

Data-Oriented System Development

Prof. Martin Wirsing

30.10.2002

Refinement Technics

Goals

- Refinement of functional requirements: functional refinement
- Refinement of data structures: change of data structures
- Transition from executable specifications to functional programs

The Refinement-Concept: Refinement through Inclusion of Model Classes

Definition:

SP' is a **Refinement** of SP (write $SP \rightsquigarrow SP'$),
if $\text{Sig}(SP') = \text{Sig}(SP)$ and $\text{Mod}(SP') \subseteq \text{Mod}(SP)$.

The Refinement-Concept: Refinement through Inclusion of Model Classes

Definition:

SP' is a **Refinement** of SP (write $SP \rightsquigarrow SP'$),
if $\text{Sig}(SP') = \text{Sig}(SP)$ and $\text{Mod}(SP') \subseteq \text{Mod}(SP)$.

Lemma:

- \rightsquigarrow is a partial order, which means \rightsquigarrow is reflexive, transitive and anti-symmetric.
- All specification-building operations are monotonic w.r.t. \rightsquigarrow . For example:

$$SP \rightsquigarrow SP_1 \quad \Rightarrow \quad SP \text{ hide } SY \rightsquigarrow SP_1 \text{ hide } SY$$

- For every specification expression $C[SP]$, built by specification-building operations we have:

$$SP \rightsquigarrow SP_1 \quad \Rightarrow \quad C[SP] \rightsquigarrow C[SP_1]$$

That means, every local refinement is global.

Definition:

1. (Σ_1, E_1) is called **axiomatic extension** of (Σ, E) if

$$\Sigma \subseteq \Sigma_1 \text{ and } E \subseteq E_1.$$

We speak of **axiomatic enrichment**, if only new axioms are added.

Definition:

1. (Σ_1, E_1) is called **axiomatic extension** of (Σ, E) if

$$\Sigma \subseteq \Sigma_1 \text{ and } E \subseteq E_1.$$

We speak of **axiomatic enrichment**, if only new axioms are added.

2. (Σ_1, E_1) is called **axiomatic refinement** of (Σ, E) if $\Sigma \subseteq \Sigma_1$ and $E_1 \vdash E$.

Definition:

1. (Σ_1, E_1) is called **axiomatic extension** of (Σ, E) if

$$\Sigma \subseteq \Sigma_1 \text{ and } E \subseteq E_1.$$

We speak of **axiomatic enrichment**, if only new axioms are added.

2. (Σ_1, E_1) is called **axiomatic refinement** of (Σ, E) if $\Sigma \subseteq \Sigma_1$ and $E_1 \vdash E$.

Axiomatic extension and refinement are refinements in the following sense:

$$(\Sigma, E) \rightsquigarrow (\Sigma_1, E_1) \text{ hide } (\Sigma_1 \setminus \Sigma)$$

Example (Axiomatic Enrichment and Refinement):

1. Axiomatic Enrichment: Adding of axioms:

```
spec LSETNAT =  
  BOOL and NAT then  
  sorts      Set  
  ops       empty : Set  
            { _ } : Nat → Set  
            add : Nat × Set → Set  
            _ U _ : Set × Set → Set  
            _ aus _ : Nat × Set → Bool  
  
  vars       $x, y : \text{Nat}, s_1, s_2 : \text{Set}$   
  axioms     $\text{add}(x, s) = s \cup \{x\}$   
             $x \text{ in empty} = \text{false}$   
             $x \text{ in } \{y\} \equiv (x = y)$   
             $x \text{ in } (s_1 \cup s_2) = (x \text{ aus } s_1) \text{ or } (x \text{ aus } s_2)$   
  
end
```

```
spec SETNAT =  
  LSETNAT then  
    vars       $s_1, s_2, s_3 : \text{Set}$   
    axioms    $s \cup s = s$   
               $s_1 \cup s_2 = s_2 \cup s_1$   
               $s_1 \cup (s_2 \cup s_3) = (s_1 \cup s_2) \cup s_3$   
end
```

Obviously $\text{LSETNAT} \rightsquigarrow \text{SETNAT}$ holds. SETNAT is an axiomatic enrichment of LSETNAT.

2. Axiomatic Refinement modulo Renaming:

Refinement of "loose" sets by sequences.

Step 1: Specification of lists of natural numbers.

```
spec LISTNAT =  
  NAT %% {Spec. of nat. numbers with boolean equality eq} % then  
  free type   List ::= nil|cons(first: Elem; rest: List)  
  ops        _ ++ _ : List × List → List  
  vars       x : Elem; xl, yl : List  
  axioms     nil ++ yl = yl  
              cons(x, xl) ++ yl = cons(x, xl ++ yl)  
end
```

Step 2: Extend LISTNAT by operations of sets and define them via list operations.

```

spec SET_by_LIST =
  LISTNAT then
    ops
      empty : List
      { _ } : Nat → List
      add : Nat × List → List
      _ ∪ _ : List × List → List
      _ in _ : Nat × List → Bool
    vars
      x, y : Nat; s, s1, s2 : List
    axioms
      %%{Explicit definition set operations empty, { _ }, add, ∪}%
      empty = nil
      { x } = cons(x, nil)
      add(x, s) = cons(x, s)
      s1 ∪ s2 = s1 ++ s2
      %%{Inductive definition of set operation _ in _}%
      x in nil = false
      x in cons(y, s) = (eq(x, y) or (x in s))
  end

```

Step 3: Rename the sort List to Set and hide the list operations.

```
spec SET_by_LIST1 =  
  (SET_by_LIST with [List  $\mapsto$  Set])  
  hide nil, cons, first, rest, ++  
end
```

Step 3: Rename the sort List to Set and hide the list operations.

```
spec SET_by_LIST1 =  
  (SET_by_LIST with [List  $\mapsto$  Set])  
  hide nil, cons, first, rest, ++  
end
```

Proposition:

SET_by_LIST1 is a refinement of LSETNAT.

Change of Data Structures

"Simulation" of a Σ -structure through a Σ_1 -structure.

- A (S, F) -structure A is simulated by a (S_1, F_1) -structure B if every carrier set A_s of A is represented by a subset

$$Rep_s \subseteq B_{s'}$$

of a carrier set $B_{s'}$ of B and if every function symbol $f \in F$ is represented by a function symbol $f_1 \in F_1$.

Change of Data Structures

"Simulation" of a Σ -structure through a Σ_1 -structure.

- A (S, F) -structure A is simulated by a (S_1, F_1) -structure B if every carrier set A_s of A is represented by a subset

$$Rep_s \subseteq B_{s'}$$

of a carrier set $B_{s'}$ of B and if every function symbol $f \in F$ is represented by a function symbol $f_1 \in F_1$.

- Plenty elements of Rep_s can represent the same element of A_s . This induces a congruence relation \sim_s .

Change of Data Structures

"Simulation" of a Σ -structure through a Σ_1 -structure.

- A (S, F) -structure A is simulated by a (S_1, F_1) -structure B if every carrier set A_s of A is represented by a subset

$$Rep_s \subseteq B_{s'}$$

of a carrier set $B_{s'}$ of B and if every function symbol $f \in F$ is represented by a function symbol $f_1 \in F_1$.

- Plenty elements of Rep_s can represent the same element of A_s . This induces a congruence relation \sim_s .
- Rep must be closed under the operations of F .
- \sim_s must be compatible with the operations of A_s .

Definition (Simulation):

1. Let $\Sigma \subseteq \Sigma_1$.

A Σ_1 -structure B **simulates identically** a Σ -structure A w.r.t. Rep^B, \sim^B , if

(a) $Rep_s^B \subseteq B_s$ for all $s \in S$,

Definition (Simulation):

1. Let $\Sigma \subseteq \Sigma_1$.

A Σ_1 -structure B **simulates identically** a Σ -structure A w.r.t. Rep^B, \sim^B , if

(a) $Rep_s^B \subseteq B_s$ for all $s \in S$,

(b) \sim_s^B is a Σ -congruence on Rep_s^B for all $s \in S$, and

Definition (Simulation):

1. Let $\Sigma \subseteq \Sigma_1$.

A Σ_1 -structure B **simulates identically** a Σ -structure A w.r.t. Rep^B, \sim^B , if

(a) $Rep_s^B \subseteq B_s$ for all $s \in S$,

(b) \sim_s^B is a Σ -congruence on Rep_s^B for all $s \in S$, and

(c) A is isomorphic to Rep^B / \sim^B whereby

$$Rep^B / \sim^B =_{def} ((Rep_s^B) / \sim_s^B)_{s \in S}.$$

Definition (Simulation):

1. Let $\Sigma \subseteq \Sigma_1$.

A Σ_1 -structure B **simulates identically** a Σ -structure A w.r.t. Rep^B, \sim^B , if

(a) $Rep_s^B \subseteq B_s$ for all $s \in S$,

(b) \sim_s^B is a Σ -congruence on Rep_s^B for all $s \in S$, and

(c) A is isomorphic to Rep^B / \sim^B whereby

$$Rep^B / \sim^B =_{def} ((Rep_s^B) / \sim_s^B)_{s \in S}.$$

2. A Σ_1 -structure B **simulates a Σ -structure A w.r.t. renaming** $\rho : \Sigma \rightarrow \Sigma_1$, Rep^B and \sim^B if

(a) $Rep_s^B \subseteq B_{\rho(s)}$ for all $s \in S$,

Definition (Simulation):

1. Let $\Sigma \subseteq \Sigma_1$.

A Σ_1 -structure B **simulates identically** a Σ -structure A w.r.t. Rep^B, \sim^B , if

(a) $Rep_s^B \subseteq B_s$ for all $s \in S$,

(b) \sim_s^B is a Σ -congruence on Rep_s^B for all $s \in S$, and

(c) A is isomorphic to Rep^B / \sim^B whereby
 $Rep^B / \sim^B =_{def} ((Rep_s^B) / \sim_s^B)_{s \in S}$.

2. A Σ_1 -structure B **simulates a Σ -structure A w.r.t. renaming** $\rho : \Sigma \rightarrow \Sigma_1$, Rep^B and \sim^B if

(a) $Rep_s^B \subseteq B_{\rho(s)}$ for all $s \in S$,

(b) \sim_s^B is a $\rho(\Sigma)$ -congruence on Rep_s^B for all $s \in S$, and

Definition (Simulation):

1. Let $\Sigma \subseteq \Sigma_1$.

A Σ_1 -structure B **simulates identically** a Σ -structure A w.r.t. Rep^B, \sim^B , if

- (a) $Rep_s^B \subseteq B_s$ for all $s \in S$,
- (b) \sim_s^B is a Σ -congruence on Rep_s^B for all $s \in S$, and
- (c) A is isomorphic to Rep^B / \sim^B whereby
 $Rep^B / \sim^B =_{def} ((Rep_s^B) / \sim_s^B)_{s \in S}$.

2. A Σ_1 -structure B **simulates a Σ -structure A w.r.t. renaming** $\rho : \Sigma \rightarrow \Sigma_1$, Rep^B and \sim^B if

- (a) $Rep_s^B \subseteq B_{\rho(s)}$ for all $s \in S$,
- (b) \sim_s^B is a $\rho(\Sigma)$ -congruence on Rep_s^B for all $s \in S$, and
- (c) $A \cong Rep^B / \sim^B$.

Definition (Simulation):

1. Let $\Sigma \subseteq \Sigma_1$.

A Σ_1 -structure B **simulates identically** a Σ -structure A w.r.t. Rep^B, \sim^B , if

(a) $Rep_s^B \subseteq B_s$ for all $s \in S$,

(b) \sim_s^B is a Σ -congruence on Rep_s^B for all $s \in S$, and

(c) A is isomorphic to Rep^B / \sim^B whereby
 $Rep^B / \sim^B =_{def} ((Rep_s^B) / \sim_s^B)_{s \in S}$.

2. A Σ_1 -structure B **simulates a Σ -structure A w.r.t. renaming** $\rho : \Sigma \rightarrow \Sigma_1$, Rep^B and \sim^B if

(a) $Rep_s^B \subseteq B_{\rho(s)}$ for all $s \in S$,

(b) \sim_s^B is a $\rho(\Sigma)$ -congruence on Rep_s^B for all $s \in S$, and

(c) $A \cong Rep^B / \sim^B$.

Therefore every identic simulation is a simulation. W.r.t. the inclusion $in : \Sigma \rightarrow \Sigma_1$.

Example (Lists simulate Sets):

We define the renaming

$$\rho : \text{Sig}(\text{SETNAT}) \rightarrow \text{Sig}(\text{SET_by_LIST})$$

as before by $[\text{Set} \mapsto \text{List}]$, which means: ρ is defined as follows:

$$\rho(\text{Set}) = \text{List} \quad \text{and} \quad \rho(x) = x, \text{ otherwise}$$

The structure \mathbb{N}^* of finite lists simulates the structure of finite sets $P^{fin}(\mathbb{N})$ on natural numbers w.r.t. the renaming ρ in following different ways:

1. Simulation of sets by the structure U of unordered lists.

$$\begin{aligned}
Rep_{\text{Set}}^U &= \mathbb{N}^* \\
Rep_{\text{Nat}}^U &= \mathbb{N} \\
Rep_{\text{Bool}}^U &= \{T, F\} \\
\text{empty}^U &= \epsilon \\
x \text{ in}^U \langle x_1, \dots, x_n \rangle &\Leftrightarrow x = x_i \text{ for some } i \in \{1, \dots, n\} \\
\{x\}^U &= \langle x \rangle \\
\text{add}^U(x, \langle x_1, \dots, x_n \rangle) &= \langle x, x_1, \dots, x_n \rangle \\
\langle x_1, \dots, x_n \rangle \cup^U \langle y_1, \dots, y_m \rangle &= \langle x_1, \dots, x_n, y_1, \dots, y_m \rangle \\
\langle x_1, \dots, x_n \rangle \sim^U \langle y_1, \dots, y_m \rangle &\Leftrightarrow \{x_1, \dots, x_n\} = \{y_1, \dots, y_m\}, \\
&\text{both sequences have the same elements}
\end{aligned}$$

2. Simulation of sets by the structure G of ordered lists.

$$\begin{aligned}
 \text{Rep}_{\text{Set}}^G &= \{ \langle x_1, \dots, x_n \rangle \mid x_1 < \dots < x_n, n \geq 0 \} \\
 \text{empty}^G &= \epsilon \\
 x \text{ in}^G \langle x_1, \dots, x_n \rangle &\Leftrightarrow x = x_i \text{ for some } i \in \{1, \dots, n\} \\
 \{x\}^G &= \langle x \rangle \\
 \text{add}^G(x, \langle x_1, \dots, x_n \rangle) &= \begin{cases} \langle x_1, \dots, x_i, x, x_{i+1}, \dots, x_n \rangle \\ \quad \text{if } x_i < x < x_{i+1} \\ \langle x_1, \dots, x_n \rangle \\ \quad \text{if } x = x_i \text{ for some } i \end{cases} \\
 \langle x_1, \dots, x_n \rangle \cup^G \langle y_1, \dots, y_m \rangle &= \langle z_1, \dots, z_k \rangle \in \text{Rep}_{\text{Set}}^G, \\
 \text{whereby } \{x_1, \dots, x_n, y_1, \dots, y_m\} &= \{z_1, \dots, z_k\} \\
 s_1 \sim^G s_2 &\Leftrightarrow s_1 = s_2
 \end{aligned}$$

3. Simulation of sets through the structure SG of weakly ordered lists.

$$\begin{aligned}
 Rep_{Set}^{SG} &= \{ \langle x_1, \dots, x_n \rangle \mid x_1 \leq \dots \leq x_n, n \geq 0 \} \\
 empty^{SG} &= \epsilon \\
 \{x\}^{SG} &= \langle \langle x \rangle \\
 x \text{ in}^{SG} \langle x_1, \dots, x_n \rangle &\Leftrightarrow x = x_i \text{ for some } i \in \{1, \dots, n\} \\
 add^{SG}(x, \langle x_1, \dots, x_n \rangle) &= \langle x_1, \dots, x_i, x, x_i + 1, \dots, x_n \rangle \text{ if} \\
 &\quad x_i \leq x \leq x_{i+1} \\
 \langle x_1, \dots, x_n \rangle \cup^{SG} \langle y_1, \dots, y_m \rangle &= \langle z_1, \dots, z_k \rangle \in Rep_{Set}^{SG} \\
 &\quad \text{whereby } \langle z_1, \dots, z_k \rangle \text{ is} \\
 &\quad \text{a weakly ordered permutation of} \\
 &\quad \langle x_1, \dots, x_n \rangle ++ \langle y_1, \dots, y_m \rangle \\
 \langle x_1, \dots, x_n \rangle \sim^{SG} \langle y_1, \dots, y_m \rangle &\Leftrightarrow \{x_1, \dots, x_n\} = \{y_1, \dots, y_m\}, \\
 &\quad \text{both sequences have the same elements}
 \end{aligned}$$

Method for constructing simulations

1. Forget: Forget all symbols, that do not stem from $\rho(\Sigma)$
2. Restrict: Restrict the carrier sets to the representing sets Rep_s
3. Identify: Build the quotient w.r.t. \sim_s .

Definition:

A specification SP_1 **FRI-implements** a specification SP w.r.t. a signature morphism $\sigma : \text{Sig}(SP) \rightarrow \text{Sig}(SP_1)$ (write $SP_1 \rightsquigarrow_{\sigma} SP$), if every model B of SP_1 simulates a model of SP w.r.t. suitable Rep^B and \sim^B .

Definition:

A specification SP_1 **FRI-implements** a specification SP w.r.t. a signature morphism $\sigma : \text{Sig}(SP) \rightarrow \text{Sig}(SP_1)$ (write $SP_1 \rightsquigarrow_{\sigma} SP$), if every model B of SP_1 simulates a model of SP w.r.t. suitable Rep^B and \sim^B .

Theorem:

The implementation relationship $\rightsquigarrow_{\sigma}$ is transitive: if $SP_1 \rightsquigarrow_{\sigma_1} SP_2$ and $SP_2 \rightsquigarrow_{\sigma_2} SP_3$ implies $SP_1 \rightsquigarrow_{\sigma_1 \circ \sigma_2} SP_3$.

Example (Specification of "functional" Arrays):

The following specification ARRAY describes Arrays with elements of the type Data over an index type, specified by INDEX.

```

spec ARRAY[INDEX] =
  DATA then
    sorts          Array
    generated type Array ::=  $\omega$  | put(Array; Index; Data)
                    %%{ $\omega$  empty array, put adding an element}%
    ops            $[-]$  : Array  $\times$  Index  $\rightarrow$  Data %%{ direct access}%
  
```

Example (Specification of "functional" Arrays):

The following specification ARRAY describes Arrays with elements of the type Data over an index type, specified by INDEX.

```

spec ARRAY[INDEX] =
  DATA then
    sorts           Array
    generated type Array ::=  $\omega$  | put(Array; Index; Data)
                    %%{ $\omega$  empty array, put adding an element}%
    ops             $[-]$  : Array  $\times$  Index  $\rightarrow$  Data %%{ direct access}%
    vars           $i, j$  : Index;  $x, y$  : Data;  $a$  : Array
    axioms         $\neg$ def  $\omega[j]$ ;
                    put( $a, i, x$ )[ $j$ ] =  $x$  when  $i = j$  else  $a[j]$ ;
  
```

Example (Specification of "functional" Arrays):

The following specification ARRAY describes Arrays with elements of the type Data over an index type, specified by INDEX.

```

spec ARRAY[INDEX] =
  DATA then
    sorts           Array
    generated type Array ::=  $\omega$  | put(Array; Index; Data)
                    %%{ $\omega$  empty array, put adding an element}%
    ops             $[-]$  : Array  $\times$  Index  $\rightarrow$  Data %%{ direct access}%
    vars           $i, j$  : Index;  $x, y$  : Data;  $a$  : Array
    axioms         $\neg$ def  $\omega[j]$ ;
                    put( $a, i, x$ )[ $j$ ] =  $x$  when  $i = j$  else  $a[j]$ ;
                    %%{additionally we may require:}%
                     $i = j \implies$  put(put( $a, i, x$ ),  $j, y$ ) = put( $a, j, y$ );
                     $\neg(i = j) \implies$  put(put( $a, i, x$ ),  $j, y$ ) = put(put( $a, j, y$ ),  $i, x$ )
  end

```

Notice: The specification ARRAY offers only the very necessary functionality for arrays. In a comfortable specification more functionality will be offered.

Example (STACK_by_ARRAYPOINTER):

```
spec STACK_by_ARRAYPOINTER=  
  ARRAY[NAT fit Index  $\mapsto$  Nat] then  
  free type                               Stack ::= pair(Array; Nat)
```

Example (STACK_by_ARRAYPOINTER):

```
spec STACK_by_ARRAYPOINTER=  
  ARRAY[NAT fit Index  $\mapsto$  Nat] then  
  free type          Stack ::= pair(Array; Nat)  
  ops                push : Data  $\times$  Stack  $\rightarrow$  Stack;  
                      empty : Stack;  
                      pop : Stack  $\rightarrow$  Stack;  
                      top : Stack  $\rightarrow$  Data
```

Example (STACK_by_ARRAYPOINTER):

```
spec STACK_by_ARRAYPOINTER=  
  ARRAY[NAT fit Index  $\mapsto$  Nat] then  
  free type          Stack ::= pair(Array; Nat)  
  ops                push : Data  $\times$  Stack  $\rightarrow$  Stack;  
                      empty : Stack;  
                      pop  : Stack  $\rightarrow$  Stack;  
                      top  : Stack  $\rightarrow$  Data  
  
  vars               $x$  : Data;  $i$  : Nat;  $a$  : Array  
  axioms            empty = pair( $\omega$ , 0);
```

Example (STACK_by_ARRAYPOINTER):

spec STACK_by_ARRAYPOINTER=

ARRAY[NAT **fit** Index \mapsto Nat] **then**

free type

Stack ::= pair(Array; Nat)

ops

push : Data \times Stack \rightarrow Stack;

empty : Stack;

pop : Stack \rightarrow Stack;

top : Stack \rightarrow Data

vars

x : Data; i : Nat; a : Array

axioms

empty = pair(ω , 0);

push(x , pair(a , i)) = pair(put(a , i , x), succ(i));

Example (STACK_by_ARRAYPOINTER):

spec STACK_by_ARRAYPOINTER=

ARRAY[NAT **fit** Index \mapsto Nat] **then**

free type

Stack ::= pair(Array; Nat)

ops

push : Data \times Stack \rightarrow Stack;

empty : Stack;

pop : Stack \rightarrow Stack;

top : Stack \rightarrow Data

vars

x : Data; i : Nat; a : Array

axioms

empty = pair(ω , 0);

push(x , pair(a , i)) = pair(put(a , i , x), succ(i));

pop(pair(a , succ(i))) = pair(a , i);

Example (STACK_by_ARRAYPOINTER):

spec STACK_by_ARRAYPOINTER=

ARRAY[NAT **fit** Index \mapsto Nat] **then**

free type

Stack ::= pair(Array; Nat)

ops

push : Data \times Stack \rightarrow Stack;

empty : Stack;

pop : Stack \rightarrow Stack;

top : Stack \rightarrow Data

vars

x : Data; i : Nat; a : Array

axioms

empty = pair(ω , 0);

push(x , pair(a , i)) = pair(put(a , i , x), succ(i));

pop(pair(a , succ(i))) = pair(a , i);

\neg def pop(pair(a , 0));

Example (STACK_by_ARRAYPOINTER):

spec STACK_by_ARRAYPOINTER=

ARRAY[NAT **fit** Index \mapsto Nat] **then**

free type

Stack ::= pair(Array; Nat)

ops

push : Data \times Stack \rightarrow Stack;

empty : Stack;

pop : Stack \rightarrow Stack;

top : Stack \rightarrow Data

vars

x : Data; i : Nat; a : Array

axioms

empty = pair(ω , 0);

push(x , pair(a , i)) = pair(put(a , i , x), succ(i));

pop(pair(a , succ(i))) = pair(a , i);

\neg def pop(pair(a , 0));

top(pair(a , succ(i))) = $a[i]$;

Example (STACK_by_ARRAYPOINTER):

```

spec STACK_by_ARRAYPOINTER=
  ARRAY[NAT fit Index  $\mapsto$  Nat] then
    free type          Stack ::= pair(Array; Nat)
    ops                push : Data  $\times$  Stack  $\rightarrow$  Stack;
                        empty : Stack;
                        pop  : Stack  $\rightarrow$  Stack;
                        top  : Stack  $\rightarrow$  Data
    vars                 $x$  : Data;  $i$  : Nat;  $a$  : Array
    axioms              empty = pair( $\omega$ , 0);
                        push( $x$ , pair( $a$ ,  $i$ )) = pair(put( $a$ ,  $i$ ,  $x$ ), succ( $i$ ));
                        pop(pair( $a$ , succ( $i$ ))) = pair( $a$ ,  $i$ );
                         $\neg$ def pop(pair( $a$ , 0));
                        top(pair( $a$ , succ( $i$ ))) =  $a[i]$ ;
                         $\neg$ def top(pair( $a$ , 0))
  end

```

Theorem:

Let $SP = (\Sigma, E)$ be a flat specification, SP' a specification with $\text{Sig}(SP') \supseteq \Sigma$ and let $\text{Ax}(Rep, \sim)$ be an axiomatisation of the characteristic predicate of Rep and the congruence relation \sim over SP' . Let

$$\text{spec } SP'' = SP' \text{ then } \text{Ax}(Rep, \sim) \text{ end}$$

Then SP' is a FRI-Implementation of SP , if SP'' fulfils the axioms E of SP on Rep modulo \sim , which means:

$$SP'' \models G_{Rep, \sim} \quad \text{for all } G \in E$$

whereby $G_{Rep, \sim}$ is defined inductively by:

$$p(t_1, \dots, t_n)_{Rep, \sim} \equiv p(t_1, \dots, t_n)$$

whereby $G_{Rep, \sim}$ is defined inductively by:

$$\begin{aligned} p(t_1, \dots, t_n)_{Rep, \sim} &\equiv p(t_1, \dots, t_n) \\ (u = v)_{Rep, \sim} &\equiv u \sim v \end{aligned}$$

whereby $G_{Rep, \sim}$ is defined inductively by:

$$\begin{aligned} p(t_1, \dots, t_n)_{Rep, \sim} &\equiv p(t_1, \dots, t_n) \\ (u = v)_{Rep, \sim} &\equiv u \sim v \\ (G_1 \wedge G_2)_{Rep, \sim} &\equiv (G_1)_{Rep, \sim} \wedge (G_2)_{Rep, \sim} \end{aligned}$$

whereby $G_{Rep, \sim}$ is defined inductively by:

$$\begin{aligned} p(t_1, \dots, t_n)_{Rep, \sim} &\equiv p(t_1, \dots, t_n) \\ (u = v)_{Rep, \sim} &\equiv u \sim v \\ (G_1 \wedge G_2)_{Rep, \sim} &\equiv (G_1)_{Rep, \sim} \wedge (G_2)_{Rep, \sim} \\ (\neg G)_{Rep, \sim} &\equiv \neg(G_{Rep, \sim}) \end{aligned}$$

whereby $G_{Rep, \sim}$ is defined inductively by:

$$\begin{aligned} p(t_1, \dots, t_n)_{Rep, \sim} &\equiv p(t_1, \dots, t_n) \\ (u = v)_{Rep, \sim} &\equiv u \sim v \\ (G_1 \wedge G_2)_{Rep, \sim} &\equiv (G_1)_{Rep, \sim} \wedge (G_2)_{Rep, \sim} \\ (\neg G)_{Rep, \sim} &\equiv \neg(G_{Rep, \sim}) \\ (\forall x : s. G)_{Rep, \sim} &\equiv \forall x : s. Rep_s(x) \implies G_{Rep, \sim} \end{aligned}$$

whereby $G_{Rep, \sim}$ is defined inductively by:

$$\begin{aligned}
 p(t_1, \dots, t_n)_{Rep, \sim} &\equiv p(t_1, \dots, t_n) \\
 (u = v)_{Rep, \sim} &\equiv u \sim v \\
 (G_1 \wedge G_2)_{Rep, \sim} &\equiv (G_1)_{Rep, \sim} \wedge (G_2)_{Rep, \sim} \\
 (\neg G)_{Rep, \sim} &\equiv \neg(G_{Rep, \sim}) \\
 (\forall x : s. G)_{Rep, \sim} &\equiv \forall x : s. Rep_s(x) \implies G_{Rep, \sim} \\
 (\exists x : s. G)_{Rep, \sim} &\equiv \exists x : s. Rep_s(x) \wedge G_{Rep, \sim}
 \end{aligned}$$

Example (Implementation-Proof STACK_by_ARRAYPOINTER):

We define axiomatically the characteristic predicate Rep of the representation set and the congruence \sim

over STACK_by_ARRAYPOINTER:

preds $Rep_{Data} : Data;$
 $Rep_{Stack} : Stack;$

Example (Implementation-Proof STACK_by_ARRAYPOINTER):

We define axiomatically the characteristic predicate Rep of the representation set and the congruence \sim

over STACK_by_ARRAYPOINTER:

preds $Rep_{Data} : Data;$
 $Rep_{Stack} : Stack;$
 $\sim_{Data} : Data \times Data;$
 $\sim_{Stack} : Stack \times Stack$

Example (Implementation-Proof STACK_by_ARRAYPOINTER):

We define axiomatically the characteristic predicate Rep of the representation set and the congruence \sim

over STACK_by_ARRAYPOINTER:

```
preds       $Rep_{Data} : Data;$   
             $Rep_{Stack} : Stack;$   
             $\sim_{Data} : Data \times Data;$   
             $\sim_{Stack} : Stack \times Stack$   
  
vars       $x, y : Data; s : Stack; a, b : Array; i, j : Nat$   
  
axioms     $Rep_{Data}(x); \quad \% \% \{ Rep_{Data} \text{ holds for all } x : Data \} \%$   
             $Rep_{Stack}(s) \Leftrightarrow \exists a : Array; i : Nat. s = pair(a, i);$ 
```

Example (Implementation-Proof STACK_by_ARRAYPOINTER):

We define axiomatically the characteristic predicate Rep of the representation set and the congruence \sim

over STACK_by_ARRAYPOINTER:

```

preds       $Rep_{Data} : Data;$ 
               $Rep_{Stack} : Stack;$ 
               $\sim_{Data} : Data \times Data;$ 
               $\sim_{Stack} : Stack \times Stack$ 
vars       $x, y : Data; s : Stack; a, b : Array; i, j : Nat$ 
axioms     $Rep_{Data}(x); \quad \% \% \{ Rep_{Data} \text{ holds for all } x : Data \} \%$ 
               $Rep_{Stack}(s) \Leftrightarrow \exists a : Array; i : Nat. s = pair(a, i);$ 
               $x \sim_{Data} y \Leftrightarrow x = y;$ 
               $pair(a, i) \sim_{Stack} pair(b, j) \Leftrightarrow i = j \wedge \forall k : Nat. k < i \implies a[k] = b[k]$ 

```

Next we show the proof of one of the axioms:

1. STACK_by_ARRAYPOINTER satisfies the axiom [top]

$$\forall x : \text{Data}; s : \text{Stack}. \text{top}(\text{push}(x, s)) = x$$

Next we show the proof of one of the axioms:

1. STACK_by_ARRAYPOINTER satisfies the axiom [top]

$$\forall x : \text{Data}; s : \text{Stack}. \text{top}(\text{push}(x, s)) = x$$

to see this we have to verify $[\text{top}]_{\text{Rep}, \sim}$:

$$\forall x : \text{Data}; s : \text{Stack}. \text{Rep}_{\text{Data}}(x) \wedge \text{Rep}_{\text{Stack}}(s) \Rightarrow \text{top}(\text{push}(x, s)) \sim_{\text{Data}} x$$

Next we show the proof of one of the axioms:

1. STACK_by_ARRAYPOINTER satisfies the axiom [top]

$$\forall x : \text{Data}; s : \text{Stack}. \text{top}(\text{push}(x, s)) = x$$

to see this we have to verify $[\text{top}]_{Rep, \sim}$:

$$\forall x : \text{Data}; s : \text{Stack}. Rep_{\text{Data}}(x) \wedge Rep_{\text{Stack}}(s) \Rightarrow \text{top}(\text{push}(x, s)) \sim_{\text{Data}} x$$

By unfolding the definitions of Rep and \sim and by predicate-logic transformation we get:

$$\forall x : \text{Data}; a : \text{Array}; i : \text{Nat}. \text{top}(\text{push}(x, \text{pair}(a, i))) = x$$

The proof of this formula is done using the axioms of

STACK_by_ARRAYPOINTER:

$$\begin{aligned} & \text{top}(\text{push}(x, \text{pair}(a, i))) \\ = & \text{top}(\text{pair}(\text{put}(a, i, x), \text{succ}(i))) \quad [\text{Def. of push}] \end{aligned}$$

STACK_by_ARRAYPOINTER:

$$\begin{aligned} & \text{top}(\text{push}(x, \text{pair}(a, i))) \\ = & \text{top}(\text{pair}(\text{put}(a, i, x), \text{succ}(i))) && \text{[Def. of push]} \\ = & \text{put}(a, i, x)[i] && \text{[Def. of top]} \end{aligned}$$

STACK_by_ARRAYPOINTER:

$$\begin{aligned} & \text{top}(\text{push}(x, \text{pair}(a, i))) \\ = & \text{top}(\text{pair}(\text{put}(a, i, x), \text{succ}(i))) && [\text{Def. of push}] \\ = & \text{put}(a, i, x)[i] && [\text{Def. of top}] \\ = & x && [\text{Def. of put}(a, i, x)[i] \text{ in ARRAY}] \end{aligned}$$

STACK_by_ARRAYPOINTER:

$$\begin{aligned}
& \text{top}(\text{push}(x, \text{pair}(a, i))) \\
= & \text{top}(\text{pair}(\text{put}(a, i, x), \text{succ}(i))) && [\text{Def. of push}] \\
= & \text{put}(a, i, x)[i] && [\text{Def. of top}] \\
= & x && [\text{Def. of put}(a, i, x)[i] \text{ in ARRAY}]
\end{aligned}$$

Notice: Notice, that the following equation does not hold in the implementation

$$\text{pop}(\text{push}(x, s)) = s$$

instead we have

$$\text{pop}(\text{push}(x, s)) \sim_{\text{Stack}} s$$

A short Introduction to SML

- SML (<http://cm.bell-labs.com/cm/cs/what/smlnj/>) is a functional programming language, that has been developed since 1978 by Robin Milner as meta language for LCF, one of the first interactive proof systems.
SML is now supported by .Net

A short Introduction to SML

- SML (<http://cm.bell-labs.com/cm/cs/what/smlnj/>) is a functional programming language, that has been developed since 1978 by Robin Milner as meta language for LCF, one of the first interactive proof systems.
SML is now supported by .Net
- SML is a strictly typed language with concepts for recursive data types, exceptions, parametric polymorphism and modules.

A short Introduction to SML

- SML (<http://cm.bell-labs.com/cm/cs/what/smlnj/>) is a functional programming language, that has been developed since 1978 by Robin Milner as meta language for LCF, one of the first interactive proof systems.
SML is now supported by .Net
- SML is a strictly typed language with concepts for recursive data types, exceptions, parametric polymorphism and modules.
- For further information see lecture notes.
- Literature:
L. Paulson: SML for the Working Programmer. Cambridge University Press, 2. Auflage 1997

Example (Simple Examples for Modules in SML):

- NUMSIG

```
signature NUMSIG =
```

```
sig
```

```
    type num
```

```
    val add: num * num -> num
```

```
    val mult: num * num -> num
```

```
end;
```

- The structure of integers

```
structure INT:NUMSIG =
```

```
struct
```

```
    type num = int
```

```
    fun add(x,y) = x + y
```

```
    fun mult(x,y) = x * y
```

```
end;
```

- The structure of integers

```
structure INT:NUMSIG =  
  struct  
    type num = int  
    fun add(x,y) = x + y  
    fun mult(x,y) = x * y  
  end;
```

- The structure of rational numbers:

```
structure RAT:NUMSIG =  
  struct  
    type num = int * int  
    fun add((z1, n1), (z2, n2)) = (z1 * n2 + z2 * n1, n1 * n2)  
    fun mult((z1, n1), (z2, n2)) = (z1 * z2, n1 * n2)  
  end;
```

[SOURCE CODE](#)

[RUN EXAMPLE](#)

Transition to Functional Programs

Definition:

Let SP be a specification, where

- every sort s is either defined absolutely free:

$$\text{free type } s ::= c_1 | \dots | c_n(s_1; \dots; s_m)$$

or is a basic data type in SML.

Transition to Functional Programs

Definition:

Let SP be a specification, where

- every sort s is either defined absolutely free:

$$\text{free type } s ::= c_1 | \dots | c_n(s_1; \dots; s_m)$$

or is a basic data type in SML.

- every operation f is defined inductively by axioms like:

$$b(\bar{t}) \implies f(\bar{u}) = e$$

whereby $\bar{t} = (t_1, \dots, t_k)$ and $\bar{u} = (u_1, \dots, u_n)$ are tuples of constructor terms.

Then SP is called in **executable normal form**.

Let SP be a specification in executable normal form. Then SP can be translated into a SML-program in the following way:

- Every sort s with constructors c_1, \dots, c_n is described as data type
datatype $s = c_1 \mid \dots \mid c_n$ of $s_1 * \dots * s_m$

Let SP be a specification in executable normal form. Then SP can be translated into a SML-program in the following way:

- Every sort s with constructors c_1, \dots, c_n is described as data type


```
datatype s = c1 | ... | cn of s1 * ... * sm
```
- Every operation f is defined inductively by pattern matching:


```
fun f(u1) = if b11 then e11 else ...
           | ...
           | f(uk) = if bk1 then ek1 else ...
```

Let SP be a specification in executable normal form. Then SP can be translated into a SML-program in the following way:

- Every sort s with constructors c_1, \dots, c_n is described as data type

$$\text{datatype } s = c_1 \mid \dots \mid c_n \text{ of } s_1 * \dots * s_m$$
- Every operation f is defined inductively by pattern matching:

$$\begin{array}{l} \text{fun } f(u_1) = \text{if } b_{11} \text{ then } e_{11} \text{ else } \dots \\ \quad \mid \dots \\ \quad \mid f(u_k) = \text{if } b_{k1} \text{ then } e_{k1} \text{ else } \dots \end{array}$$
- For every undefinedness axiom one raises an exception.
- Every algebraic signature is translated into the corresponding SML-signature.
- Every specification is translated into an implementable structure.
- A hierarchic specification **spec** $SP = SP_1$ **then** $body$ **end** is translated into an implementation

$$\begin{array}{l} \text{structure } SP_1 = \text{struct } \dots \text{ end} \\ \text{of } SP_1 \text{ and an implementation of } SP \\ \text{structure } SP = \text{struct } \langle \text{Translation of } body \rangle \text{ end} \end{array}$$

Example (Stack):

The specification of stacks

```
spec STACK =  
  ELEM then  
  free type Stack ::= empty|push(Elem; Stack)  
  ops      top : Stack →?Elem;  
          pop : Stack →?Stack  
  
  vars     $x$  : Elem;  $s$  : Stack  
  axioms   $\neg$ def top(empty);  
          top(push( $d$ ,  $s$ )) =  $d$ ;  
           $\neg$ def pop(empty);  
          pop(push( $d$ ,  $s$ )) =  $s$   
  
end
```

is in executable normal form.

Example of SML code generation

First we form the signature:

```
signature STACKSIG =
```

```
sig
```

```
    exception emptyException
```

```
    type 'a stack
```

```
    val empty : 'a stack
```

```
    val push : 'a * 'a stack -> 'a stack
```

```
    val top : 'a stack -> 'a
```

```
    val pop : 'a stack -> 'a stack
```

```
end;
```

Then we construct the structure:

```
structure STACK =
```

```
sig
```

```
  exception emptyException
```

```
  datatype 'a stack = empty | push of 'a * 'a stack
```

```
  fun top(empty) = raise emptyException
```

```
    | top(push(x, s)) = x
```

```
  fun pop(empty) = raise emptyException
```

```
    | pop(push(x, s)) = s
```

```
end
```

SOURCE CODE

RUN EXAMPLE

Summary

- Program development is the process starting with requirements up to constructing an implementation. This process is described formally by a chain of refinements:

$$SP_n \rightsquigarrow SP_{n-1} \rightsquigarrow \dots \rightsquigarrow SP_0$$

with SP_n as requirement specification and SP_0 as solution in an executable form.

- A specification SP' is a **refinement** of SP , if the signatures of SP and SP' are equal and SP' fulfils more properties than SP . A refinement is called **axiomatic enrichment**, if only new axioms are added, and **axiomatic refinement modulo renaming**, if the names in SP' have to be matched to the names of SP .
- If a Σ_1 -algebra B simulates a Σ -algebra A as follows (called **change of data structure**):
Every carrier set of A is represented by a subset Rep of a carrier set of B , and every function symbol of Σ is represented by a function symbol of Σ_1 . Several elements of Rep can represent the same element of A , thus inducing an equivalence relation \sim on Rep . B is a simulation if \sim is a Σ -congruence on Rep and if the quotient algebra Rep/\sim is isomorphic to A .

- A specification SP_1 **FRI-implements** a specification SP w.r.t. a signature morphism ρ , if every model of SP_1 simulates a model of SP w.r.t. suitable Rep and \sim .
- Implementation relationships are proved on the level of specifications. The characteristic predicate of Rep is used for this purpose. A specification SP' **FRI-implements** a specification SP , if Rep and \sim can be defined over SP' in such a way that $E_{Rep, \sim}$ holds in SP' for any axiom E of SP .
- A specification is in executable normal form, if every sort is absolutely free, and if every operation is structurally recursively defined by conditional equations over constructor terms as follows:

$$b(\bar{t}) \implies f(\bar{u}) = e$$

Such specifications can be schematically translated into the functional language SML.