Algebraic Specification

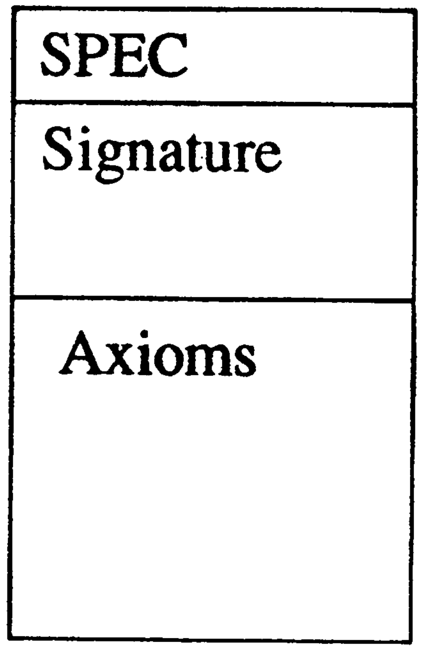
Axel Poigné

Topics

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

- Parameterization
- Realization and Implementation
- Extensions of Algebra

What is a Module ?



Signature - comprises the information of what syntactically given

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

Axioms - state what is "true"

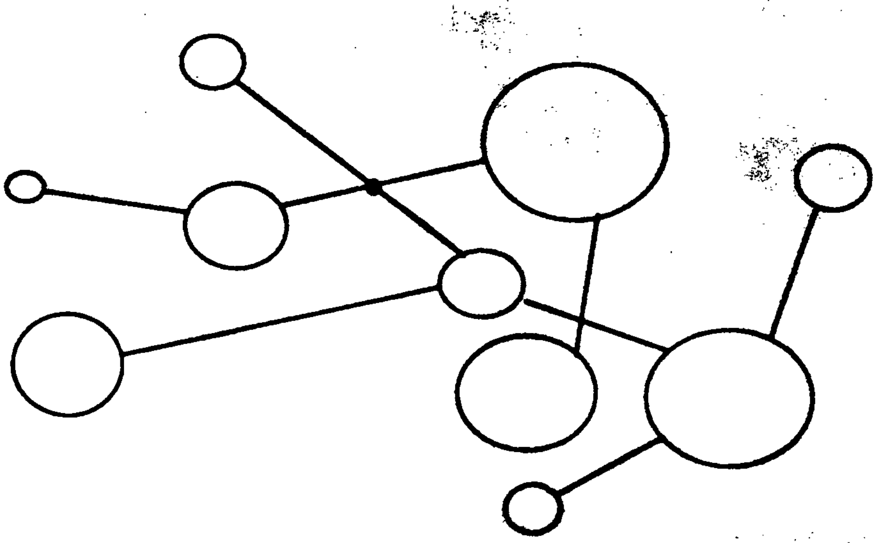
Pragmatics - can be implemented independently

Why Modules ?

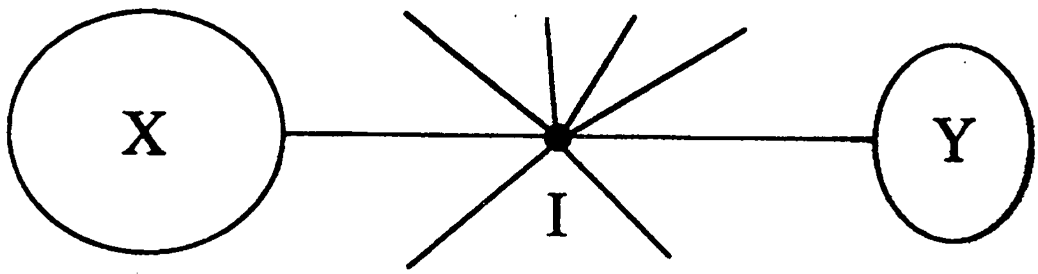
A Program System is

- 

- A set of small programs = **modules** connected by **interfaces**



A convential view of **interfaces**

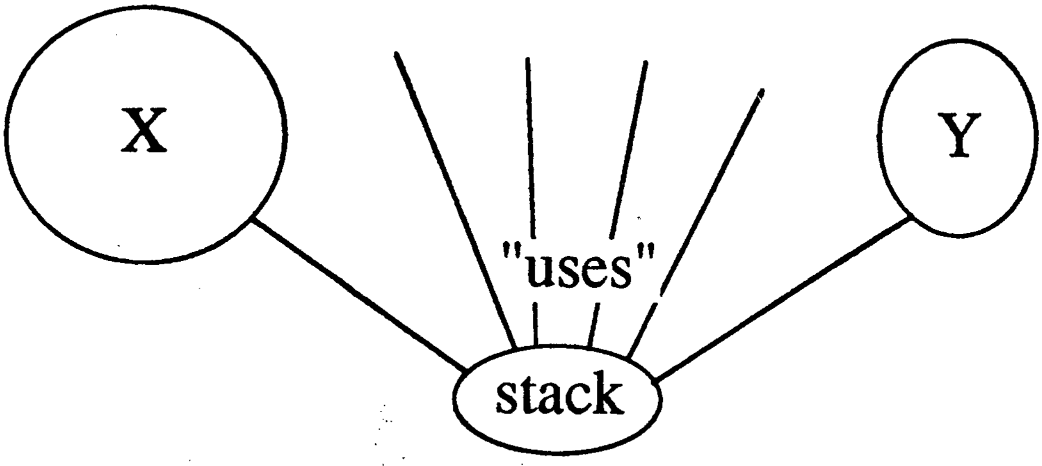



```
I = record  a : array [1..20] of integer;
            p : integer
    end
```

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 !



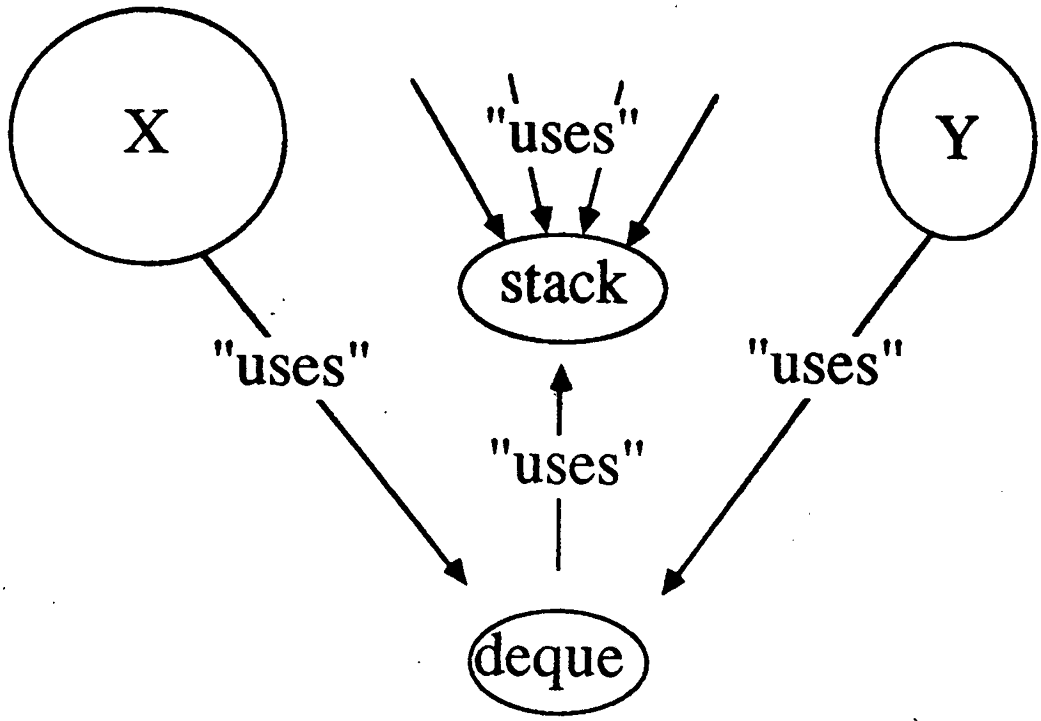
- homogeneous: Interfaces are modules **as any** other element in the system

- abstract

- **changes can be kept local**

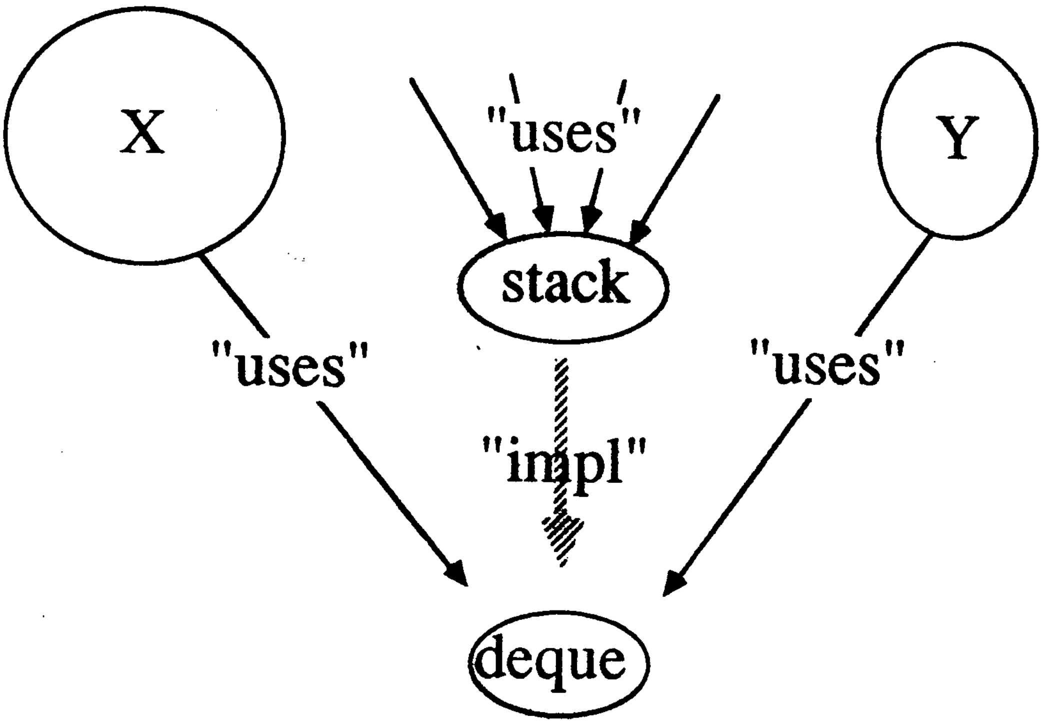
  (e.g.Overflow check can be done once and for all in stack)

Modularisation allows local change



The "interface module" STACK is retained for all modules except for X and Y.

Alternative View



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*

---

*J.Goguen - R.Burstall: Institutions: Abstract Model Theory for Computer Science, LNCS 164, 1984

OBJ2 - MODULES (theories)

```
th INTEGER is
sorts Integer                          |
op    0 : → Integer                    |
op    suc : Integer → Integer          | "Signature"
op    pred : Integer → Integer         |
var   M : Integer
eq:   pred(suc(M)) = M                 |
eq:   suc(pred(M)) = M                 | "Axioms"
endth
```

```
th STACK* is
using INTEGER
sort Stack
op   empty : → Stack
op   push : Stack Integer → Stack
op   pop : Stack → Stack
op   top : Stack → Integer
var  S : Stack, M : Integer
eq : pop(push(S,M)) = S
eq : top(push(S,M)) = M
endth
```

* In spite of a widespread misunderstanding; stacks are in general not used to demonstrate the advantages of algebraic specification but rather the 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 aximatization.

(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 : → stack
push : stack integer → 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 : → stack
push : stack data → stack

state machine

state s : stack
ops pop : 1 → data
axioms pre(pop) = {s ≠ empty}
post(pop) = {∀s' : stack, d' : data .
s_0 = push(s',d) ⇒
s = s' ∧ d = d'}

---

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

RELATIONAL DATABASE

signature **sorts** name, address, price, size

relators persons : name × address

houses : address × price × size

frames join(persons,houses)

project(persons,name)

Model

persons

name	address
Tom	Rye 66
Mary	Rye 66

houses

address	price	size
Rye 66	150000	big
Lane 7	90000	middle
Crescent 1	60000	little

Evaluation

Val(join(persons,houses)) =

name	address	address	price	size
Tom	Rye 66	Rye 66	150000	big
Mary	Rye 66	Rye 66	150000	big

Val(project(persons,name)) = Tom
Mary

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*

...

endth

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

obj **INTEGER** *is*
sorts **Integer**
op **0 : → Integer**
op **suc : Integer → Integer**
op **pred : Integer → Integer**
var **M : Integer**
eq: **pred(suc(M)) = M**
eq: **suc(pred(M)) = M**
jbo

Objects have initial algebra semantics

Objects are animated by

Rewriting

"Compute" equations in one direction only

pred(suc(M)) → M
suc(pred(M)) → M

pred(suc(pred(pred(suc(0)...)
→ pred(pred(suc(0)))
→ pred(0)

pred(suc(pred(pred(suc(0)...)
→ pred(suc(pred(0)))
→ pred(0)

Rewrite rules are used "inside terms"

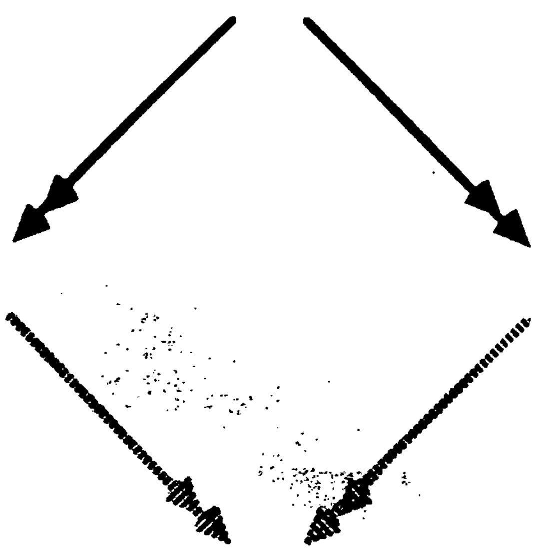
Problems

- 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'

Πραγματιχσ

Distinguish Constructors (which generate the data)

Define the other operators by case analysis

add(M,0) → M

add(M,suc(N)) → suc(add(M,N))

add(M,pred(N) → pred(add(M,N))

Q What are the constructors for **STACK** ?

Something wrong ?

A Nasty One

obj **SET** *is*
extending **INTEGER , BOOL**
sort **Set**
op $\varnothing$: $\rightarrow$ **Set**
op _ $\cup$ _ : **Set Set** $\rightarrow$ **Set**
op {_} : **Integer** $\rightarrow$ **Set**
op _ ε _ : **Integer Set** $\rightarrow$ **Bool**
var **S,S',S" : Set, M,N : Integer**
eq : $S \cup (S' \cup S'') = (S \cup S') \cup S''$
eq : $S \cup S' = S' \cup S$
eq : $S \cup S = S$
eq : $S \cup \varnothing = S$
eq : $M \, \varepsilon \, \{M\} \cup S$ = **true**
eq : $\{M\} \, \varepsilon \, \varnothing$ = **false**
endth

Q Why is this <u>object</u> unpleasant ?

Evaluation

$\{0\} \cup \{suc(0)\}$

$\rightarrow \{suc(0)\} \cup \{0\}$

$\rightarrow \{0\} \cup \{suc(0)\}$

$\rightarrow ...$ may not terminate

$\{0\} \cup \{0\}$

$\rightarrow \{0\}$ may terminate or not

$(\{0\} \cup \{suc(0)\}) \cup \{suc^2(0)\}$

$\rightarrow \{0\} \cup (\{suc(0)\} \cup \{suc^2(0)\})$

$\rightarrow (\{suc(0)\} \cup \{suc^2(0)\}) \cup \{0\}$

$\rightarrow \{suc(0)\} \cup (\{suc^2(0)\} \cup \{0\})$

$\rightarrow ...$

Termination is hard to check

$suc(0)\ \varepsilon\ \{pred(0)\} \cup (\{0\} \cup \{add(0,suc(0))\})$

$\rightarrow$ **$suc(0)\ \varepsilon\ \{pred(0)\} \cup (\{0\} \cup \{suc(add(0,0))\})$**

$\rightarrow$ **$suc(0)\ \varepsilon\ \{pred(0)\} \cup (\{0\} \cup \{suc(0)\})$**

$\rightarrow$ **$suc(0)\ \varepsilon\ \{pred(0)\} \cup (\{suc(0)\} \cup \{0\})$**

$\rightarrow$ **$suc(0)\ \varepsilon\ (\{suc(0)\} \cup \{0\}) \cup \{pred(0)\}$**

$\rightarrow$ **$suc(0)\ \varepsilon\ \{suc(0)\} \cup (\{0\} \cup \{pred(0)\})$**

$\rightarrow$ **true**

Mixture of "real" and "organisational computation

AC - Rewriting

$$t \underset{AC}{\rightarrow} t' \quad :\Leftrightarrow \quad t \underset{AC}{=} t'' \rightarrow t'$$

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

$$\begin{aligned}
& \text{suc}(0)\ \varepsilon\ \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{add}(0,\text{suc}(0))\}) \\
& \rightarrow \text{suc}(0)\ \varepsilon\ \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{suc}(\text{add}(0,0))\}) \\
& \rightarrow \text{suc}(0)\ \varepsilon\ \{\text{pred}(0)\} \cup (\{0\} \cup \{\text{suc}(0)\}) \\
& \underset{AC}{=} \text{suc}(0)\ \varepsilon\ \{\text{pred}(0)\} \cup (\{\text{suc}(0)\} \cup \{0\}) \\
& \underset{AC}{=} \text{suc}(0)\ \varepsilon\ (\{\text{suc}(0)\} \cup \{0\}) \cup \{\text{pred}(0)\} \\
& \underset{AC}{=} \text{suc}(0)\ \varepsilon\ \{\text{suc}(0)\} \cup (\{0\} \cup \{\text{pred}(0)\}) \\
& \rightarrow \text{true}
\end{aligned}$$

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

  card(∅) = 0

  card({M} ∪ S) = suc(card(S))

  Wright or Wrong ???

- Delete an element from a set

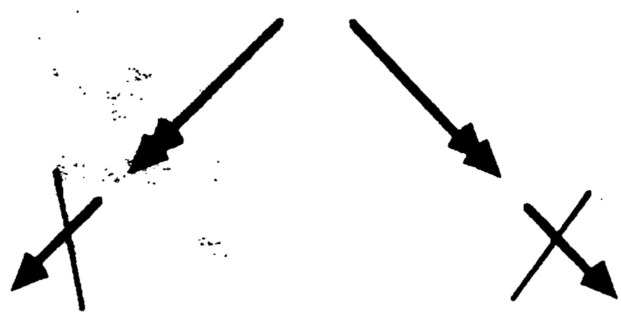
Theory

Normal Form

$$\cdots \twoheadrightarrow \not\rightarrow$$

Strong Normalization $\wedge$ Church Rosser

$\Rightarrow$ Unique Normal Forms



Contradiction

More Theory

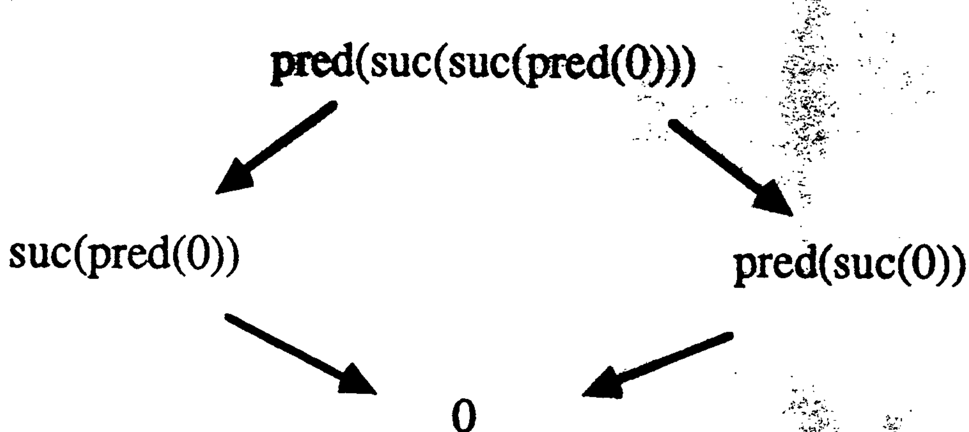
Observation

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

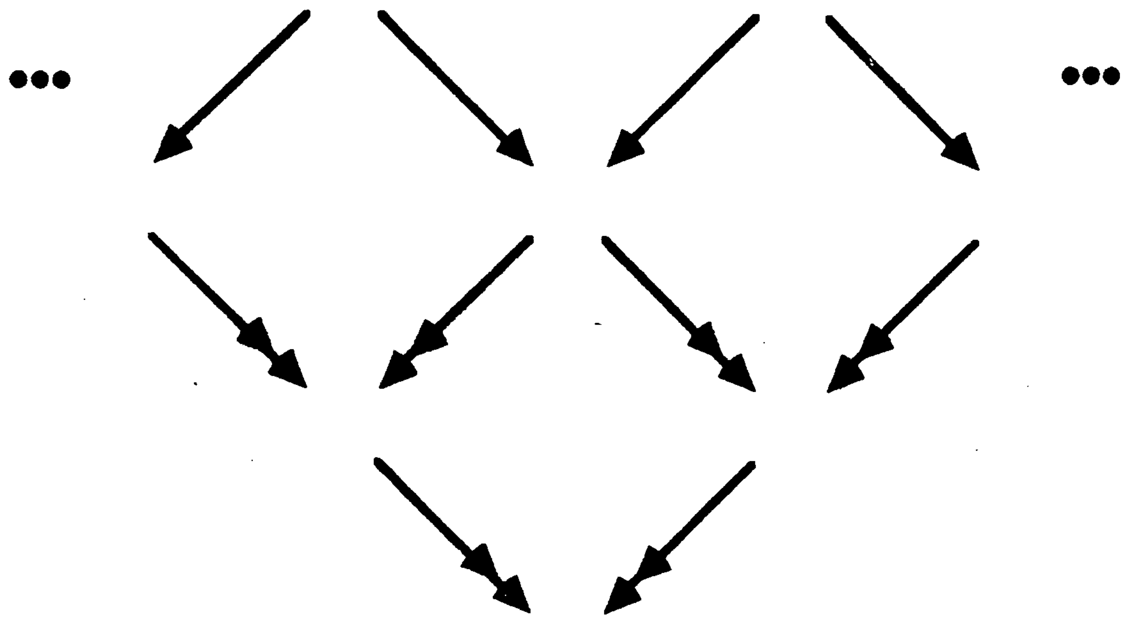
$$\Downarrow$$

suc(pred(0)) ← pred(suc(suc(pred(0)))) → pred(suc(0))

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



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

Seperate 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 <u>is</u> 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 and operations (and relations), e.g.

- an algebraic specification by a data structure and functional programs,
- a "data base" by a relational data base 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

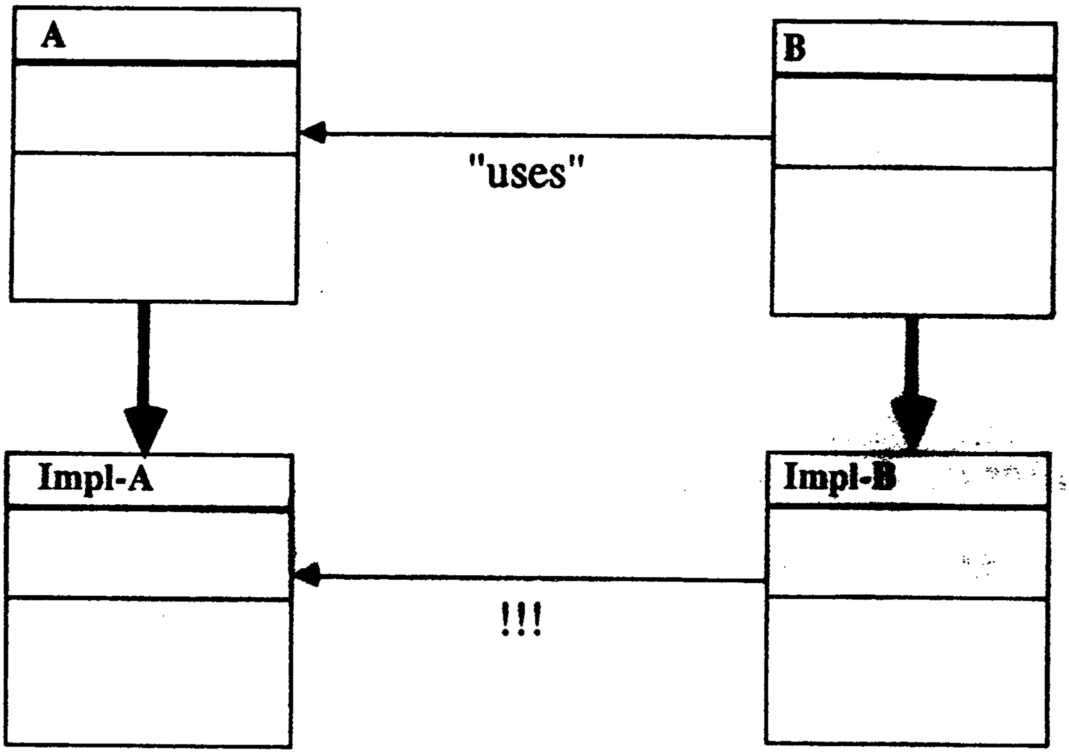
STACK implemented by **ARRAY** with **pointer**

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

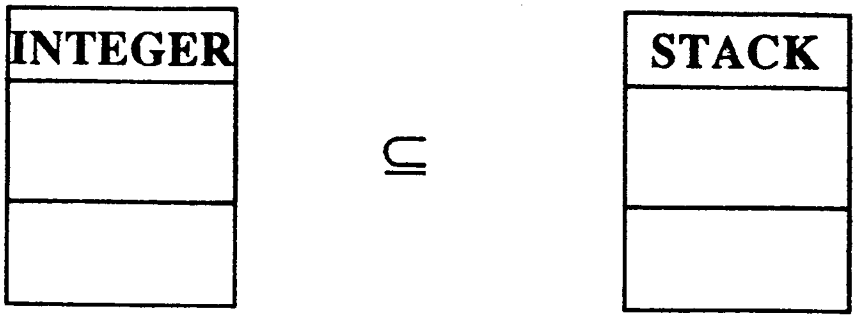
The language of B can only refer to data of A which can be refered to in the language of A:
B is **suffiently complete** w.r.t. A

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

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

top(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 push(push(...push(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 : → integer
suc : integer → integer
pred : integer → integer
var m : integer
pred(suc(m)) = m

suc(pred(m)) = m

uses

spec NAT **is**
sorts integer
ops 0 : → integer
suc : integer → 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

obj **B** *is*
using **A** — **"No restriction"**

. . .

obj **B** *is*
extending **A** — **"Consistency"**
"No confusion"

. . .

obj **B** *is*
protecting **A** — **"Persistency" = "Consistency" & "Suff.Completeness"**
"No confusion & no junk"

. . .

MORAL

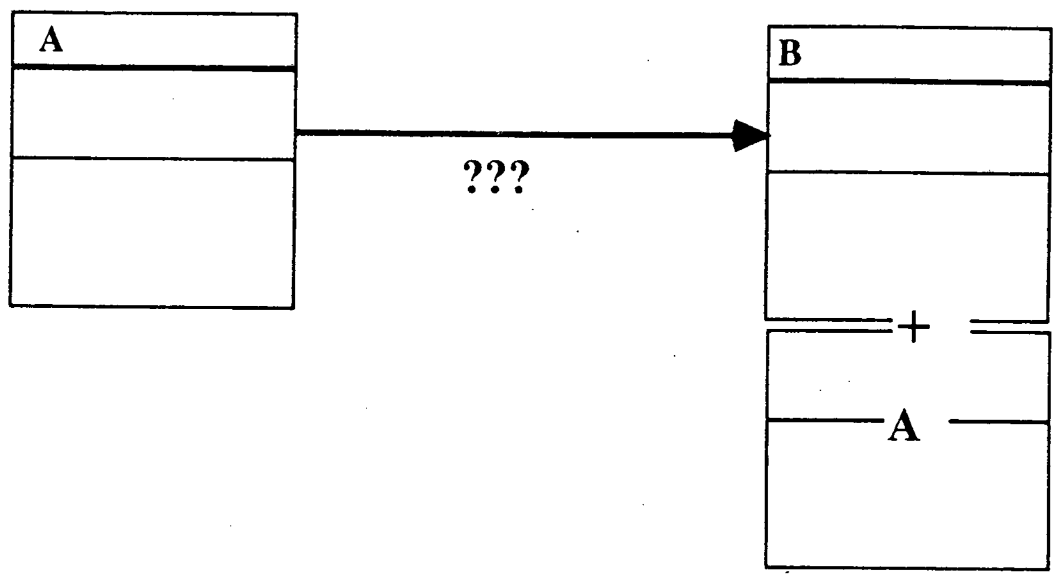
A theory of "programming in the large" is concerned with

- 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



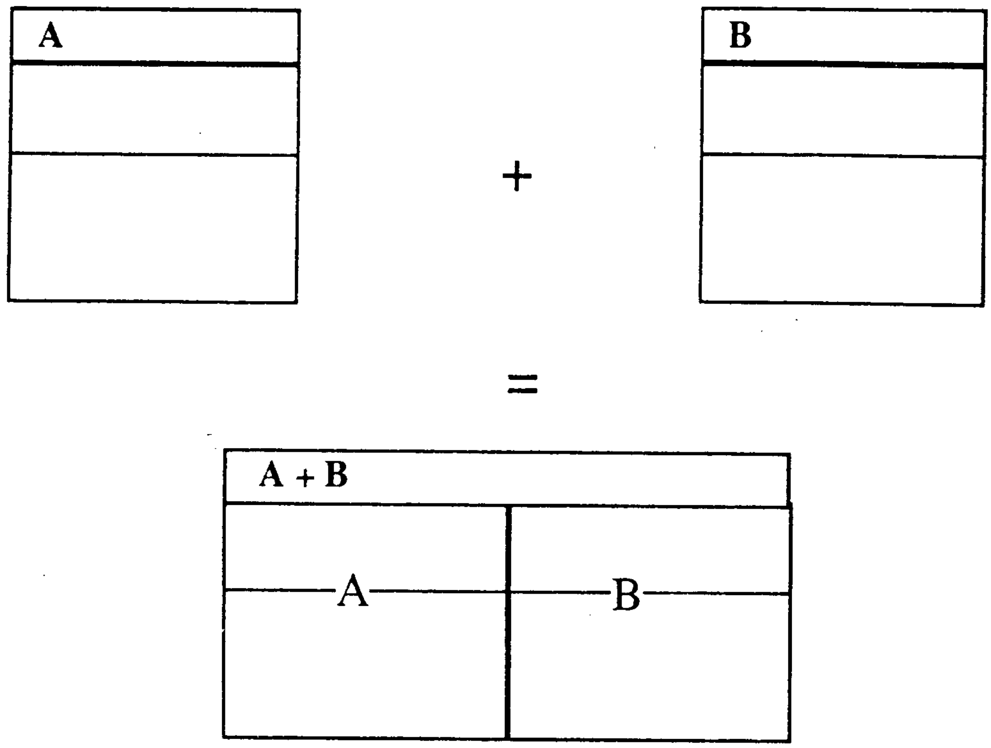
Does B extend A ? *No confusion*

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

$\Uparrow$

PROOF OBLIGATION

SUM



Union or Disjoint Union ?

- Union: may cause confusion if same names used
- Disjoint Union : automatic renaming

Example BOOL + INTEGER

th **BOOL** *is*
sort **Bool**
op **0 :** → **Bool**
op **1 :** → **Bool**
op **_ + _ : Bool Bool** → **Bool**
eq : **0 + 0 = 0**
eq : **0 + 1 = 1**
eq : **1 + 0 = 1**
eq : **1 + 1 = 1**
endth

Union : Overloading of operators

(May be disambiguated syntactically, e.g. annotation by sorts)

th **BOOL+** *is*
protecting **BOOL**
op **- : Bool → Bool**
eq : **- 0 = 1**
eq : **- 1 = 0**
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

_ BOOL.+_   **_ INTEGER.+ _**

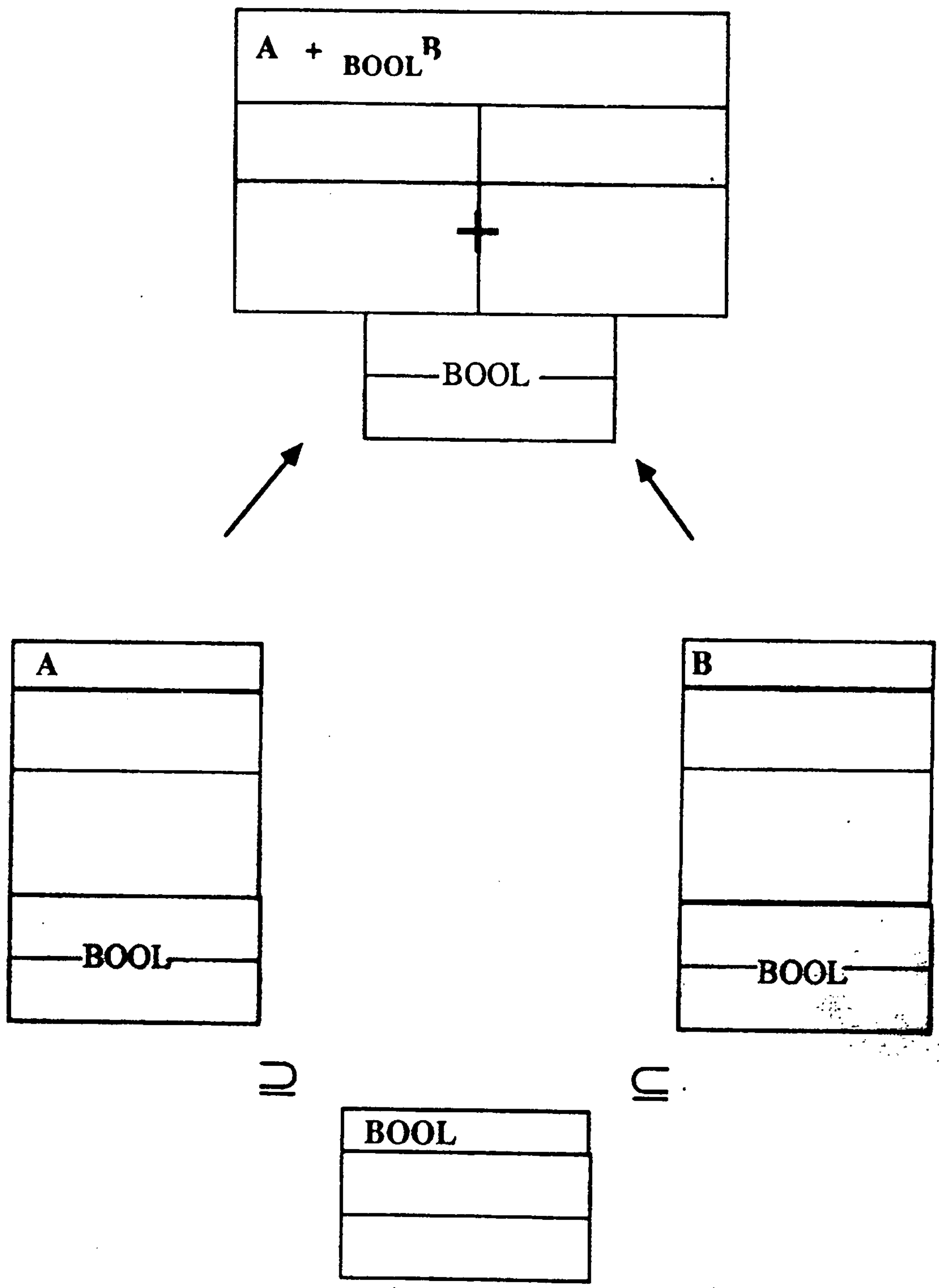
Advantage : Unambiguous reference to a module

Disadvantage: Naming conventions may become complicated

INTEGER + INTEGER

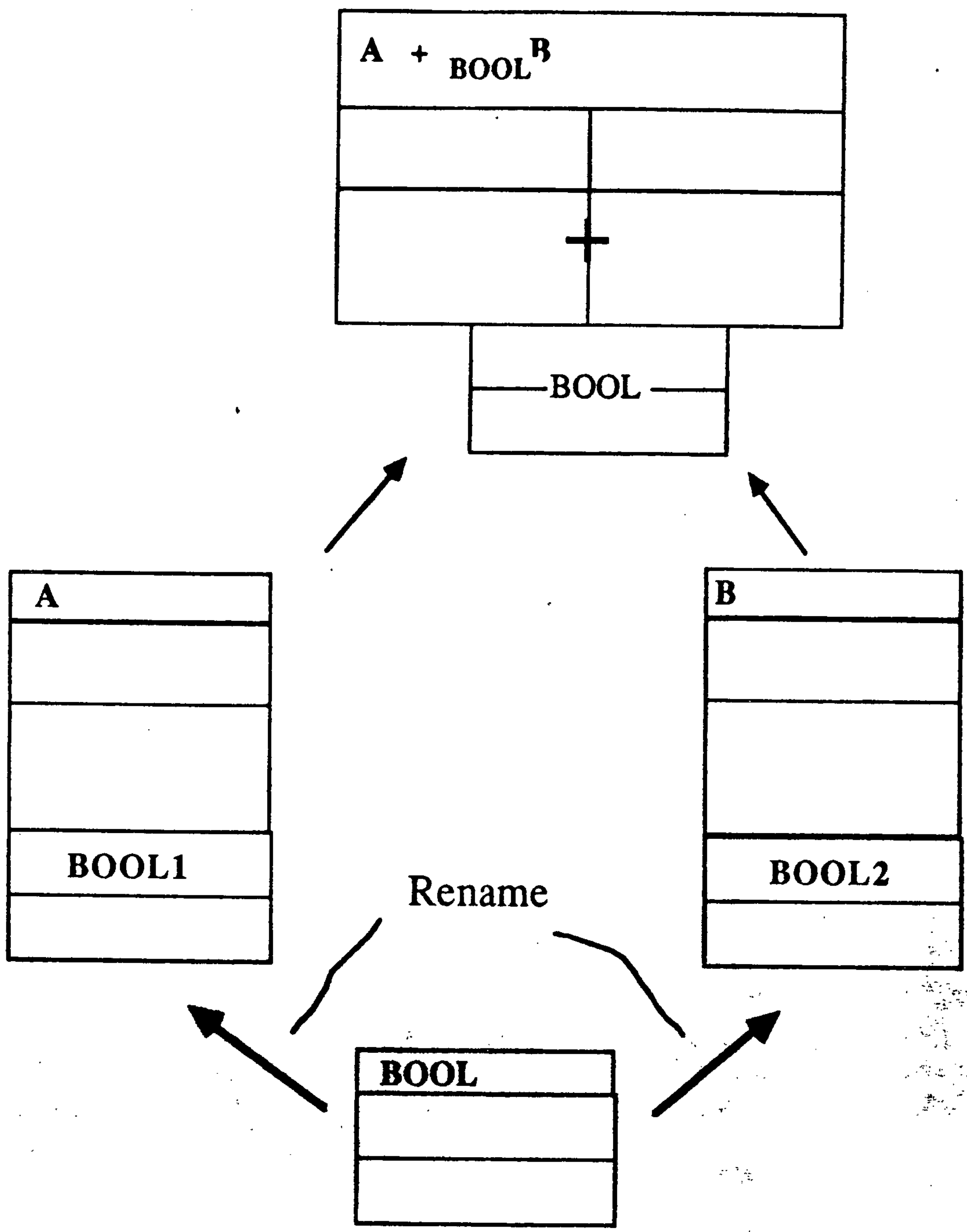
OBJ2 - Union

SHARED SUBMODULES

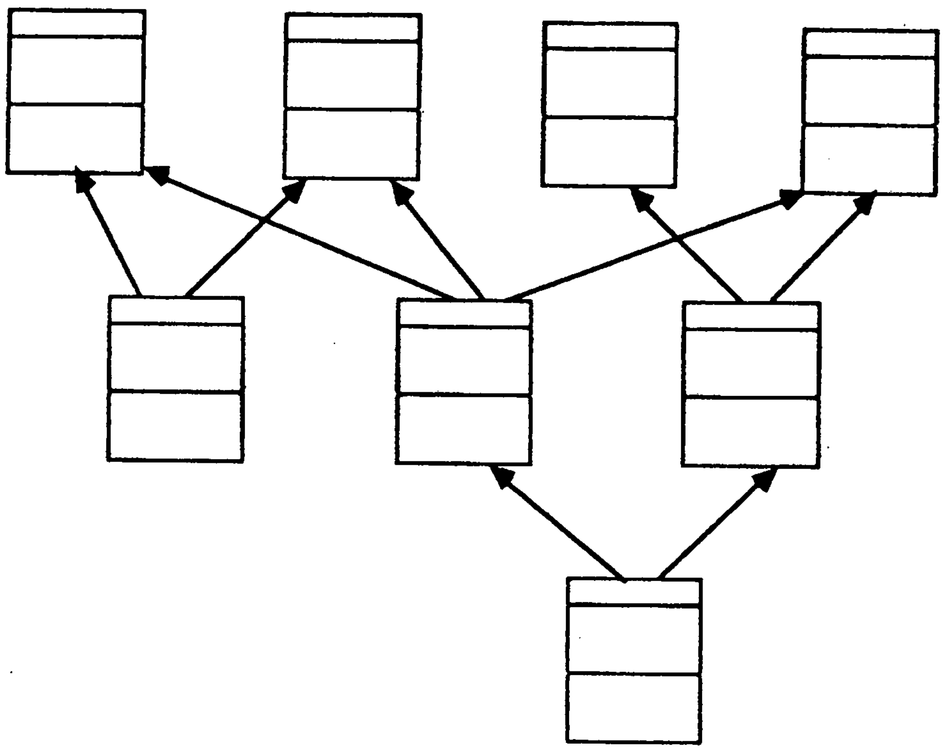


Posh name: PUSHOUT

Different modules may use different notations



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

$$\mathrm{SPEC1} \xrightarrow{\text{view}} \mathrm{SPEC2}$$

$$\mathcal{A}_{\mathrm{SPEC1}} \xleftarrow{\text{reduce}} \mathcal{A}$$

More precisely

$$\mathcal{A}_{\mathrm{SPEC1}} = \begin{array}{ll} \mathcal{A}_{\mathrm{view}(s)} & s \in \mathrm{SPEC1} \\ \mathrm{view}(\sigma)_{\mathcal{A}} & \sigma \in \mathrm{SPEC1} \end{array}$$

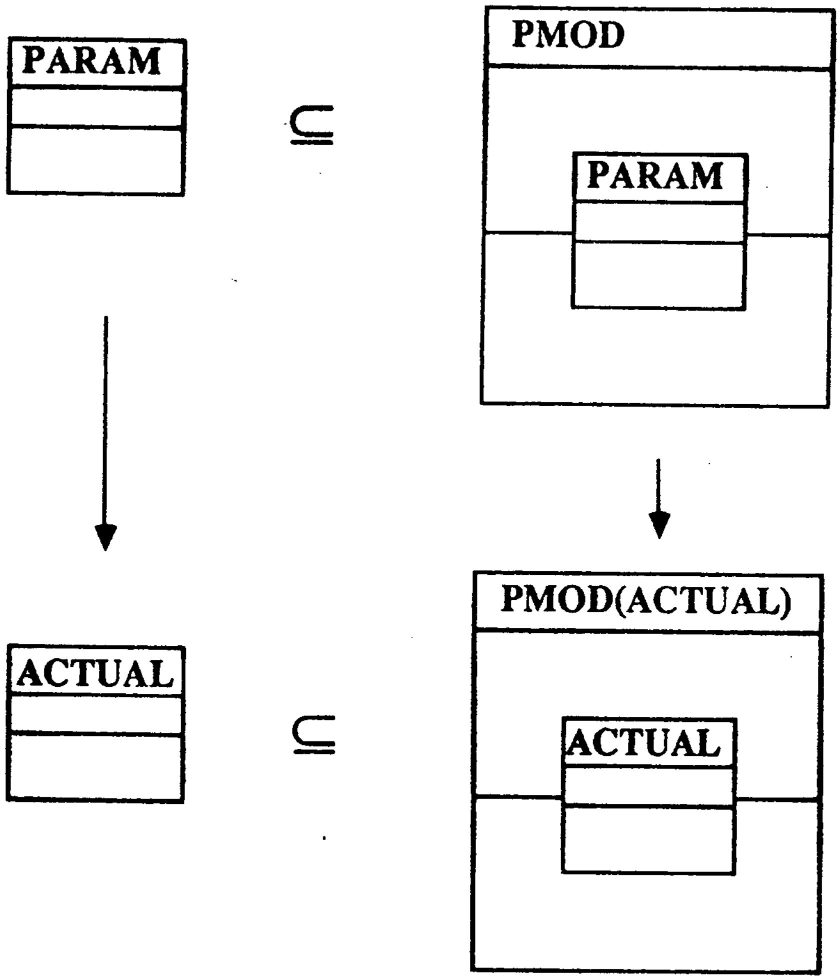
Proviso : The view preserves properties,

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

e.g. true or false = true → add(1,0) = 1

PARAMETERISATION

Many modules can be defined relative to a parameter, e.g. stacks, arrays, qeues, finite sets, etc.



```
obj  ARRAY  /INDEX :: TRIV, DATA :: EQ/ is
protecting  BOOL
sort    Array
op      new :  → Array
op      get : Array Elt.INDEX  →  Elt.DATA
op      update : Array Elt.INDEX Elt.DATA  →  Array
op      eq : Elt.DATA Elt.DATA → Bool
op      if : Bool Elt.DATA Elt.DATA  →  Elt.DATA
op      if : Bool Array Array  →  Array
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  →  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")
ndth
```

NOTE: BOOL
is a shared submodule.

Updating

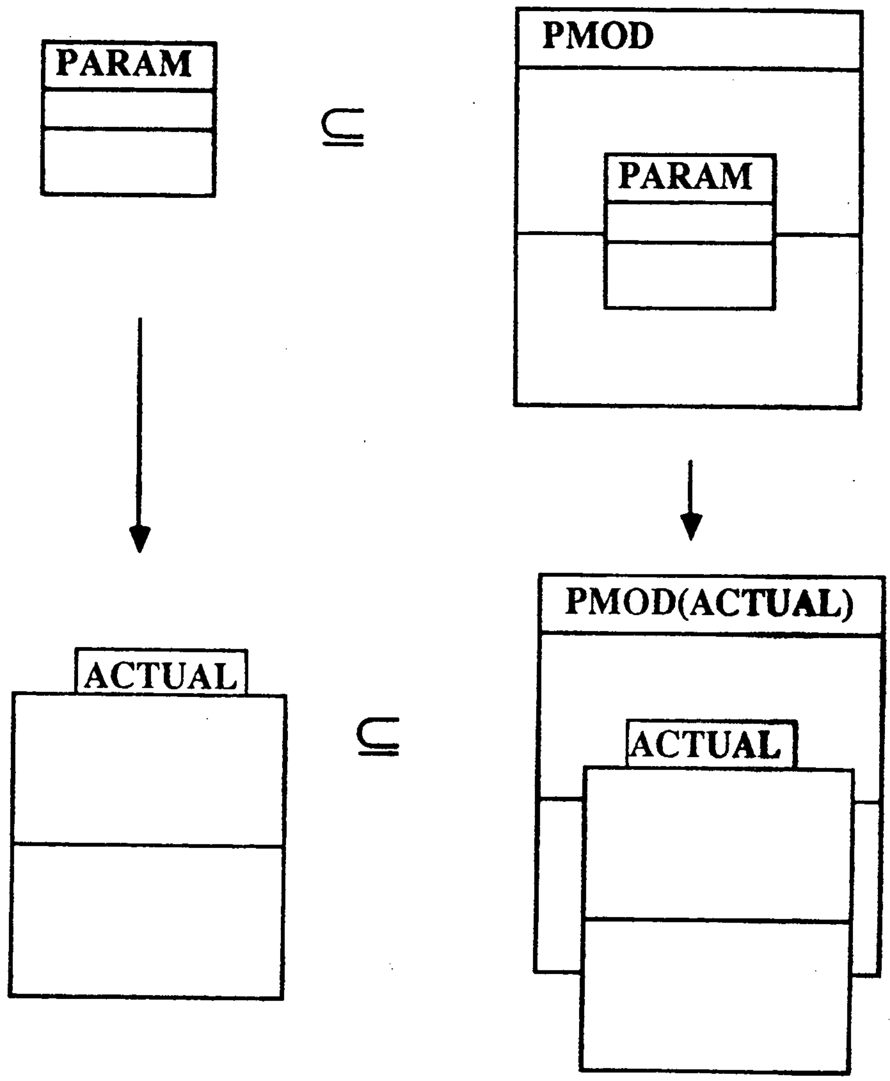
view **INTEGER_AS_ELT** *of* **INTEGER** *as* **TRIV**
sort **Elt** *to* **Integer**
endview

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.

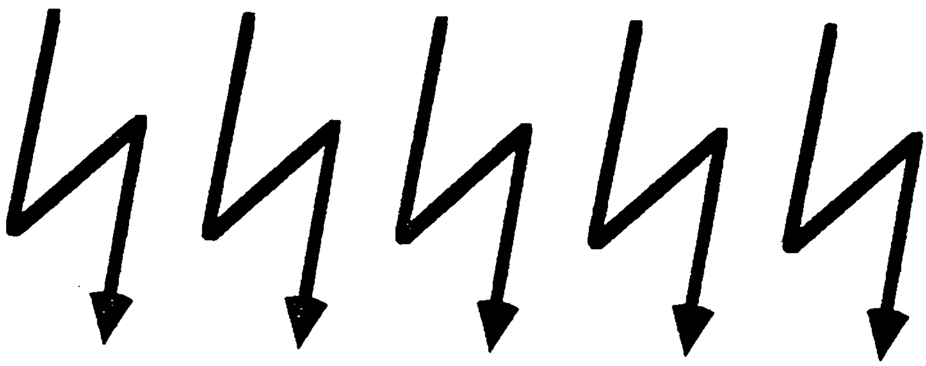
Therefore new correctness criteria are needed[2]

---

1 "Parameter protection" and

2 "Passing compatibility" in algebraic specifications.
H.Ehrig,H.-J.Kreowski,J.W.Thatcher,E.G.Wagner,J.B.Wright, *Parameter Passing in Algebraic Specification Languages*, TCS 33, 1984

"Initial" Semantics of Parameterization



Informally

PARAMETER $\subseteq$ BODY

$A \rightarrow$ "A + BODY-**Data**"

Special Cases:

Empty Parameter → Initial algebra

TRIV → Construct the initial algebra, but with the elements of the TRIV-model as additional constants

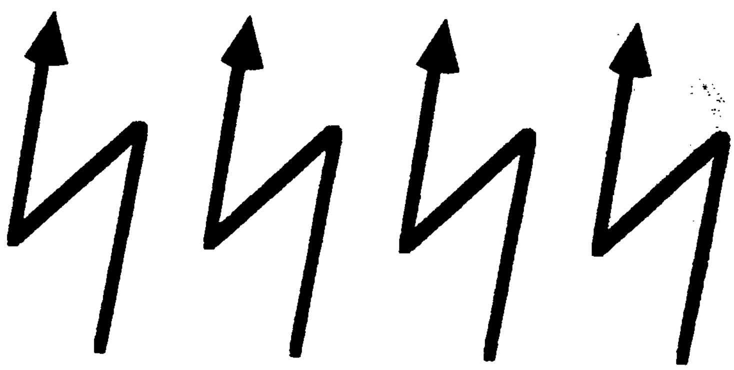
Gives the general idea

- 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

obj **ARRAY-POINTER1** *is*
protecting **ARRAY /NAT, /**
sort **Array×nat**
op < _ , _ > : **Array Nat** → **Array×nat**

Realization

ARRAY-POINTER1 *real* **STACK** *by*
sort **Array×nat** *real* **Stack**
op **<new,0>** *real* **empty**
<a,n> *real* **s**
⇒ **<update(a,suc(n),d),suc(n)>** *real* **push(s,d)**
<a,suc(n)> *real* **s** ⇒ **<a,n>** *real* **pop(s)**
<a,n> *real* **s** ⇒ **get(a,n)** *real* **top(s)**

Observations:

- No datas are identified, but multiple representation is used

  < ...,0> represents the empty stack

- Implicitly defines operations on '**Array×nat**'

  s.t. $\sigma'(x_1,\ldots)$ *real* $\sigma(y_1,\ldots)$ if x_i *real* y_i

  e.g. push' : **Array×nat** <u>data</u> → **Array×nat**

  push'(<a,n>,d) = <update(a,suc(n),d),suc(n)>

- Realisation here does not preserve properties,

  e.g. po(push(s,d)) = s, but

  pop'(push'(<a,n>,d)) = <update(a,suc(n),d),n> ≠ <a,n>

Correctness Criteria

- If x*real* y and x*real* y' and y = y'

- $\forall$ y $\exists$ x . x *real* y

Serious gap :

No realization for operators provided

Hence **alternatively,**

```
obj ARRAY-POINTER2 is
protecting  ARRAY /NAT, /
sort  Array×nat
op    < _ , _ > : Array Nat  →  Array×nat
op    empty' :  → Array Nat
        push' : Array Nat Elt  →  Array Nat
. . .
eq :   empty' = <new,0>
       push'(s,d) = <update(a,suc(n),d),suc(n)>
        . . .
```

Realisation

```
ARRAY-POINTER2 real STACK by
sorts  Array Nat  real  Stack
ops       empty'  real  empty
          push': Array Nat Elt  →  Array Nat
              real  push : Stack Elt  →  Stack
                   . . .
```

Danger The definitions of σ' may generate new "data" which should not be used for implementation, e.g.

```
obj ARRAY-POINTER3 is
protecting  ARRAY /NAT, /
sort  Array×nat
op    < _ , _ > : Array Nat  →  Array×nat
op    empty' :  → Array Nat
        push' : Array Nat Elt  →  Array Nat
. . .
```

NO EQUATIONS

ARRAY-POINTER3 *real* STACK *by*

To avoid problems

```
obj REALSTACK is
 protecting ¢ ARRAY-POINTER1
 op       empty' :  →  Array×nat
 op       push' : Array×nat Elt  →  Array×nat
 op       pop' : Array×nat  →  Array×nat
 op       top' : Array×nat  →  Elt
 var      A : Array, N :Nat, D : Elt
 eq :     empty'  =  <new,0>
 eq :     push'(<A,N>,D)  =  update(A,suc(N),D)
 eq :     pop'(<A,suc(N)>)  =  <A,N>
 eq :     top'(<A,N>)  =  get(A,N)
```

CORRECTNESS

"Every stack translates to data indexed by ¢"

(OP-completeness)

"No identification of data"

(RI-Correctness)

ALTERNATIVELY*

spec REALSTACK **is**
¢ ARRAY-POINTER **with**
sorts stack
ops empty : → stack
push : stack data → stack
pop : stack → stack
top : stack → data
¢ code : array×nat → 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

AT LAST (?)

one may be unhappy that realisation does not preserve properties, e.g. pop(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:

pop'(push'(<a,n>,d)) = s   holds only for <a,n> which "realise a stack"

Epilogue

Some extensions of the algebraic language

Conditional equations

th POSET *is*
protecting BOOL .
sort **Elt** .
op **_ < _ : Elt Elt → Bool** .
vars **E E' E"** : Elt .
eq : **E < E = false** .
ceq : **E < E" = true** if **(E < E' and E' < E")**.
endth

? *ceq* : **E < E" = true** if **(E < E' and E' < E")** ?.

Conditional Equations

$$t_1 = t'_1, \ldots, t_n = t_n \;\Rightarrow\; t = t'$$

$$E < E' = \text{true}, E' < E'' = \text{true} \;\Rightarrow\; E < E'' = \text{true}$$

Conditional equations for **error handling**

obj **STACK2** /DATA :: **TRIV**/*is*
protecting **BOOL**
...
op : **isempty : Stack $\rightarrow$ Bool**
...
eq : **isempty(empty) = true**
eq : **isempty(push(S,D)) = false**
...
eq : **isempty(S) = true $\Rightarrow$ pop(push(S,D)) = S**
eq : **isempty(S) = true $\Rightarrow$ top(push(S,D)) = D**

Using **Subsorts for the same purpose**

obj **STACK3** /DATA :: **TRIV**/ *is*

sort **Nestack** $<$ **Stack**

op **empty : $\rightarrow$ Stack**
op **push : Stack Elt.**DATA $\rightarrow$ **Nestack**
op **pop : Nestack $\rightarrow$ Stack**
op **top : Nestack** $\rightarrow$ **Elt.**DATA
var **S** : **Nestack, D** : **Elt.**DATA
eq : **pop(push(S,D)) = S**
eq : **top(push(S,D)) = D**

? pop(pop(push(push(empty,D),D')))) ?

Subsorts and Overloading

```
obj INT_NAT is
sort Nat < Integer
op  0 : → Nat
op  suc : Nat → Nat
op  suc : Integer → Integer
op  add : Nat Nat → Nat
op  add : Integer Integer → Integer
var M,N : Integer
eq : add(M,0) = M
eq : add(M,suc(N)) = suc(add(M,N))
```

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

Another Problem

Bounded stacks

" **length(S) < bound = true ⇒ push(S,D) : Stack** "

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

- "push" is a partial function ⇒ Consider partial algebras
- Intoduce unary type predicates _ **ε Stack**
- Say that a term is "defined" if it has a type

Read

" **length(S) < bound = true ⇒ push(S,D) ε Stack** "

as

"if the length of a stack is less than a bound then **push(S,D)** is defined"

Better ensure that the arguments are defined

"**S ε Stack, D ε Data, length(S) < bound = true**
⇒ push(S,D) ε Stack "

Why not arbitrary relations ? - Yes, why not ?

"**S ε Stack, D ε Data, length(S) < bound**
⇒ push(S,D) ε Stack "

Why not more general types ?

$\_ \; \varepsilon \; \{x \mid \varphi(x)\}$

"**N ε Nat,** S ε Stack(suc(N)) ⇒ pop(S) ε Stack(N)

where Stack(N) = {X | Stack(X,N)}

i.e. 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 ε ob  ∴  A ε mor(A,A)
          A,B,C ε ob, f ε mor(A,B), g ε mor(B,C)
                        ⊢  f ; g ε mor(A,C)
          . . .

         where  mor(A,B) = {f : Mor | mor(A,B,f)}
```

Resumé

A lot can be done with algebra

Let's do it