Algebraic

Specification

Axel Poigné
Topics

- Why modules
- OBJ - specifications
- Animation by rewriting
- Correctness criteria

---------------------------

- Parameterization
- Realization and Implementation
- Extensions of Algebra
What is a Module?

<table>
<thead>
<tr>
<th>SPEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
</tr>
<tr>
<td>Axioms</td>
</tr>
</tbody>
</table>

**Signature** - comprises the information of what syntactically given

e.g. conventionally "sorts", "terms", "formulas"

**Axioms** - state what is "true"

**Pragmatics** - can be implemented independently
Why Modules?

A Program System is

- A BIG PROGRAM

- A set of small programs = modules connected by interfaces
A conventional view of interfaces

I = record a : array [1..20] of integer;
   p : integer
end

CRITICISM

- Too concrete
  (the interface may be thought of as a stack)
- Global changes necessary
  (X & Y change to a "deque"
  = record a : array [1..20] of integer;
     p, p' : integer
  end)
- Inhomogeneous structure
  (programs & interfaces)
BETTER!

• homogeneous: Interfaces are modules as any other element in the system

• abstract

• changes can be kept local
  (e.g. Overflow check can be done once and for all in stack)
Modularisation allows local change

The "interface module" STACK is retained for all modules except for X and Y.
The "interface module" STACK is replaced by DEQUE, or

STACK is "implemented" on DEQUE
The lecture analyzes the concept of a module, the nature of relations between modules and of operations on modules.

We use OBJ2 as a specific specification language.

However

- the arguments often apply to other specification methods and provide a rationale for investigation of such methods
- several of the concepts can be formalized on a rather abstract level*

---

OBJ2 - MODULES (theories)

\textit{th INTEGER is}
\begin{itemize}
  \item \texttt{sorts} Integer \hspace{1cm} | \hspace{1cm} |
  \item \texttt{op} 0 : \rightarrow \texttt{Integer} \hspace{1cm} |
  \item \texttt{op} suc : \texttt{Integer} \rightarrow \texttt{Integer} \hspace{1cm} |
  \item \texttt{op} pred : \texttt{Integer} \rightarrow \texttt{Integer} \hspace{1cm} |
  \item \texttt{var} M : \texttt{Integer} \hspace{1cm} |
  \item \texttt{eq}: pred(suc(M)) = M \hspace{1cm} |
  \item \texttt{eq}: suc(pred(M)) = M \hspace{1cm} |
\end{itemize}

\textit{endth}

\textit{th STACK* is}
\begin{itemize}
  \item \texttt{using} INTEGER \hspace{1cm} |
  \item \texttt{sort} Stack \hspace{1cm} |
  \item \texttt{op} empty : \rightarrow \texttt{Stack} \hspace{1cm} |
  \item \texttt{op} push : Stack Integer \rightarrow \texttt{Stack} \hspace{1cm} |
  \item \texttt{op} pop : Stack \rightarrow \texttt{Stack} \hspace{1cm} |
  \item \texttt{op} top : Stack \rightarrow \texttt{Integer} \hspace{1cm} |
  \item \texttt{var} S : Stack, M : Integer \hspace{1cm} |
  \item \texttt{eq}: pop(push(S,M)) = S \hspace{1cm} |
  \item \texttt{eq}: top(push(S,M)) = M \hspace{1cm} |
\end{itemize}

\textit{endth}

\* In spite of a widespread misunderstanding; stacks are in general not used to demonstrate the advantages of algebraic specification but rather the \textbf{disadvantages}. 
th DEQUE is
eusing STACK

op pushtail : Stack Integer → Stack
op poptail : Stack → Stack
op toptail : Stack → Integer
var S : Stack, M, M' : Integer
eq: pushtail(empty, M) = push(empty, M)
eq: pushtail(push(S, M), M') = push(pushtail(S, M'), M)
eq: poptail(push(empty, M)) = empty
eq: poptail(push(push(S, M), M')) = 
               push(poptail(push(S, M)), M')
eq: toptail(push(empty, M)) = M
eq: toptail(push(push(S, M), M')) = 
               push(toptail(push(S, M)), M')
endth

DEQUE "uses" STACK !!!

STACK can be "implemented" by DEQUE !!!
Exercises

(A) Sets of integers come along with an empty set, one-point sets, a union and a membership operator.

Give an equational axiomatization.

(B) What are the typical operations of an array

Give the signature, and some equations
Other Specification Methods : A Sample

"PROLOG-GT* -Module"

sorts integer, stack
const empty : \rightarrow stack
    push : stack integer \rightarrow stack
rel   pop : stack stack
    top : stack integer
axioms pop(push(s,d),s)
    top(push(s,d),d)

"VDM - SL - Module"

data sorts data, stack
funct empty : \rightarrow stack
    push : stack data \rightarrow stack
state machine
    state s : stack
ops pop : 1 \rightarrow data
axioms pre(pop) = \{ s \neq empty \}
    post(pop) = \{ \forall s' : stack, d' : data .
                   s_0 = push(s',d) \Rightarrow
                   s = s' \land d = d' \}

* All our notation only faintly resemble any formalism we allude to
RELATIONAL DATABASE

**signature**
- **sorts** name, address, price, size
- **relators** persons : name \times address
  - houses : address \times price \times size

**frames**
- join(persons, houses)
- project(persons, name)

**Model**

<table>
<thead>
<tr>
<th>persons</th>
<th>houses</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>address</td>
</tr>
<tr>
<td>Tom</td>
<td>Rye 66</td>
</tr>
<tr>
<td>Mary</td>
<td>Rye 66</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Evaluation**

\[
\text{Val}(\text{join}(\text{persons}, \text{houses})) = \\
\begin{array}{c|c|c|c|c}
\text{name} & \text{address} & \text{address} & \text{price} & \text{size} \\
\hline
\text{Tom} & \text{Rye 66} & \text{Rye 66} & 150000 & \text{big} \\
\text{Mary} & \text{Rye 66} & \text{Rye 66} & 150000 & \text{big} \\
\end{array}
\]

\[
\text{Val}(\text{project}(\text{persons}, \text{name})) = \begin{array}{c}
\text{Tom} \\
\text{Mary}
\end{array}
\]
MEANING = SEMANTICS

is a matter of taste and conviction.

Some alternatives
• model based,
  - one model, e.g. initial-Herbrand, final
  - class of models
  - theory of a class of models
    (algebraic specifications)

• proof-theoretic

  (logical specification)
IN CASE OF ALGEBRAIC SPECIFICATIONS

Loose Semantics

= all models which satisfy the axioms

This is the semantics for OBJ2 theories, i.e.

th XYZ is

...

end th

Initial Semantics

= initial algebra

(terms modulo all closed equations, e.g.

pred(suc(0)) = 0, pred(pred(suc(0))) = pred(0) )

The semantics of OBJ2 objects
OBJ2 objects

\textit{obj} INTEGER is

\textit{sorts} Integer

\textit{op} 0 : \rightarrow \text{Integer}

\textit{op} \text{suc} : \text{Integer} \rightarrow \text{Integer}

\textit{op} \text{pred} : \text{Integer} \rightarrow \text{Integer}

\textit{var} M : \text{Integer}

\textit{eq:} \quad \text{pred}(\text{suc}(M)) = M

\textit{eq:} \quad \text{suc}(\text{pred}(M)) = M

\textit{jbo}

Objects have initial algebra semantics

Objects are animated by
Rewriting

"Compute" equations in one direction only

\[ \text{pred(suc(M)) } \rightarrow \text{M} \]
\[ \text{suc(pred(M)) } \rightarrow \text{M} \]

\[ \text{pred(suc(pred(pred(suc(0))))) } \rightarrow \text{pred(0)} \]

Rewrite rules are used "inside terms"
Problems

• Does this always terminate?
  = STRONG NORMALIZATION
    (yes in the example)

• Is evaluation independent of the chosen strategy?
  = CHURCH-ROSSER PROPERTY

Claim (OBJ2):
'Experienced programmers usually write rules which satisfy these properties'
Πραγματικά

Distinguish Constructors (which generate the data)
Define the other operators by case analysis

\[
\begin{align*}
\text{add}(M, 0) & \rightarrow M \\
\text{add}(M, \text{suc}(N)) & \rightarrow \text{suc}(\text{add}(M, N)) \\
\text{add}(M, \text{pred}(N)) & \rightarrow \text{pred}(\text{add}(M, N))
\end{align*}
\]

Q What are the constructors for STACK?
Something wrong?
A Nasty One

\[\begin{align*}
\text{obj } & \text{SET } \text{is} \\
\text{extending } & \text{INTEGER, BOOL} \\
\text{sort } & \text{Set} \\
\text{op } & \emptyset : \rightarrow \text{Set} \\
\text{op } & \_ \cup \_: \text{Set Set} \rightarrow \text{Set} \\
\text{op } & \{\_\} : \text{Integer} \rightarrow \text{Set} \\
\text{op } & \_ \epsilon \_: \text{Integer Set} \rightarrow \text{Bool} \\
\text{var } & S, S', S\'' : \text{Set, M, N : Integer} \\
\text{eq } & S \cup (S' \cup S\'\') = (S \cup S') \cup S\'' \\
\text{eq } & S \cup S' = S' \cup S \\
\text{eq } & S \cup \emptyset = S \\
\text{eq } & M \epsilon \{M\} \cup S = \text{true} \\
\text{eq } & \{M\} \epsilon \emptyset = \text{false} \\
\end{align*}\]

Why is this object unpleasant?
Evaluation

\{0\} \cup \{\text{suc}(0)\}
\rightarrow \{\text{suc}(0)\} \cup \{0\}
\rightarrow \{0\} \cup \{\text{suc}(0)\}
\rightarrow \ldots \quad \text{may not terminate}

\{0\} \cup \{0\}
\rightarrow \{0\} \quad \text{may terminate or not}

\{(0) \cup \{\text{suc}(0)\} \cup \{\text{suc}^2(0)\}
\rightarrow \{0\} \cup (\{\text{suc}(0)\} \cup \{\text{suc}^2(0)\})
\rightarrow (\{\text{suc}(0)\} \cup \{\text{suc}^2(0)\}) \cup \{0\}
\rightarrow \{\text{suc}(0)\} \cup (\{\text{suc}^2(0)\} \cup \{0\})
\rightarrow \ldots

\text{Termination is hard to check}

\text{suc}(0) \in \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{add}(0,\text{suc}(0))\})
\rightarrow \text{suc}(0) \in \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{suc}(\text{add}(0,0))\})
\rightarrow \text{suc}(0) \in \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{suc}(0)\})
\rightarrow \text{suc}(0) \in \{\text{pred}(0)\} \cup (\{\text{suc}(0)\} \cup \{0\})
\rightarrow \text{suc}(0) \in (\{\text{suc}(0)\} \cup \{0\}) \cup \{\text{pred}(0)\}
\rightarrow \text{suc}(0) \in \{\text{suc}(0)\} \cup (\{0\} \cup \{\text{pred}(0)\})
\rightarrow \text{true}

\text{Mixture of "real" and "organisational computation}
AC - Rewriting

\[ t \xrightarrow{\text{AC}} t' : \iff t \xrightarrow{\text{AC}} t'' \rightarrow t' \]

\{0\} \cup \{0\} \rightarrow \{0\}

\text{suc}(0) \in \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{add}(0,\text{suc}(0))\})

\rightarrow \text{suc}(0) \in \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{suc}(\text{add}(0,0))\})

\rightarrow \text{suc}(0) \in \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{suc}(0)\})

\equiv \text{suc}(0) \in \{\text{pred}(0)\} \cup (\{\text{suc}(0)\} \cup \{0\})

\equiv \text{suc}(0) \in (\{\text{suc}(0)\} \cup \{0\}) \cup \{\text{pred}(0)\}

\equiv \text{suc}(0) \in \{\text{suc}(0)\} \cup (\{0\} \cup \{\text{pred}(0)\})

\rightarrow \text{true}

\textbf{Important}

Only finite number of terms which are equal modulo AC

(gives a clue how to generalize to rewriting "mod E")
Exercise

• How to specify that an integer is not element of a set?

• "Cardinality" is specified by
  \[ \text{card}(\emptyset) = 0 \]
  \[ \text{card}({M} \cup S) = \text{suc}(\text{card}(S)) \]
  Wright or Wrong ???

• Delete an element from a set
Normal Form

Strong Normalization $\land$ Church Rosser

$\Rightarrow$ Unique Normal Forms
Contradiction
More Theory

Observation

\[
\text{suc}(\text{pred}(0)) = \text{pred}(\text{suc}(\text{suc}(\text{pred}(0))))) = \text{pred}(\text{suc}(0))
\]

\[
\downarrow
\]

\[
\text{suc}(\text{pred}(0)) \leftarrow \text{pred}(\text{suc}(\text{suc}(\text{pred}(0))))) \rightarrow \text{pred}(\text{suc}(0))
\]

Any sequence of equation can be translated in a sequence of rewritings (not necessarily in the same direction). Then

\[
\begin{array}{c}
\text{pred}(\text{suc}(\text{suc}(\text{pred}(0)))) \\
\text{suc}(\text{pred}(0)) \\
0 \\
\text{pred}(\text{suc}(0))
\end{array}
\]
In general

A rewrite system which is strongly normalizing and which is Church-Rosser defines an initial algebra

(where equality is obtained from rewriting by symmetry)

*Justifies* OBJ2 -Semantics
Correctness of "Usage"
- A paradigmatic discussion

Separate between Specification and Implementation, between "what" and "how"

is a doctrine of the theory of ABSTRACT DATA TYPES

Matter of taste:

- Module comprises specification and implementation, or
- Modules are specifications, implementation is a relation between modules.

We will adhere to the second view. However, we assume that every specification is implemented eventually.

* A more sophisticated view is expressed in Ehrig & al, Algebraic Theory of Module Specification with Constraints, MFCS'86, LNCS 239, 1986
What is an Implementation?

Various views depending on the concept of meaning:

A. Model-based:
Representation of data and operations (and relations), e.g.
- an algebraic specification by a data structure and functional programs,
- a "data base" by a relational database system

B. Deduction-based:
by a deduction system plus axioms resp. derivation rules, e.g.
- algebraic specification by a rewrite system
  "suc(pred(x)) → x"
- a PROLOG-GT-module by a proof system (PROLOG ?)
- a "data base" by PROLOG
An Example

STACK implemented by ARRAY with pointer

Stack → record a : array[1..?] of integer
        p : integer
        end

push → push(<a,p>, d)
       = <update(a,p,d), suc(p)>

top → top(<a,p>) = <a,pred(p)>
"Usage"

Modules should be implemented independently!

Then the implementation of module B must be able to use the implementation of module A if the module A uses the module B!
Consequences

A. Model-based implementation

The language of B can only refer to data of A which can be referred to in the language of A: B is sufficiently complete w.r.t. A

\( \forall t \in T_B \exists t' \in T_A . t \approx t' \)

OTHERWISE

the implementation of A may not provide a value as one would only implement what is necessary to be implemented from the viewpoint of A.
Example

Not sufficient complete top(empty) not in STACK

Modify

\[ \text{top}(\text{empty}) = 0 \]

Now sufficient complete
?? PROOF ??

IDEA : Use Rewriting

(i) STACK is strongly normalizing and Church-Rosser !!!

(ii) Use this to prove that every term of sort Stack reduces to a term of the form \( \text{push(push(...push(\text{empty}, m_1)...), m_{n-1}), m_n) } \)

where the \( m_i \)'s are INTEGER -terms, and every term of sort Integer reduces to an INTEGER -term.
Exercise

spec INTEGER is
sorts integer
ops 0: $\rightarrow$ integer
   suc : integer $\rightarrow$ integer
   pred : integer $\rightarrow$ integer
var m : integer
   pred(suc(m)) = m
   suc(pred(m)) = m

uses

spec NAT is
sorts integer
ops 0: $\rightarrow$ integer
   suc : integer $\rightarrow$ integer

Q Is this usage correct?
Another Phenomenon

obj INTEGER_MOD_5 is
using INTEGER

eq : suc^5(M) = M

jbo

The integers need a reimplementation

Q Add suc^5(M) \rightarrow M. Good enough?

Necessary for independency

Every formula expressible in A which holds in B must already hold in A:
B is consistent over A, or a conservative extension of A

(" \vdash_B t = t' \Rightarrow \vdash_A t = t' for A-terms t,t' ")
Back to OBJ2

The different notions of "usage" are reflected as follows

\[
\text{obj B is using A} \quad \text{"No restriction"}
\]

\[
\text{obj B is extending A} \quad \text{"Consistency"}
\quad \text{"No confusion"}
\]

\[
\text{obj B is protecting A} \quad \text{"Persistency" =}
\quad \text{"Consistency" &}
\quad \text{"Suff.Completeness"}
\quad \text{"No confusion & no junk"}
\]
MORAL

A theory of "programming in the large" is concerned with

• Modules as basic entities
• Operations and Relations on modules

These operations and relations come along with correctness criteria reflecting kind and degree of independency of modules relative to other modules.
OPERATIONS ON MODULES

USING

Does B extend A? *No confusion*
Does B protect A? *No confusion & No Junk*

PROOF OBLIGATION
Union or Disjoint Union?

- Union: may cause confusion if same names used
- Disjoint Union: automatic renaming
Example \[ \text{BOOL + INTEGER} \]

\[
\text{th BOOL is}
\]
\[
\text{sort Bool}
\]
\[
op 0 : \rightarrow \text{Bool}
\]
\[
op 1 : \rightarrow \text{Bool}
\]
\[
op \_ + \_ : \text{Bool Bool} \rightarrow \text{Bool}
\]
\[
eq : 0 + 0 = 0
\]
\[
eq : 0 + 1 = 1
\]
\[
eq : 1 + 0 = 1
\]
\[
eq : 1 + 1 = 1
\]
\[\text{enth}\]

Union: Overloading of operators
(May be disambiguated syntactically, e.g. annotation by sorts)

\[
\text{th BOOL+ is}
\]
\[
\text{protecting BOOL}
\]
\[
op \_ : \text{Bool} \rightarrow \text{Bool}
\]
\[
eq : - 0 = 1
\]
\[
eq : - 1 = 0
\]
\[\text{enth}\]

Must be disambiguated

Possible disadvantage: Reference to module is lost

ok for flat implementations (e.g. rewrite system), otherwise?
Disjoint Union

Seperate the specifications, for instance by prefixing with the module name

ボール + インTEGER +

Advantage: Unambiguous reference to a module
Disadvantage: Naming conventions may become complicated

INTEGER + INTEGER

OBJ2 - Union
SHARED SUBMODULES

Posh name: PUSHOUT
Different modules may use different notations

\[ A + \text{BOOL}_B \]

- BOOL

A

- BOOL1

B

- BOOL2

Rename
More complicated

Important Point

These kind of ideas work independently of the actual notion of module
Views map specifications to specifications

view VIEW of DEQUE as STACK
sort Integer to Integer
sort Stack to Stack
var M : Integer
vars S : Stack
op : 0 to : 0
op : suc(M) to : suc(M)
...
op : push(S,M) to : push(S,M)
...
endview

Idea: View a "deque" as a "stack"

view INTEGER_AS BOOL of INTEGER as BOOL
sort Bool to Integer
var B, B' : Bool
op : false to : 0
op : true to : suc(O)
op : B or B' to : add(B,B')
endview
Semantically

\[ \text{SPEC1} \xrightarrow{\text{view}} \text{SPEC2} \]

reduce

\[ A_{\text{SPEC1}} \leftarrow A \]

More precisely

\[ A_{\text{SPEC1}} = A_{\text{view}}(s) \quad s \in \text{SPEC1} \]

\[ \text{view}(\sigma)_A \quad \sigma \in \text{SPEC1} \]

Proviso: The view preserves properties,

i.e. the translation of every equation in SPEC1 must be derivable in SPEC2

e.g. true or false = true \rightarrow add(1,0) = 1
PARAMETERISATION

Many modules can be defined relative to a parameter, e.g. stacks, arrays, queues, finite sets, etc.
object ARRAY /INDEX :: TRIV, DATA :: EQ/ is
protecting BOOL

sort Array

op new : $\rightarrow$ Array

op get : Array Elt. INDEX $\rightarrow$ Elt. DATA

op update : Array Elt. INDEX Elt. DATA $\rightarrow$ Array

op eq : Elt. DATA Elt. DATA $\rightarrow$ Bool

op if : Bool Elt. DATA Elt. DATA $\rightarrow$ Elt. DATA

var A, A' : Array, I, J : Elt. INDEX, D, D', D'' : Elt. DATA

eq : get(update(A,I,D),J) = if(eq(I,J),D,get(A,J))

eq : update(update(A,I,D),J,D') =
if(eq(I,J),update(A,I,D'), update(update(A,J,D'),I,D))

eq : if(true,D,D') = D

eq : if(false,D,D') = D'

eq : if(true,A,A') = A

eq : if(false,A,A') = A'

jbo


th TRIV is

sort Elt

endth

th EQ is

protecting BOOL

sort Elt

op eq : Elt Elt $\rightarrow$ Bool

var D, D', D'' : Elt

eq : eq(D,D) = true

eq : q(D,D') = eq(D',D)

eq : eq(D,D'') = eq(D,D') and eq(D',D'')

NOTE: BOOL is a shared submodule.
Updating

view INTEGER_AS_ELT of INTEGER as TRIV
sort Elt to Integer
endview

view INTEGER_AS_ELT&EQ of INTEGER as TRIV
sort Elt to Integer
var D, D' : Elt
op : eq(D,D') to : eq(D,D')
endview

ARRAY /INTEGER_AS_ELT, INTEGER_AS_ELT&EQ/

and so on
Thus

Can cause problems  !!

pto
Correctness Criteria (very superficially)

- PMOD(ACTUAL) uses ACTUAL, hence the correctness criteria of "usage" apply\(^1\).
- In a sense PMOD(ACTUAL) also uses PMOD.

An "implementation" can only be a construction which yields an implementation of PMOD(ACTUAL) provided that an implementation of ACTUAL is given, e.g. ARRAY, STACK.

There are, however, hiccups; For instance descriptions such as 'update(new,0,get(new,0))' in ARRAY(INTEGER) would not have an implementation.

Therefore new correctness criteria are needed\(^2\)

\(^1\) "Parameter protection" and

\(^2\) "Passing compatibility" in algebraic specifications.

"Initial" Semantics of Parameterization

Informally

\[ \text{PARAMETER} \subseteq \text{BODY} \]

\[ A \rightarrow "A + \text{BODY-Data}" \]
Special Cases:

Empty Parameter $\rightarrow$ Initial algebra

TRIV $\rightarrow$ Construct the initial algebra, but with the elements of the TRIV-model as additional constants

* Gives the general idea *

- Use the elements of the "actual parameter model" as additional constants for the initial algebra construction.

- May generate too many elements,

  eg. update EQ by INTEGER_MOD_5 (with suitable equality)
  then eq(5,0) is generated but not eq(5,0) = true

  hence identify as "necessary"
Implementation

STACK by ARRAY

Steps:
- Extend ARRAY by a "pointer"
- Represent every stack by an array and a pointer

Extension

\[ \text{obj ARRAY-POINTER1 is} \]
\[ \text{protecting ARRAY / NAT, /} \]
\[ \text{sort Array\times nat} \]
\[ \text{op } \langle _, _ \rangle : \text{Array Nat } \rightarrow \text{Array\times nat} \]

Realization

\text{ARRAY-POINTER1 real STACK by}
\[ \text{sort Array\times nat real Stack} \]
\[ \text{op } \langle \text{new,0} \rangle \text{ real empty} \]
\[ \langle a,n \rangle \text{ real } s \]
\[ \Rightarrow \langle \text{update}(a,\text{suc}(n),d),\text{suc}(n) \rangle \text{ real push}(s,d) \]
\[ \langle a,\text{suc}(n) \rangle \text{ real } s \Rightarrow \langle a,n \rangle \text{ real pop}(s) \]
\[ \langle a,n \rangle \text{ real } s \Rightarrow \text{get}(a,n) \text{ real top}(s) \]
Observations:

- No datas are identified, but multiple representation is used
  
  \(<...,0>\) represents the empty stack

- Implicitly defines operations on 'Array\times\text{nat}'
  
  \[\sigma'(x_1,...) \text{ real } \sigma(y_1,...) \text{ if } x_i \text{ real } y_i\]
  
  e.g. \(\text{push}' : \text{Array}\times\text{nat data} \rightarrow \text{Array}\times\text{nat}\)
  
  \(\text{push}'(<a,n>,d) = <\text{update}(a,\text{suc}(n),d),\text{suc}(n)>\)

- Realisation here does not preserve properties,
  
  e.g. \(\text{po}(\text{push}(s,d)) = s\), but
  
  \(\text{pop}'(\text{push}'(<a,n>,d)) = <\text{update}(a,\text{suc}(n),d),n> \neq <a,n>\)

Correctness Criteria

- If \(x \text{ real } y\) and \(x \text{ real } y'\) and \(y = y'\)

- \(\forall y \exists x . \ x \text{ real } y\)
Serious gap:
No realization for operators provided

Hence alternatively,

\[
\begin{align*}
\text{obj ARRAY-POINTER2 is} \\
\text{protecting ARRAY /NAT, /} \\
\text{sort Array×nat} \\
pop <_,_> : \text{Array Nat} \rightarrow \text{Array×nat} \\
\text{pop empty'} : \rightarrow \text{Array Nat} \\
\text{push'} : \text{Array Nat Elt} \rightarrow \text{Array Nat} \\
\ldots \\
eq : \text{empty'} = <\text{new},0> \\
\text{push'}(s,d) = <\text{update}(a,\text{suc}(n),d),\text{suc}(n)> \\
\ldots
\end{align*}
\]

Realisation

\[
\begin{align*}
\text{ARRAY-POINTER2 real STACK by} \\
\text{sorts Array Nat real Stack} \\
\text{ops empty'} real empty \\
\text{push'}: \text{Array Nat Elt} \rightarrow \text{Array Nat} \\
\text{real push : Stack Elt} \rightarrow \text{Stack} \\
\ldots
\end{align*}
\]
Danger The definitions of σ’ may generate new "data" which should not be used for implementation, e.g.

\[\text{obj ARRAY-POINTER3 is}\]
\[\text{protecting ARRAY/NAT, /}\]
\[\text{sort Array\times nat}\]
\[\text{op } \langle \_, \_ \rangle : \text{Array Nat } \rightarrow \text{Array\times nat}\]
\[\text{op } \text{empty'} : \rightarrow \text{Array Nat}\]
\[\text{push'} : \text{Array Nat Elt } \rightarrow \text{Array Nat}\]

\[\ldots\]

**NO EQUATIONS**

**ARRAY-POINTER3 real STACK by**

To avoid problems

\[\text{obj REALSTACK is}\]
\[\text{protecting } \& \text{ ARRAY-POINTER1}\]
\[\text{op } \text{empty'} : \rightarrow \text{Array\times nat}\]
\[\text{op } \text{push'} : \text{Array\times nat Elt } \rightarrow \text{Array\times nat}\]
\[\text{op } \text{pop'} : \text{Array\times nat } \rightarrow \text{Array\times nat}\]
\[\text{op } \text{top'} : \text{Array\times nat } \rightarrow \text{Elt}\]
\[\text{var } A : \text{Array, N :Nat, D : Elt}\]
\[\text{eq : empty'} = \langle \text{new,0}\rangle\]
\[\text{eq : push'}(\langle A,N\rangle,D) = \text{update}(A,\text{suc}(N),D)\]
\[\text{eq : pop'}(\langle A,\text{suc}(N)\rangle) = \langle A,N\rangle\]
\[\text{eq : top'}(\langle A,N\rangle) = \text{get}(A,N)\]
CORRECTNESS

"Every stack translates to data indexed by ψ"  
(OP-completeness)

"No identification of data"  
(RI-Correctness)

ALTERNATIVELY*

spec REALSTACK is  
ψ ARRAY-POINTER with  
sorts stack  
ops empty : → stack  
push : stack data → stack  
pop : stack → stack  
top : stack → data  
ψ code : arraynat → stack  
var a : array, n :nat, d : data  
eqns empty = code(<new,0>)  
push(code(<a,n>),d) = code(update(a,suc(n),d))  
pop(code(<a,suc(n)>)) = code(<a,n>)  
top(code(<a,n>)) = get(a,n)

* This is the approach of: H. Ehrig, H.-J. Kreowski, B. Mahr, P. Padawitz, Algebraic Implementation of Abstract Data Types, TCS 20, 1982
one may be unhappy that realisation does not preserve properties, e.g. \( \text{pop}(\text{push}(s,d)) = s \).

In order to achieve this one may require stronger correctness criteria such as consistency or conservativeness ("all equations hold for the 'primed' operators")

This may be too severe, thus one might use relativisation predicates:

\[
\text{pop}'(\text{push}'(<a,n>,d)) = s \quad \text{holds only for} \quad <a,n> \quad \text{which "realise a stack"}
\]
Epilogue

Some extensions of the algebraic language

Conditional equations

\[
\text{th POSET is } \\
\text{protecting BOOL .} \\
\text{sort Elt .} \\
op _< : \text{Elt Elt } \to \text{Bool .} \\
\text{vars } E, E', E'' : \text{Elt .} \\
eq : E < E = \text{false .} \\
ceq : E < E'' = \text{true if (E < E' and E' < E'' ).} \\
\text{endth}
\]

? \text{ceq : E < E'' = true if (E < E' and E' < E'' ) .}

Conditional Equations

\[
t_1 = t'_1, \ldots, \ t_n = t_n \ \Rightarrow \ t = t'
\]

E < E' = true, E' < E'' = true \Rightarrow E < E'' = true
Conditional equations for error handling

\[ \text{obj} \quad \text{STACK2} \quad \text{DATA} :: \text{TRIV/} \text{is} \]
\[ \text{protecting} \quad \text{BOOL} \]
\[ \ldots \]
\[ \text{op} : \text{isempty} : \text{Stack} \rightarrow \text{Bool} \]
\[ \ldots \]
\[ \text{eq} : \text{isempty} (\text{empty}) = \text{true} \]
\[ \text{eq} : \text{isempty} (\text{push}(\text{S}, \text{D})) = \text{false} \]
\[ \ldots \]
\[ \text{eq} : \text{isempty} (\text{S}) = \text{true} \Rightarrow \text{pop} (\text{push}(\text{S}, \text{D})) = \text{S} \]
\[ \text{eq} : \text{isempty} (\text{S}) = \text{true} \Rightarrow \text{top} (\text{push}(\text{S}, \text{D})) = \text{D} \]

Using Subsorts for the same purpose

\[ \text{obj} \quad \text{STACK3} \quad \text{DATA} :: \text{TRIV/} \text{is} \]
\[ \text{sort} \quad \text{Nestack} \quad \ll \quad \text{Stack} \]
\[ \text{op} \quad \text{empty} : \quad \rightarrow \quad \text{Stack} \]
\[ \text{op} \quad \text{push} : \text{Stack \ Elt.DAT} \rightarrow \text{Nestack} \]
\[ \text{op} \quad \text{pop} : \text{Nestack} \rightarrow \text{Stack} \]
\[ \text{op} \quad \text{top} : \text{Nestack} \rightarrow \text{Elt.DAT} \]
\[ \text{var} \quad \text{S} : \text{Nestack}, \text{D} : \text{Elt.DAT} \]
\[ \text{eq} : \text{pop} (\text{push}(\text{S}, \text{D})) = \text{S} \]
\[ \text{eq} : \text{top} (\text{push}(\text{S}, \text{D})) = \text{D} \]

? \quad \text{pop} (\text{pop} (\text{push} (\text{push} (\text{empty}, \text{D}), \text{D}')))) \quad ?
Subsorts and Overloading

\[\begin{align*}
\text{obj INT\_NAT is} \\
\text{sort Nat} &< \text{Integer} \\
\text{op 0:} &\rightarrow \text{Nat} \\
\text{op suc: Nat} &\rightarrow \text{Nat} \\
\text{op suc: Integer} &\rightarrow \text{Integer} \\
\text{op add: Nat Nat} &\rightarrow \text{Nat} \\
\text{op add: Integer Integer} &\rightarrow \text{Integer} \\
\text{var M,N: Integer} \\
\text{eq: add(M,0) = M} \\
\text{eq: add(M,suc(N)) = suc(add(M,N))}
\end{align*}\]

The operators "suc" and "add" behave on natural numbers as integers just as on integers
Another Problem

Bounded stacks

" \text{length}(S) < \text{bound} = \text{true} \Rightarrow \text{push}(S,D) : \text{Stack} "

There are various theories to cope with this situation, let us try a new one:

• "push" is a partial function \( \Rightarrow \) Consider partial algebras

• Introduce unary type predicates \( _e \in \text{Stack} \)

• Say that a term is "defined" if it has a type

Read

" \text{length}(S) < \text{bound} = \text{true} \Rightarrow \text{push}(S,D) \in \text{Stack} "

as

"if the length of a stack is less than a bound then \text{push}(S,D) is defined"

Better ensure that the arguments are defined

"S \in \text{Stack}, D \in \text{Data}, \text{length}(S) < \text{bound} = \text{true} \Rightarrow \text{push}(S,D) \in \text{Stack} "
Why not arbitrary relations? - Yes, why not?

"S \in \text{Stack}, D \in \text{Data}, \text{length}(S) < \text{bound} \Rightarrow \text{push}(S, D) \in \text{Stack}"

Why not more general types?

_ \in \{ x \mid \varphi(x) \}

"N \in \text{Nat}, S \in \text{Stack(suc(N))} \Rightarrow \text{pop}(S) \in \text{Stack}(N)"

where \quad \text{Stack}(N) = \{ X \mid \text{Stack}(X, N) \}

i.e. \textbf{Dependent Types}

Nothing to worry:
Mathematics is the same
spec CATEGORY is
sorts MOR
ops _ ; _ : MOR MOR → MOR
rel ob : MOR
        mor : OB OB MOR
axioms A ∈ ob :: A ∈ mor(A,A)
        A,B,C ∈ ob, f ∈ mor(A,B), g ∈ mor(B,C)
        ⊢ f ; g ∈ mor(A,C)

where mor(A,B) = \{ f : Mor \mid mor(A,B,f) \}
Resumé

A lot can be done with algebra

Let's do it