarrow e_1 = e_2$

Example (Contd)

OLISTS(**sorts** *el*, **opns** $\_ \sqsubseteq \_ : el \times el \to bool$) **is**
...;
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
$(0 \leq n) = True$
$(Succ(m) \leq 0) = False$
$(Succ(m) \leq Succ(n)) = (m \leq n)$
endpspec;
OLISTS(**sorts** *nat*, **opns** $\leq : nat \times nat \to bool$) $\circ$ *NATBOOL*

Specification of ordered list of natural numbers; specification is adequate, because $\leq$ satisfies the axioms imposed on $\sqsubseteq$

Open Issues

- Constructs **extend** and **model** have loose semantics.
  - Initial semantics counterparts require the notion of "free extensions".
    - Generalization of the notion of "initial algebra".
    - Algebras in free extension have common "stem" which does not "take part" in initiality.
  - Initial counterpart of **extend** is (**freely extend** *sp* **by** (S, Ω)).
    - Constructs only free extensions (rather than all extensions.
  - Initial counterpart of **model** is (*sp* **quotient** Φ).
    - Builds quotient algebras (rather than removing algebras).
- Specifications can be flattened.
  - Compound specifications can be translated to equivalent atomic ones.
- There exist alternative parameterization mechanisms.
  - We have used the *renaming approach* with a syntactic flavor.
  - There exists approaches with a semantic flavor.
    - Based on λ-calculus or on category theory.
  - However, all approaches are ultimately equivalent in expressive power.

CafeOBJ

CafeOBJ supports some of the described constructions.

- Named modules:
  - *n* **is loose (initial) spec . . . endspec**
    ```
    module* (module!) n { ... }
    ```
  - *n* **is** . . . (arbitrary module expression)
    ```
    make n (...)
    ```
- References to named modules: *n*
  ```
  n
  ```
- Union: $sp_1 + sp_2$
  ```
  SP1 + SP2
  ```
- Renaming: **rename** *sp* **by** . . .
  ```
  SP * { sort s1 -> s1' op w1 -> w1' ... }
  ```
- Extension and Modelling: *sp* **extend** . . . **model** . . .
  ```
  protecting (SP) signature { ... } axioms { ... }
  ```
- . . .

CafeOBJ (Contd)

- . . .
- Parameterized Modules
  - Parameters are whole modules (rather than sorts or operations).
    ```
    module* SP1 { [ s1 ... ] op o1: ... }
    module* (module!) SP (P1::SP1, ...) { ... }
    ```
- Module Instantiation
  - "Views" specify bindings of actual arguments to formal parameters.
    ```
    module! SP2 { [ s2 ... ] op o2: ... }
    view V from SP1 to SP2 { sort s1 -> s2, op o1 -> o2, ... }
    ```
  - Instantiation of parameter module by a declared view
    ```
    SP(P1 <= V1, ...)
    ```
  - Instantiation of parameter module by ad-hoc view
    ```
    SP(P1 <= view to SP2
              { sort s1 -> s2, op o1 -> o2. ... }, ...)
    ```

See the CafeOBJ manual for more details

Parameterized Modules in Programming

Parameterized modules are now part of various programming languages.

- ML functors
  ```
  signature ELEM = sig ... end;
  functor STACK(structure EL: ELEM) = struct ... end;
  ```
- C++ templates (type checking only after instantiation)
  ```
  template <class EL> class Stack { ... }
  ```
- Java generic types
  ```
  interface ELEM { ... }
  class Stack<EL implements ELEM> { ... }
  ```
- C# generic types
  ```
  interface ELEM { ... }
  class Stack<EL> where EL:ELEM { ... }
  ```
- . . .