# Formale Techniken in der Software-Entwicklung

## Change of Data Structure

Christian Prehofer

Based on Material from Martin Wirsing

SS 2011

# Goals

- Refinement of data structures and specifications
  - Simulation of data structures
  - FRI-implementations
- Parameterized specifications
  - Signature and theory morphism
  - Parameter passing in Maude

# Change of Data Structures

"Simulation" of a $\Sigma$-structure through a $\Sigma_1$-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 $A_s$.

- **Remark**
  - Often Σ1 has to be extended by definitions of the Σ-operations.
  - In Maude, Rep is usually represented by a subsort.

# Simulation

**Definition(Simulation):**

1. Let $\Sigma \subseteq \Sigma_1$.
   A $\Sigma_1$-structure $B$ simulates identically a $\Sigma$-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 $\Sigma$-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 $\Sigma_1$-structure $B$ simulates a $\Sigma$-structure $A$ w.r.t. renaming $\rho : \Sigma \to \Sigma_1$, $Rep^B$ 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 $in : \Sigma \to \Sigma_1$.

# Example: Sets by lists

- Consider the following signature of sets over natural numbers:

```
Sig(CHAR-SET) =
     including Sig(BOOL) .      including Sig(CHAR) .
     sort Set .
     op mt : Set .
     op {_} : Char -> Set .
     op add : Char Set -> Set .
     op _∪_ : Set Set -> Set .
     op _in_ : Char Set -> Bool .
```

- Let $P_{fin}(A)$ be the standard Sig(`CHAR-SET`)-algebra of finite sets of character symbols.
- Let Sig(SET-by-SEQ) be a signature of lists over character symbols which includes the operations of Sig(`CHAR-SET`); i.e. Sig(SET-by-SEQ) has the form

```
     sorts Char, Seq, … op empty: -> Seq . …
```

- Let ρ: Sig(`CHAR-SET`) -> Sig(SET-by-LIST) defined by

```
     sort Set to Seq .
```

i.e. ρ(`Set`) = `Seq` and ρ(x) = x otherwise .

# Umbenennung von Spezifikationen (Renaming)

- Ein *Signaturmorphismus (Signature Morphism, Renaming)* ist eine Abbildung zwischen Signaturen, bei der die Funktionalität der abgebildeten Funktionssymbole mit der Abbildung der Sorten verträglich ist.

*Seien* $\Sigma = (S, F)$ *und* $\Sigma' = (S', F')$ *Signaturen. Eine Abbildung* $\sigma = (\sigma_{sort}, \sigma_{op})$ *mit*

$$\sigma_{sort} : S \to S' \qquad \sigma_{op} : F \to F'$$

*heißt* Signaturmorphismus *von* $\Sigma$ *nach* $\Sigma'$, *geschrieben* $\sigma : \Sigma \to \Sigma'$, *wenn für alle* $f \in F_{\langle\langle s_1,\ldots,s_n\rangle,s\rangle}$ *gilt*

$$\sigma_{op}(f) : \sigma_{sort}(s_1), \ldots, s_{sort}(s_n) \to \sigma_{sort}(s)$$

*das heißt, wenn die Funktionalität von* $\sigma_{op}(f)$ *mit der Abbildung der Sorten verträglich ist.*

# Signaturmorphismus

**Beispiel** Verträglichkeit von $\rho_{sort}$ und $\rho_{op}$.

Ist etwa

- $\rho_{sort}$(Set) = Seq, $\rho_{sort}$(Char) = Char, $\rho_{sort}$(Bool) = Bool und $\rho_{op}$(in) = in und gilt
- in : Char **Set** -> Bool in der Signatur $\Sigma$,

dann muss in der Bildsignatur gelten

- in : Char **Seq** -> Bool

# Signaturmorphismus in Maude

Seien Signaturen Σ und Σ′ gegeben.

Eine Umbenennung σ (zwischen Zeichen) der Form

sort $s_1$ to q1 .

…

sort $s_k$ to $q_k$ .

op $f_1$ to $g_1$ .

…

op $f_l$ to $g_l$ .

wobei

Sorten von Σ auf Sorten von Σ' und

Funktionszeichen von Σ auf Funktionszeichen von Σ'

(ohne Angabe der Funktionalität)

so abgebildet werden, dass sie verträglich mit der Abbildung der Sorten sind,

ist ein Signaturmorphismus $\sigma : \Sigma \to \Sigma'$ .

# Specification of Finite Sets in Maude

```
fmod CHAR-SET is
   protecting BOOL . protecting STRING .

   sorts Set .        subsorts Char < Set .

   var E E1 : Char . var S : Set .

   op mt : -> Set                  [ctor] .
   op __ : Set  Set  -> Set  [ctor comm assoc id: mt] .
   eq E E = E .

   op _in_ : Char Set -> Bool .
   eq E in (E1 S) = (E == E1) or (E in S) .
  eq E in mt = false .
   op add : Char Set -> Set .
   eq add(E, S) = E S .
. . .
```

# Specification of Finite Lists in Maude

```
fmod CHAR-SEQ is
  protecting BOOL .   protecting STRING .

  sorts Seq NeSeq  .  subsorts Char < NeSeq < Seq .

  op empty : -> Seq                 [ctor] .
  op _;_ : Seq  Seq  -> Seq       [ctor assoc id: empty] .
  op _;_ : NeSeq NeSeq -> NeSeq [ctor assoc id: empty] .

  vars S S1 : Seq .   vars E E1 : Char .

  op _in_ : Char Seq -> Bool .
  eq E in (E1 ; S) = (E == E1) or (E in S) .

  op del1 : Char Seq -> Seq .
  *** del1(E, S) deletes one occurrence of E in S
  op delAll : Char Seq -> Seq .
  *** delAll(E, S) deletes all occurrences of E in S
  . . .
```

# Simulation of Sets by Lists

The structure A* of finite lists simulates the structure of finite sets $P_{fin}(A)$ on character symbols w.r.t. the renaming $\rho$ in the following different ways:

- Let U be the structure (A, A*, $\varepsilon$ …) of (unordered) lists over character symbols.
- Let G be the structure

  $$(A, \{<x_1, \ldots, x_n> \mid x_1 < \ldots < x_n, n >= 0\}, \varepsilon \ldots)$$

  of ordered lists.
- Let SG be the structure

  $$(A, \{<x_1, \ldots, x_n> \mid x_1 <= \ldots <= x_n, n >= 0\}, \varepsilon \ldots)$$

  of weakly ordered lists.

# Simulation of Sets by Unordered Lists

$\text{Rep}^{U}_{\text{Set}} = A^*$

$$
\begin{array}{rcl}
\mathsf{empty}^U & = & \epsilon \\
x \ \mathsf{in}^U \ \langle x_1, \ldots, x_n \rangle & \Leftrightarrow & x = x_i \ \text{for some} \ i \in \{1, \ldots, n\} \\
\{x\}^U & = & \langle x \rangle \\
\mathsf{add}^U(x, \langle x_1, \ldots, x_n \rangle) & = & \langle x, x_1, \ldots, x_n \rangle \\
\langle x_1, \ldots, x_n \rangle \cup^U \langle y_1, \ldots, y_m \rangle & = & \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle \\
\langle x_1, \ldots, x_n \rangle \sim^U \langle y_1, \ldots, y_m \rangle & \Leftrightarrow & \{x_1, \ldots, x_n\} = \{y_1, \ldots, y_m\}, \\
 & & \text{both sequences have the same elements}
\end{array}
$$

# Realisation in Maude

## Extension of CHAR-SEQ by the operations of Set

```
fmod CHAR-SEQ-UNORDERED is
   protecting BOOL .   protecting CHAR-SEQ .
   var E E1 : Char.    var S S1 : Seq .

   op mt : -> Seq .
   eq mt = empty .

   op __ : Seq Seq -> Seq  .
   op __ : NeSeq NeSeq -> NeSeq  .
   eq S S1 = S ; S1 .

   op add : Char Seq -> NeSeq .
   eq add(E, S) = E ; S .

   op delete : Char Seq -> Seq .
   eq delete(E, S) =  delAll(E, S) .
 endfm
```

# Realisation in Maude

## Definition of Rep and ~

```
fmod CHAR-SET-BY-UnOSEQ is
    protecting CHAR-SEQ-UNORDERED .
    sort Rep .          subsort Rep < Seq .

    var E : Char .      var S S1 : Seq .        var R R1 : Seq .

    op _sub_ : Seq Seq -> Bool .       *** Auxiliary operation
    eq empty sub S  = true .
    eq E ; S sub S1 = (E in S1) and (S sub S1) .

    mb S : Rep .

    op _~_ : Seq Seq -> Bool .
    eq R ~ R1 = (R sub R1) and (R1 sub R) .
 endfm
```

Then the initial algebra U of `CHAR-SET-UNORDERED` simulates $P_{fin}(A)$ w.r.t.
ρ : Sig(CHAR-SET) -> Sig(CHAR-SEQ-UNORDERED), sort Set to Seq,
$Rep^U$, $\sim^U$

# Simulation of Sets by Ordered Lists

$$
\begin{array}{rcl}
Rep^G_{\mathsf{Set}} & = & \{\langle x_1, \ldots, x_n\rangle | x_1 < \ldots < x_n, n \geq 0\} \\
\mathsf{empty}^G & = & \epsilon \\
x \ \mathsf{in}^G \ \langle x_1, \ldots, x_n\rangle & \Leftrightarrow & x = x_i \text{ for some } i \in \{1, \ldots, n\} \\
\{x\}^G & = & \langle x\rangle \\
\mathsf{add}^G(x, \langle x_1, \ldots, x_n\rangle) & = & \begin{cases} \langle x_1, \ldots, x_i, x, x_{i+1}, \ldots, x_n\rangle \\ \quad \text{if } x_i < x < x_{i+1} \\ \langle x_1, \ldots, x_n\rangle \\ \quad \text{if } x = x_i \text{ for some } i \end{cases} \\
\langle x_1, \ldots, x_n\rangle \cup^G \langle y_1, \ldots, y_m\rangle & = & \langle z_1, \ldots, z_k\rangle \in Rep^G_{\mathsf{Set}}, \\
\text{whereby } \{x_1, \ldots, x_n, y_1, \ldots, y_m\} & = & \{z_1, \ldots, z_k\}
\end{array}
$$

$$
s_1 \sim^G s_2 \quad \Leftrightarrow \quad s_1 = s_2
$$

# Realisation in Maude

- The specification of ordered lists in Maude is also based on CHAR-SEQ.
- CHAR-SEQ is extended by a boolean function isOrdered and auxiliary operations for inserting and merging elements:

```
fmod CHAR-SEQ-ORDERED is
  protecting BOOL .  protecting CHAR-SEQ .
  var E E1 : Char .  var S S1 : Seq .    var NeS : NeSeq .

  op isOrdered :  Seq -> Bool .
  eq isOrdered(empty) = true .
  eq isOrdered(E) = true .
  eq isOrdered(E ; E1 ; S) = (E < E1) and isOrdered(E1 ; S) .

  op insert : Char Seq -> Seq .
  eq insert(E, empty) = E .
  eq insert(E, E ; S) = E ; S .
 ceq insert(E, E1 ; S) = E ; E1 ; S if (E < E1) .
  eq insert(E, E1 ; S) = E1 ; insert(E, S) [owise] .

  op merge : Seq  Seq  -> Seq .
  eq merge(empty, S) = S .
  eq merge(S1 ; E, S) = merge(S1, insert(E, S))  .
```

# Realisation in Maude

## Definition of Rep and ~

```
fmod CHAR-SET-BY-OSEQ is
   protecting CHAR-SEQ-ORDERED .
   sort Rep .            subsort Rep < Seq .

   var E E1 E2 : Char .          var S S1 S2 : Seq .

   cmb S : Rep if isOrdered(S)= true .
   op _~_ : Rep Rep -> Bool .
   eq empty ~ empty = true .
   eq (E ; S) ~ empty = false .
   eq empty ~ (E ; S) = false .
   eq (E1 ; S1) ~ (E2 ; S2) = (E1 == E2) and (S1 ~ S2) .
 endfm
```

Then the initial algebra G of CHAR-SEQ-ORDERED simulates $P_{fin}(A)$ w.r.t.

$\rho$ : Sig(CHAR-SET) -> Sig(CHAR-SEQ-ORDERED), sort Set to Seq,

$Rep^G$, $\sim^G$

# Simulation of Sets by Weakly Ordered Lists

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

# Constructing Simulations

The Forget-Restrict-Identify 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. $\sim_s$.

- **Remark**
  - Often $\Sigma 1$ has to be extended by definitions of the $\Sigma$-operations. The all sorts and operations not occurring in $\Sigma$ are forgotten.
  - In Maude, Rep is usually represented by a subsort.

# FRI-Implementation

**Definition:**

A specification $SP_1$ FRI-implements a specification $SP$ w.r.t. a signature morphism $\sigma : \mathsf{Sig}(SP) \rightarrow \mathsf{Sig}(SP_1)$ (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 $\sim^B$.

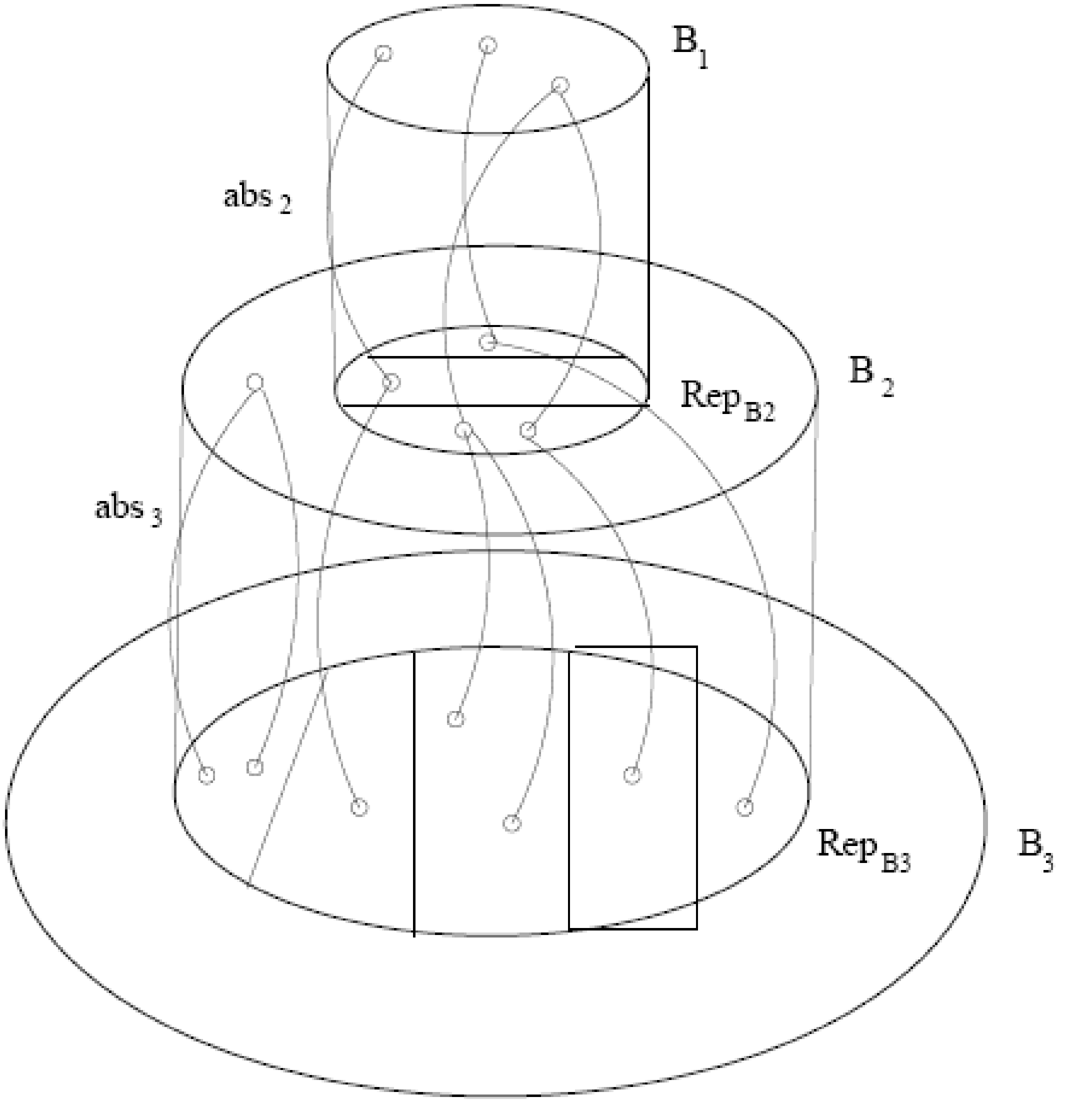
- Examples
  - `CHAR-SEQ-ORDERED` FRI-implements `CHAR-SET w.r.t`
    - `ρ:Sig(CHAR-SET) -> CHAR-SEQ-ORDERED), sorts Set to Seq` and
    - Rep and ~ as defined by `CHAR-SET-BY-OSEQ`

**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 $SP_1 \rightsquigarrow_{\sigma_1 \circ \sigma_2} SP_3$.

# FRI-Implementation

- Proof Idea



# Parameterized Specifications

- Many standard data structures such as sets, lists and stacks can be seen as parameterized structures where the data structure of elements plays the role of formal parameter.

# Example: Parameterized Stacks

```
fmod STACK{X :: TRIV} is
  sorts Stack{X} NeStack{X} .
  subsort NeStack{X} < Stack{X} .

  op empty : -> Stack{X} [ctor] .
  op push : X$Elt Stack{X} -> NeStack{X} [ctor] .
  op pop : NeStack{X} -> Stack{X} .
  op top : NeStack{X} -> X$Elt  .


  var E : X$Elt .
  var S : Stack{X} .

  eq top(push(E, S)) = E .
  eq pop(push(E, S)) = S .
endfm
```

**Parametrisierte Sorten**

**Qualifizierte Sorte des formalen Parameters**

# Parametrisierte Spezifikationen

Eine **parametrisierte Spezifikation** (oder generische Spezifikation) hat die Form

```
fmod SN{SP1, …, SPk} is
     Body
endfm
```

wobei

SN der Name der parametrisierten Spezifikation,

SP1, … , SPk die Namen der formalen Parameter**theorien** und

Body der Rumpf der Spezifikation ist.

SN ist wohldefiniert, wenn der Rumpf die formalen Parameter erweitert, d.h. wenn

including SP1 . … including SPk . Body

wohldefiniert ist .

# Parameterübergabe

Zur Parameterübergabe muss der formale Parameter mit dem aktuellen Parameter in Beziehung gebracht werden:

- Die Signatur des formalen Parameters muss in die Signatur des aktuellen Parameters umbenannt werden und
- der aktuelle Parameter muss die Anforderungen des formalen Parameters erfüllen.

# Theoriemorphismus

Eine aktuelle Parameterspezifikation SP2 ist korrekt bzgl. einer formalen Parametertheorie SP1 , wenn SP2 alle Eigenschaften (Axiome) von SP1 erfüllt:

Ein Theoriemorphismus $\alpha$: SP1 $\rightarrow$ SP2 erhält die Theorie von SP1;

**Definition (Theoriemorphismus**):

Ein **Theoriemorphismus** $\alpha$: SP1 $\rightarrow$ SP2 ist ein

Signaturmorphismus $\alpha$: sig(SP1) $\rightarrow$ sig(SP2),

so dass für jedes Modell M $\in$ Mod(SP2) gilt:

$M|_\alpha \in$ Mod(SP1);

d.h.

SP2 erfüllt alle Axiome von SP1 (modulo Umbenennung).

# Sicht (View)

Jeder Theoriemorphismus $\alpha$: SP1 $\rightarrow$ SP2 induziert eine Sicht von SP1 nach SP2:

**Definition (Sicht in Maude)**

Sei $\alpha$: SP1 $\rightarrow$ SP2 ein **Theoriemorphismus**.

Dann ist

```
view SM from SP1 to SP2 is α endv
```

eine **Sicht** in Maude.

# Sicht (View)

**Beispiel**: Eine Sicht von TRIV nach STRING

```
view Char from TRIV to STRING is
      sort Elt to Char .
endv
```

wobei

```
fth TRIV is
      sort Elt .
endfth
fmod STRING is
   sort Char .
   sort String .
   . . .
 endfm
```

# Parameterübergabe

**Beispiel:**

- Instantiierung von `STACK{X :: TRIV}` mit der Spezifikation STRING .

  Die Sorte `Elt` wird in `Char` umbenannt, d.h. der formale Parameter von `STACK` muss eine Sicht auf `STRING` besitzen.

```
fmod CHAR-STACK is
  including STACK{Char} .
endfm
```

- Instantiierung von `STACK{X :: TRIV}` mit der Spezifikation NATURAL .

```
view Natural from TRIV to NATURAL is
  sort Elt to Natural .
endv
fmod NAT-STACK is
  including STACK{Natural} .
endfm
```

# Parameterized Finite Maps

```
fmod MAP{X :: TRIV, Y :: TRIV} is
  sorts Entry{X,Y} Map{X,Y} .
  subsort Entry{X,Y} < Map{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Map{X,Y} [ctor] .
  op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y}
       [ctor assoc comm id: empty prec 121] .
  op undefined : -> [Y$Elt] [ctor] .
    . . .
  var M : Map{X,Y} . var D : X$Elt . var R : Y$Elt .
  op _[_] : Map{X,Y} X$Elt -> [Y$Elt] .
  ceq (M, D |-> R)[D] = R if $hasMapping(M, D) = false .
  eq M[D] = undefined [owise] .
```

# Parameterized Finite Maps

```
MAP{X :: TRIV, Y :: TRIV} continued:

  op $hasMapping : Map{X,Y} X$Elt -> Bool .
  eq $hasMapping((M, D |-> R), D) = true .
  eq $hasMapping(M, D) = false [owise] .

  op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
  eq insert(D, R, (M, D |-> R')) =
     if $hasMapping(M, D) then insert(D, R, M)
     else (M, D |-> R)
     fi .
  eq insert(D, R, M) = (M, D |-> R) [owise] .
endfm
```

# Example: Implementing Stacks by Finite Maps (Arrays)

- **Simulating stacks over finite maps indexed by nat. numbers :**

```
fmod STACK-BY-MAP{X :: TRIV} is
  protecting MAP{Natural, X} .
  sorts Stack{X} NeStack{X} .
  subsort NeStack{X} < Stack{X} .
  op pair : Map{Natural, X} Natural -> Stack{X} [ctor] .
  op emptyStack : -> Stack{X} .
  op push : X$Elt Stack{X} -> NeStack{X} .
  op pop : NeStack{X} -> Stack{X} .
  op top : NeStack{X} -> X$Elt  .

  var E : X$Elt . var I : Natural .
  var M : Map{Natural, X} .
  eq emptyStack = pair(empty, 0) .
  eq push(E, pair(M, I)) = pair(insert(I, E, M), s I) .
  eq top(pair(M, s I)) = M[I] .
  eq pop(pair(M, s I)) = pair(M, I) .
endfm
```

# Example: Implementing Stacks by Finite Maps (Arrays)

- Instantiating maps by character symbols:

```
fmod CHAR-STACK-BY-MAP is
  including STACK-BY-MAP{Char} .
endfm

red in CHAR-STACK-BY-MAP :
  top(push("a", push("b", emptyStack))) .
red in CHAR-STACK-BY-MAP :
  top(pop(push("a", push("b", emptyStack)))) .
```

# Example: Implementing Stacks by Finite Maps (Arrays)

**Theorem**

`STACK-BY-MAP{X :: TRIV}` is an FRI-implementation of `STACK{X :: TRIV}`.

**Proof:**

Let M0 be any model of `STACK-BY-MAP` and restrict it to the signature of STACK:

$$M = M0|_{sig(STACK\{X :: TRIV\})}$$

Define the following representation set and congruence:

$Rep^M_{Elt} = M_{Elt}$

$Rep^M_{Stack\{X\}} =$

$\{pair^M(m, i) \mid m \in M_{Map}; i <= |m|+1$ and $\forall k : Natural: k < i => m[k]^M$ is defined$\}$

$pair^M(m_1, i) \sim_M pair^M(m_2, j)$ iff

$i = j \wedge \forall k : Natural: k < i => m_1[k]^M = m_2[k]^M$

# Example: Maude Representation

```
fmod STACK-REP{X :: TRIV} is
 protecting STACK-BY-MAP{X} .
 sort Rep{X} .       subsort Rep{X} < Stack{X} .
 op welldefined : Stack{X} -> Bool .
 op equ : Map{Natural, X} Map{Natural, X} Natural -> Bool .
 op _~_ : Stack{X} Stack{X} -> Bool .

 var E : X$Elt .                   var I J : Natural .
 var M M1 M2 : Map{Natural, X} .  var S : Stack{X} .

 cmb S : Rep{X} if welldefined(S) .
 eq welldefined(pair(M, 0)) = true .
 eq welldefined(pair(M, s I)) =
       welldefined(pair(M, I)) and $hasMapping(M, I) .
 eq equ(M1, M2, 0) = true .
 eq equ(M1, M2, s I) = equ(M1, M2, I) and (M1[I] == M2[I]) .
 eq pair(M1,I) ~ pair(M2,J) = (I == J) and equ(M1, M2, I) .
endfm
```

# Proof of the Theorem

- The quotient M′ = $Rep_M/\sim_M$, is a well-defined sig(STACK{X :: TRIV}) algebra which is generated by emptyStack and push.
- The two STACK axioms hold in M′ :

**1. M′ |= `top(push(x, pair(m, i))) = x`:**

Let v be any valuation with elements of M′ . Then

M′,v |= `top(push(x, pair(m, i)))` = [Def. of push]

`top(pair(insert(i, x, m), s i))` = [Def. of top, insert]

`m[i]` = [Def. of _[_] in MAP]

`x`

**2. M′ |= `pop(push(x, pair(m, i))) = pair(m, i)`:**

Let v be any valuation with elements of M′ . Then

M′,v |= `pop(push(x, pair(m, i)))` = [Def. of push]

`pop(pair(insert(i, x, m), s i))` = [Def. of pop]

`pair(insert(i, x, m), i)` =

[ for all `k < i : insert(i,x,m)[k] = m[k]` ]

~ `pair(m, i)` q.e.d

**Theorem:**

Let SP = (Σ,E) be a functional specification,
SP' a specification with $\Sigma \subset Sig(SP')$ and let
Ax(Rep, ~) be an axiomatisation of

- a characteristic predicate Rep and
- a Σ-congruence relation ~ over SP'.

Let

```
fmod SP" is
        protecting SP' .
        Ax(Rep,~) .
endfm
```

Then SP' is a FRI-Implementation of SP, if

- Rep/~ is freely generated by the Σ-constructors of SP'
- SP'' fulfils the axioms E of SP on Rep/~ , i.e.

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

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

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

# Example: Implementing Stacks by Finite Maps (Arrays)

We define axiomatically the characteristic predicate Rep of the representation set and the congruence ~ over STACK-by-MAP:

```
RepElt : X$Elt -> Bool .
RepStack{X} : Stack{X} -> Bool .
_~Elt_: X$Elt X$Elt -> Bool .
_~Stack{X}_: Stack{X} Stack{X} -> Bool .
vars E, E' : X$Elt . var St : Stack{X} .

vars M, M' : Map{Nat, X} . vars  I, J : Natural .
eq RepElt(E) = true . ***RepElt holds for all E ∈ X$Elt
eq RepStack{X}(pair(M, I)) = true
     if I <= |M| + 1 and
        forall(k : Natural. k < I => M[k] >= 0) .
eq  E ~Elt E' = (E == E').
eq  pair(M, I) ~Stack{X} pair(M', J)  =
     I == J and
     forall(k : Natural . k < I => M[k] == M'[k]) .
```

Then we can prove the STACK axioms as follows:

**1.** `∀ E: X$Elt . ∀ St : Stack{X} . top(push(E, St)) = E :`

Relativization w.r.t. Rep and ~ yields

```
∀ E: X$Elt . ∀ St : Stack{X} .
    Rep_Elt(E) and Rep_Stack{X}(St) => top(push(E, St))) ~_Elt E .
```

By the definitions of Rep and ~ we get:

```
∀ E: X$Elt . ∀ M : Map{Nat, X} . ∀ I : Natural .
    (I <= |M|+1) => top(push(E, pair(M, I))) = E .
```

We prove this by the axioms of STACK-by-MAP:

```
top(push(E, pair(M, I))) =                 [Def. of push]
top(pair(insert(I, E, M), s I)) =          [Def. of top, insert]
M[I] =                                     [Def. of _[_] in MAP]
E
```

**2. ∀ E: X$Elt . ∀ St : Stack{X} . pop(push(E, St))) = St:**

Relativization w.r.t. Rep and ~ yields

∀ E: X$Elt . ∀ St : Stack{X} .

$\text{Rep}_{\text{Elt}}$(E) and $\text{Rep}_{\text{Stack\{X\}}}$(St) => pop(push(E, St))) $\sim_{\text{Stack\{X\}}}$ St

By the definition of Rep we get:

∀ E: X$Elt . ∀ M : Map{Nat, X} . ∀ I : Natural .

(I <= |M|+1) =>

pop(push(E, pair(M, I))) $\sim_{\text{Stack\{X\}}}$ pair(M, I))) .

We prove this by the axioms of STACK-by-MAP:

pop(push(E, pair(M, I))) = [Def. of push]

pop(pair(insert(I, E, M), s I)) = [Def. of pop]

pair(insert(I, E, M), I) $\sim_{\text{Stack\{X\}}}$

[for all k < I : insert(I,E,M)[k] = M[k] ]

pair(M, I) q.e.d.

# Summary (I)

- If a $\Sigma_1$-algebra $B$ simulates a $\Sigma$-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 $\Sigma$ is represented by a function symbol of $\Sigma_1$. Several elements of $Rep$ can represent the same element of $A$, thus inducing an equivalence relation $\sim$
- A specification $SP_1$ FRI-implements a specification $SP$ w.r.t. a signature morphism $\rho$ , 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 $SP$.

# Summary (II)

- Maude unterstützt Strukturierung von Spezifikationen durch Umbenennung und Parametrisierung.
- Eine parametrisierte Spezifikation hat Theorien als formale Parameter.
- Ein aktueller Parameter SPA muss (modulo Umbenennung) die Signatur des formalen Parameters T enthalten und alle Eigenschaften von T erfüllen, d.h. es muss einen Theoriemorphismus von T nach SPA geben.