Algebraic Specifications

Bernhard Westfechtel

Lehrstuhl für Informatik III, RWTH Aachen

Characterization

- Formal specification of abstract data types
- Behavioral specification with the help of equations over terms
- Semantics defined by algebras (sorts + operations)
- Under certain restrictions algebraic specifications are operational
- Specifications may be refined in an evolutionary way
- Proof techniques: term rewriting, induction

Foundations

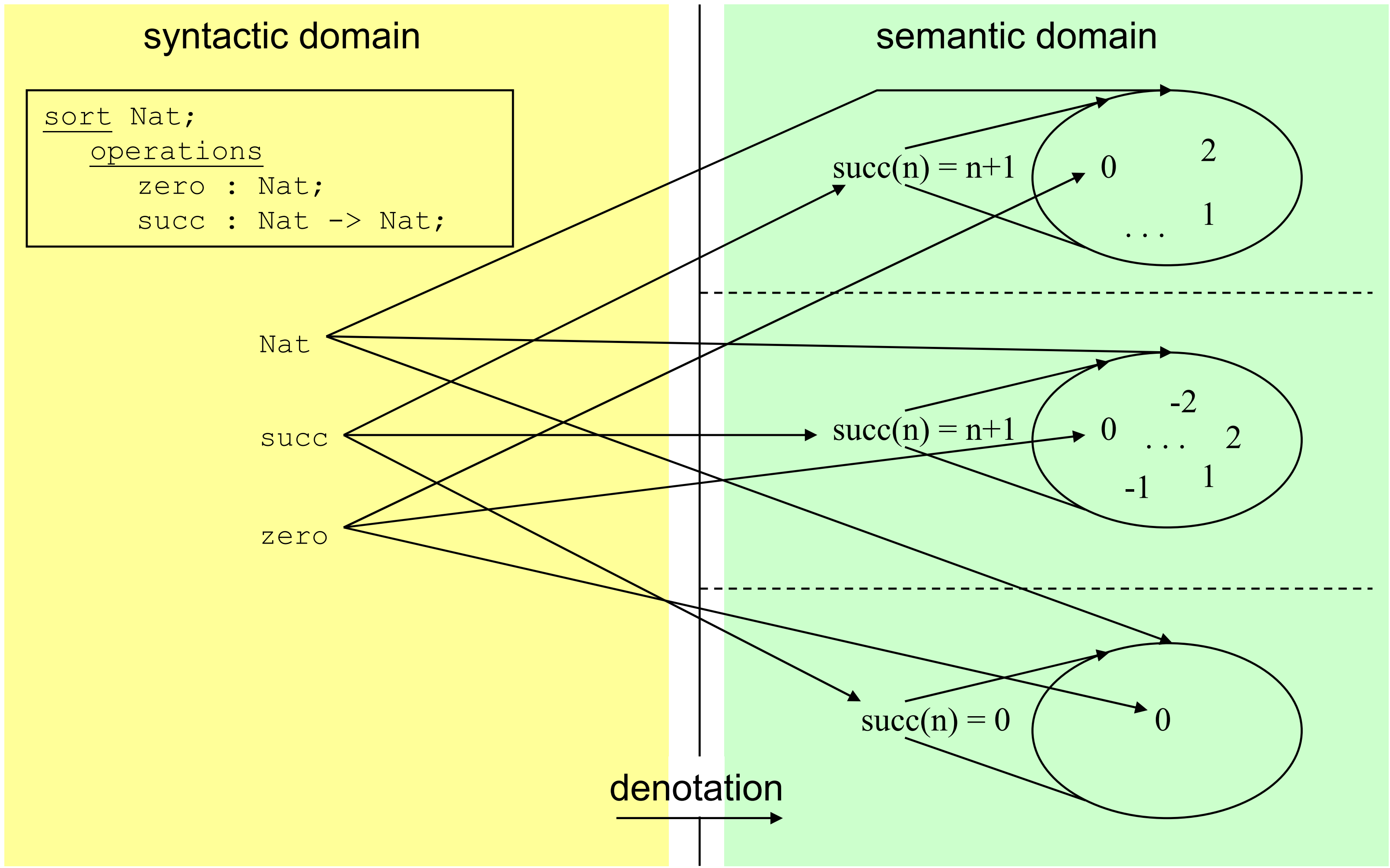
Signature and algebra

- **Signature** (syntactic domain)
  $\Sigma = <SN, FN, domN, ranN>$
  » $SN = \{sn_1, ..., sn_k\}$ set of sort names
  » $FN = \{fn_1, ..., fn_m\}$ set of function names
  » $domN: FN \rightarrow SN^*$ domain
  » $ranN: FN \rightarrow SN$ range
- **Algebra** (semantic domain)
  $A = <S, F, dom, ran>$
  » $S = \{S_1, ..., S_k\}$ set of sorts
  » $F = \{f_1, ..., f_m\}$ set of functions
  » $dom: F \rightarrow S^*$ domain
  » $ran: F \rightarrow S$ range

Denotation

- **Denotation** (mapping syntactic $\rightarrow$ semantic domain)
  $\delta : \Sigma \rightarrow A$
  - δ: $sn_i \rightarrow S_j$ (δ maps each sort name into a sort)
  - δ: $fn_i \rightarrow F_j$ (analogously for function names)
  - $dom(\delta(fn_i)) = \delta(domN(fn_i))$, $ran(\delta(fn_i)) = \delta(ranN(fn_i))$
    (domain and range "are preserved")

Example


syntactic domain
semantic domain
sort Nat;
operations
zero : Nat;
succ : Nat -> Nat;
Nat
succ
zero
succ(n) = n+1
0
2
1
. . .
succ(n) = n+1
0
-2
. . .
2
-1
1
succ(n) = 0
0
denotation

Terms

- **Term language**

  Let Σ = <*SN, FN, domN, ranN*> be a signature and *X* be a set of typed variables (i.e. each variable *x* is mapped into a sort name $sn \in SN$). Terms are defined inductively as follows:

  - Each variable *x* is a term of its respective type *sn*.
  - Each function name *fn*, where $fn: \rightarrow sn$ (i.e. *fn* denotes a nullary function/constant) is a term of type *sn*.
  - Given $fn: sn_1 \times \ldots \times sn_k \rightarrow sn$ ($k \geq 1$), $t_1, \ldots, t_k$ terms of type $sn_1, \ldots, sn_k$. $fn(t_1, \ldots, t_k)$ is a term of type *sn*.
  - Each element of the term language may be generated by a finite derivation applying the rules given above.

- **Variable-free term language**

  - Terms which do not contain variables

Example: stack

signature

```
sorts Stack, Nat, Bool;
   operations
      true, false : -> Bool;
      zero : -> Nat;
      succ : Nat -> Nat;
      newstack : -> Stack;
      push : Stack x Nat -> Stack;
      isnewstack : Stack -> Bool;
      pop : Stack -> Stack;
      top : Stack -> Nat;
```

terms

```
zero
succ(zero)
succ(succ(zero))
newstack
push(newstack,zero)
isnewstack(newstack)
pop(newstack)
top(newstack)
pop(push(newstack,zero))
top(push(newstack,zero))
push(x,y)
push(x,succ(succ(y)))
...
```

Word algebra

- **Word algebra**
  - » Variable-free terms, interpreted as strings

terms	strings
`zero`	"zero"
`succ(zero)`	"succ(zero)"
`succ(succ(zero))`	"succ(succ(zero))"
`newstack`	"newstack"
`push(newstack,zero)`	"push(newstack,zero)"
`isnewstack(newstack)`	"isnewstack(newstack)"
`pop(newstack)`	"pop(newstack)"
`top(newstack)`	"top(newstack)"
`pop(push(newstack,zero))`	"pop(push(newstack,zero))"
`top(push(newstack,zero))`	"top(push(newstack,zero))"
`...`	...

Example: δ(`push`)("newstack", "zero") = "push(newstack, zero)"

Substitution

- **Substitution**

  Let $T(\Sigma)$ be a set of terms over a signature Σ, X be a set of typed variables. A substitution σ is defined as follows:

  » $\sigma : X \rightarrow T(\Sigma)$, where x and $\sigma(x)$ must have the same type

- **Ground substitution**

  Substitution of variables by variable-free terms

```
push(x,y)
    |
    |   y -> succ(zero)
    v
push(x,succ(zero))
    |
    |   x -> push(newstack,zero)
    v
push(push(newstack,zero),succ(zero))
```

Properties of abstract data types

- **Presentation**
  - » A presentation (Σ, E) is a signature Σ, combined with a set of equations E
  - » Each equation $e \in E$ is built up as follows:
    $t_1 == t_2$ (t_1, t_2 terms)
- **Satisfies-Relation**
  Let (Σ, E) be a presentation, A be an algebra and $\delta : \Sigma \rightarrow A$ be a denotation. A satisfies the presentation (Σ, E) if and only if:
  - » $t_1 == t_2 \Rightarrow \delta(t_1) = \delta(t_2)$ for all ground substitutions of variables
- **Variety**
  Let (Σ, E) be a presentation. The variety V is the set of all algebras A which satisfy the presentation.

Example of a presentation

```
sorts Stack, Nat, Bool;
   operations
      true, false : -> Bool;
      zero : -> Nat;
      succ : Nat -> Nat;
      newstack : -> Stack;
      push: Stack x Nat -> Stack;
      isnewstack : Stack -> Bool;
      pop : Stack -> Stack;
      top : Stack -> Nat;
   declare s : Stack; n : Nat;
   axioms
      isnewstack(newstack) == true;
      isnewstack(push(s,n)) == false;
      pop(newstack) == newstack;
      pop(push(s,n)) == s;
      top(newstack) == zero;
      top(push(s,n)) == n;
```

equations

Relationships between algebras

- **Homomorphism**

  Let A and B be algebras for the signature Σ, i.e. there are denotations $\delta_A : \Sigma \rightarrow A$, $\delta_B : \Sigma \rightarrow B$. A homomorphism $h : A \rightarrow B$ is a set of functions $h_1, ..., h_m$ with the following properties:
  - » $h_i : \delta_A(sn_i) \rightarrow \delta_B(sn_i)$ (for all sort names)
  - » Let $fn : \rightarrow sn_i$ be a name of a nullary function. Then: $h_i(\delta_A(fn)) = \delta_B(fn)$
  - » Let $fn : sn_{j1} \text{ x } sn_{jk} \rightarrow sn_i$, $k > 0$. The following condition must hold for all suitably typed $s_{j1}, ..., s_{jk}$:
    $h_i(\delta_A(fn)(s_{j1}, ..., s_{jk})) = \delta_B(fn)(h_{j1}(s_{j1}), ..., h_{jk}(s_{jk}))$
- **Isomorphism**

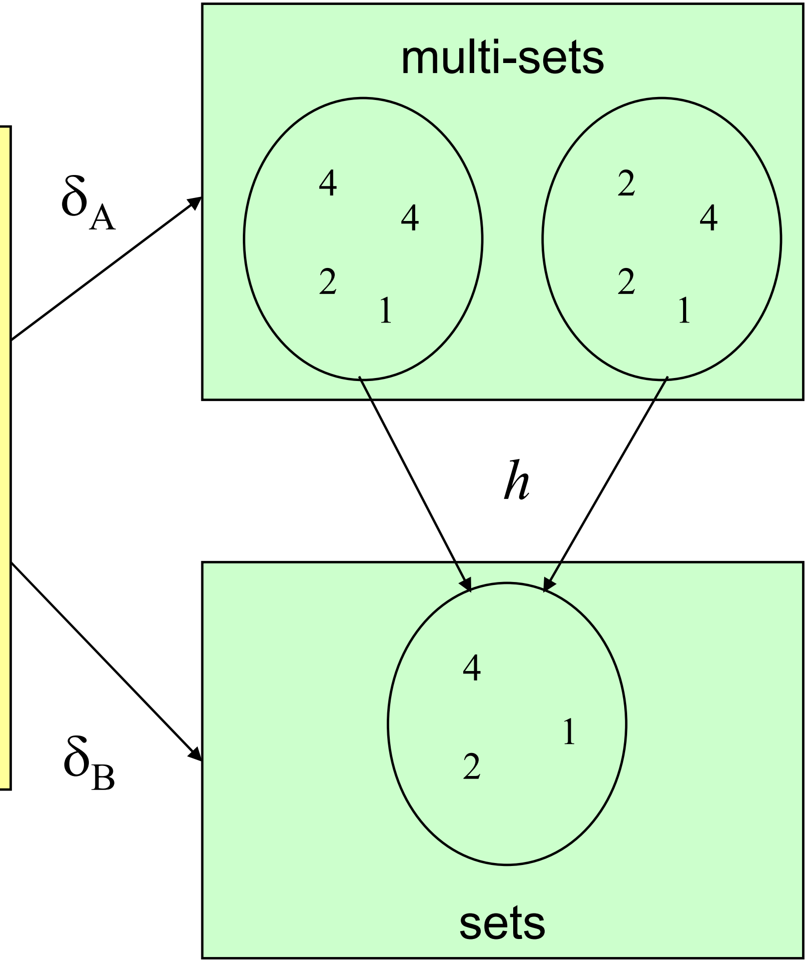
  An isomorphism is a bijective homomorphism.

Illustration by a commutative diagram

$$\begin{tikzcd}
A & B \\
(s_{j1}, \ldots, s_{jk}) \arrow[r, "h_{j1}, \ldots, h_{jk}"] \arrow[d, "\delta_A(fn)"] & (h_{j1}(s_{j1}), \ldots, h_{jk}(s_{jk})) \arrow[d, "\delta_B(fn)"'] \\
s_{iA} \arrow[r, "h_i"] & s_{iB}
\end{tikzcd}$$

Example: sets and multi-sets

```
sorts S, Nat, Bool;
   operations
      ...
      empty : -> S;
      insert : Nat x S -> S;
      isin : Nat x S -> Bool;
   declare n, n1, n2 : Nat; s : S;
   axioms
      insert(n1,insert(n2,s)) ==
         insert(n2,insert(n1,s));
      isin(n,empty) == false;
      isin(n1,insert(n2,s)) ==
         if eq(n1,n2)
         then true
         else isin(n1,s);
```



Initial and final algebras

- **Category**
  A category C consists of sets of algebras and homomorphisms such that:
  » $h_1 : A_1 \rightarrow A_2 \wedge h_2 : A_2 \rightarrow A_3 \Rightarrow$
  $h_1 \circ h_2 : A_1 \rightarrow A_3$ is a homomorphism and belongs to C
  » $(h_1 \circ h_2) \circ h_3 = h_1 \circ (h_2 \circ h_3)$
- **Initial algebra**
  Finest algebra of a category:
  » $I \in C \wedge A \in C \Rightarrow \exists\, h : I \rightarrow A$
- **Final algebra**
  Coarsest algebra of a category:
  » $F \in C \wedge A \in C \Rightarrow \exists\, h : A \rightarrow F$
- Initial and final algebra of a variety exist and are uniquely defined up to isomorphism

Construction of the initial algebra

- **Quotient algebra** of the word algebra:
  - Subsume all words representing equal terms in an equivalence class

"newstack"
"pop(push(newstack,zero))"
"pop(push(newstack,succ(zero)))"
"pop(pop(push(push(newstack,zero),zero)))"
...

equivalence class for the empty stack

"push(newstack,zero)"
"push(pop(push(newstack,zero)),zero)"
"push(pop(push(newstack,succ(zero))),zero)"
"push(pop(pop(push(push(newstack,zero),zero))),zero)"
...

equivalence class for the stack which contains the single element 0

Equation-based reasoning

Reflexivity

```
declare <declaration part>
axiom
  t == t;
```

Symmetry

```
declare <declaration part>
axiom
  t1 == t2;
```

⇓

```
declare <declaration part>
axiom
  t2 == t1;
```

Substitutability

```
declare x : S;<declaration part 1>
axiom
  t1 == t2;
declare <declaration part 2>
axiom
  t3 == t4;
```

⇓

```
declare <declaration part 1>
        <declaration part 2>
axiom
  t1[x/t3] == t2[x/t4];
```

Transitivity

```
declare <declaration part>
axiom
  t1 == t2;
  t2 == t3;
```

⇓

```
declare <declaration part>
axiom
  t1 == t3;
```

Example

```
declare s : Stack; n : Nat;
axiom
   top(push(s,n)) == n;
```

Reflexivity

```
declare s : Stack; x : Nat;
axiom
   push(s,x) == push(s,x);
```

Symmetry

```
declare s : Stack; n : Nat;
axiom
   n == top(push(s,n));
```

Substitutability

```
declare s : Stack; n : Nat;
axiom
   push(s,n) == push(s,top(push(s,n)));
```

Proofs by induction

- **Induction**

  A predicate $P(x)$ is proved as follows:
  - » P is proved for all elementary, i.e.
    $P[x/c]$ must hold for all constants c
  - » Assuming that P holds for a term t,
    it is proved that P also holds for $f(t)$ for each function f

Example

Presentation

```
sort Z;
  operations
    zero : -> Z;
    succ : Z -> Z;
    pre : Z -> Z;
    add : Z x Z -> Z;
  declare i, j : Z;
  axioms
    pre(succ(i)) == i;                     --1--
    succ(pre(i)) == i;                     --2--
    add(zero,i) == i;                      --3--
    add(succ(i),j) == succ(add(i,j)); --4--
    add(pre(i),j) == pre(add(i,j));    --5--
```

To demonstrate

```
  declare i : Z;
  axiom
    i == add(i,zero);
```

Example

Start of induction

Axiom 3

```
add(zero,i) == i
```

Substitution of `i` with `zero`

```
add(zero,zero) == zero
```

Symmetry

```
zero == add(zero,zero)
```

Example

Induction step (only for `i` $\Rightarrow$ `succ(i)`)

Induction assumption

`i == add(i,zero)`

Reflexivity

`succ(j) == succ(j)`

Substitution of `j`

`succ(i) == succ(add(i,zero))`

Axiom 4

`succ(i) == add(succ(i),zero)`

Modules

Module concept for algebraic specifications

- A specification is composed of reusable units (**modules**)
- Definition of **export** and **import interfaces**
- **Generic modules** with constrained genericity
- **Formal parameters** are **abstract modules**
- **Semantic** in addition to **syntactic constraints**

EBNF for modular specifications

```
<specification> = (<module>)+

<module> = "module" [<module name>] ";"
              [<import clause>]
              [<export clause>]
              [<sorts part>]
              [<operations part>]
              [<declarations part>]
              [<axioms part>]
           "end" "module" [<module name>] ";"

<import clause> = "import" (<item name list> "from" <module name list> ";")+

<export clause> = "export" (<item name list> ["from" <module name list>] ";")+

<item name list> =  <item name> ("," <item name>)*
                  | "all" ["except" <item name> ("," <item name>)*]

<item name> = <sort name> | <operation name>

<module name list> = <module name> ("," <module name>)*
```

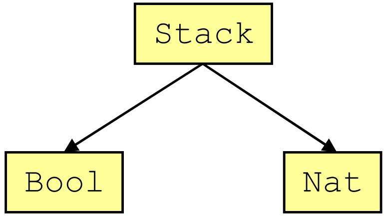
Examples of exports and imports

```
module Stack;
   import Bool, true, false from Bool;
      Nat, zero from Nat;
   export all;
   sort Stack;
   operations
      newstack : -> Stack;
      push : Stack x Nat -> Stack;
      isnewstack : Stack -> Bool;
      pop : Stack -> Stack;
      top : Stack -> Nat;
   declare s : Stack; n : Nat;
   axioms
      isnewstack(newstack) == true;
      isnewstack(push(s,n)) == false;
      pop(newstack) == newstack;
      pop(push(s,n)) == s;
      top(newstack) == zero;
      top(push(s,n)) == n;
end module Stack;
```

```
module Bool;
   export Bool, true, false;
   sort Bool;
   operations
      true, false : -> Bool;
end module Bool;
```

```
module Nat;
   export Nat, zero, succ;
   sort Nat;
   operations
      zero : -> Nat;
      succ : Nat -> Nat;
end module Nat;
```

Graphical representation



Semantic integrity constraints

- Let H be a hierarchy of modules, M be a new module which is added to H.
- **Consistency**: Two objects which were different in the initial algebra must not become equal by insertion of M, i.e.:
  If the equation $o_1 == o_2$ does not hold in H,
  then it must not hold in $H \cup M$.
- **Completeness**: Insertion of M must not involve the insertion of new objects, i.e.:
  If a term t belongs to the term language $H \cup M$ and its sort s is alreay present in H,
  then there is a term t' in H with $t == t'$.

Example of a consistent and complete addition

```
module ExtendedStack;
   import Nat, zero, succ from Nat;
      Stack, newstack, push, pop, top, isnewstack from Stack;
   export length from ExtendedStack;
      Stack, newstack, push, pop, top, isnewstack from Stack;
   operation
      length : Stack -> Nat;
   declare
      s : Stack; n : Nat;
   axioms
      length(newstack) == zero;
      length(push(s,n)) == succ(length(s));
end module ExtendedStack;
```

Example of an inconsistent and erroneous addition

```
module Bool;
   export all;
   sort Bool;
   operations
      true, false : -> Bool;
      and : Bool x Bool -> Bool;
   declare b : Bool;
   axioms
      and(true,true) == true;
      and(false,b) == false;
      and(b,false) == false;
end module Bool;

module ExtendedBool;
   import all from Bool;
   export all from Bool;
   axioms
      true == false;
end module Bool;
```

Example of an inconsistent yet meaningful addition

Multi-sets

```
module MultiSet;
   import all from Nat;
   export all;
   sort S;
   operations
      empty : -> S;
      insert : Nat x S -> S;
      isin : Nat x S -> Bool;
   declare n, n1, n2 : Nat; s : S;
   axioms
      insert(n1,insert(n2,s)) ==
         insert(n2,insert(n1,s));
      isin(n,empty) == false;
      isin(n1,insert(n2,s)) ==
         if eq(n1,n2)
         then true
         else isin(n1,s)
end module MultiSet;
```

Sets

```
module Set;
   import all from Nat, MultiSet;
   export all from MultiSet;
   declare n : Nat; s : S;
   axiom
      insert(n,insert(n,s)) ==
      insert(n,s)
end module Set;
```

Example of an incomplete yet meaningful addition

Binary logic

```
module BinaryLogic;
   export all;
   sort Bool;
   operations
      true, false : -> Bool;
      and : Bool x Bool -> Bool;
   declare b : Bool;
   axioms
      and(true,true) == true;
      and(false,b) == false;
      and(b,false) == false;
end module BinaryLogic;
```

Ternary logic

```
module TernaryLogic;
   import all from BinaryLogic;
   export all;
   operations
      unknown : -> Bool;
   declare b : Bool;
   axioms
      and(unknown,true) == unknown;
      and(true,unknown) == unknown;
end module TernaryLogic;
```

Parameterized specifications (genericity)

- **Reusability** of data types is increased by **formal parameters**
- Parameters are **formal modules**
- **Instantiations** of parameterized specifications yield abstract data types
- **Constrained genericity**: actual parameters must meet the requirements defined by formal modules
- **Semantic constraints**: axioms of formal modules must hold

EBNF for parameterized specifications

```
<scheme> = "scheme" <scheme name> ["["(<requirement>)+"]"]";"
              (<module>)+
           "end scheme" [<scheme name>]";"

<requirement> = "requirement" [<requirement name>] ";"
                   [<import clause>]
                   [<export clause>]
                   [<sorts part>]
                   [<operations part>]
                   [<declarations part>]
                   [<axioms part>]
                "end" "requirement" [<requirement name>] ";"

<instantiation> = "instantiate" <scheme name> [rename clause] ";"
                    ( "with" <requirement name> "as" <module name>
                         ("," <item name> "as" <item name>)* ";" )+
                  "end" "instantiate" [<scheme name>] ";"

<rename clause> = "rename"
                    <item name> "as" <item name>
                    ("," <item name> "as" <item name>)*
```

Example of parameterized specifications

```
scheme StackScheme [
   requirement Item;
      export all;
      sort Item;
      operation error : -> Item;
   end requirement Item;
   ];

   module Stack;
      ...
   end module Stack;
end scheme StackScheme;
```

```
module Stack;
   import Bool, true, false from Bool;
      all from Item;
   export all;
   sort Stack;
   operations
      newstack : -> Stack;
      push: Stack x Item -> Stack;
      isnewstack : Stack -> Bool;
      pop : Stack -> Stack;
      top : Stack -> Item;
   declare s : Stack; it : Item;
   axioms
      isnewstack(newstack) == true;
      isnewstack(push(s,it)) == false;
      pop(newstack) == newstack;
      pop(push(s,it)) == s;
      top(newstack) == error;
      top(push(s,it)) == it;
end module Stack;
```

Example of an instantiation of a parameterized specification

Instantiation clause

```
instantiate StackScheme;
   with Item as Nat,
      error as zero;
end instantiate StackScheme;
```

Instantiated specification

```
module Stack;
   import Bool, true, false from Bool;
      all from Nat;
   export all;
   sort Stack;
   operations
      newstack : -> Stack;
      push: Stack x Nat -> Stack;
      isnewstack : Stack -> Bool;
      pop : Stack -> Stack;
      top : Stack -> Nat;
   declare s : Stack; it : Nat;
   axioms
      isnewstack(newstack) == true;
      isnewstack(push(s,it)) == false;
      pop(newstack) == newstack;
      pop(push(s,it)) == s;
      top(newstack) == zero;
      top(push(s,it)) == it;
end module Stack;
```

Another example of a parameterized specification (1)

```
scheme ArrayScheme [
   requirement Attribute; (* For array elements *)
      export all;
      sort Attribute;
      operation error : -> Attribute;
   end requirement Attribute;

   requirement Index; (* For indices *)
      import Bool, true, _ and _ from Bool;
      export all;
      sort Index;
      operation
         _ = _ : Index x Index -> Bool; (* Infixnotation *)
      declare i, i1, i2, i3 : Index;
      axioms
         i = i == true; (* Reflexivity *)
         i1 = i2 == i2 = i1; (* Symmetry *)
         (i1 = i2) and (i2 = i3) => (i1 = i3) == true;
            (* Transitivity *)
   end requirement Index; ]

   module Array ...
end scheme StackArrayScheme;
```

Another example of a parameterized specification (2)

```
module Array;
   import Bool, true, false, not _ from Bool; all from Attribute, Index;
   export all;
   sort Array;
   operations empty : -> Array;
      _[_/_] : Array x Attribute x Index -> Array;
         (* Replacement of an array element *)
      isundefined : Array x Index -> Bool;
      read : Array x Index -> Attribute;
   declare ar : Array; i, i1, i2 : Index; at, at1, at2 : Attribute;
   axioms
      not (i1 = i2) => ar[at1/i1][at2/i2] == ar[at2/i2][at1/i1];
      ar[at1/i][at2/i] == ar[at2/i];
      isundefined(empty,i) == true;
      isundefined(ar[at/i1],i2) ==
         if i1 = i2 then false else isundefined(ar,i2) end if;
      read(empty,i) == error;
      read(ar[at/i1],i2) ==
         if i1 = i2 then at else read(ar,i2) end if;
end module Array;
```

Constructive Specifications

Rapid prototyping with constructive specifications

- Implementation of an abstract data type by a **term rewriting system**
- Separation between **constructors** for building up objects and other **operations**
- Equations for operations are interpreted from left to right as **term rewrite rules**
- Additional constraints must hold for constructive specifications
- Constructive specifications are **operational** and thus less abstract than non-constructive ones
- **Axioms** of non-constructive specifications become **theorems** of constructive specifications

Example: stack

```
scheme StackScheme [
    requirement Item;
        export all;
        sort Item;
        operation error : -> Item;
    end requirement Item;
    ];

    module Stack;
        ...
    end module Stack;
end scheme StackScheme;
```

```
module Stack;
    import Bool, true, false from Bool;
        all from Item;
    export all;
    sort Stack;
    constructors
        newstack : -> Stack;
        push : Stack x Item -> Stack;
    operations
        isnewstack : Stack -> Bool;
        pop : Stack -> Stack;
        top : Stack -> Item;
    declare s : Stack; it : Item;
    operation axioms
        isnewstack(newstack) == true;
        isnewstack(push(s,it)) == false;
        pop(newstack) == newstack;
        pop(push(s,it)) == s;
        top(newstack) == error;
        top(push(s,it)) == it;
end module Stack;
```

Constructors

Operations

Operation axioms

Examples of term rewriting

```
pop(push(pop(push(push(newstack,5),7)),9)) ==
    pop(push(s,n)) == s
pop(push(push(newstack,5)),9)) ==
    pop(push(s,n)) == s
push(newstack,5)

isnewstack(pop(push(pop(push(newstack,5)),7))) ==
    pop(push(s,n)) == s
isnewstack(pop(push(newstack),7))) ==
    pop(push(s,n)) == s
isnewstack(newstack) ==
    isnewstack(newstack) == true
true

pop(push(newstack,top(push(newstack,8)))) ==
    top(push(s,n)) == n
pop(push(newstack,8)) ==
    pop(push(s,n)) == s
newstack
```

Constraints for constructive specifications

- The outermost operation of a left-hand side of an axiom is no constructor, all inner operations are constructors
- A variable occurs at most once on the left-hand side
- All variables of the right-hand side occur on the left-hand side
- The system of axioms is **unique** with respect to a (non-constructor) operation, i.e. for each tuple of argument terms there is at most one matching rule
- The system of axioms is **complete** with respect to a (non-constructor) operation, i.e. for each tuple of argument terms there is at least one matching rule
- The system of axioms is **terminating**, i.e. for variable-free terms there are only derivations of finite length

Axioms and theorems

```
module Bool;
    export all;
    sort Bool;
    constructors true, false : -> Bool;
    operations
        not _ : Bool -> Bool; _ and _ : Bool x Bool -> Bool;
        _ or _ : Bool x Bool -> Bool; _ => _ : Bool x Bool -> Bool;
        _ <= _ : Bool x Bool -> Bool; _ <=> _ : Bool x Bool -> Bool;
    declare b, b1, b2, b3 : Bool;
    operation axioms
        not true == false; not false == true;
        b and true == b; b and false == false;
        b or true == true;  b or false == b;
        true => b == b; false => b == true;
        b <= true == b; b <= false == true;
        true <=> b == b; false <=> b == not b;
    theorems
        b and b == b; b or b == b;
        b1 and b2 == b2 and b1; b1 or b2 == b2 or b1;
        b1 and (b1 or b2) == b1;
        b1 or (b1 and b2) == b1;
        b and not b == false; b or not b == true;
        ...
    end module Bool;
```

Axioms

Theorems

Semi-constructive specifications

- Often, operation axioms do not suffice to specify the semantics of an abstract data type
- Thus, **constructor axioms** are added to make the initial algebra “sufficiently coarser”
- The semantics of operations are still specified only by operation axioms
- Constructor axioms are used only to prove that objects are equal
- Constructor axioms must not allow for non-terminating derivations $\Rightarrow$ equality is decidable

Example: sets

```
module Set;
  import Bool, true, false from Bool;
  all from Item;
  export all;
  sort Set;
  constructors
    Ø : -> Set;
    insert : Item x Set -> Set;
  operations
    delete : Item x Set -> Set;
    { _ } : Item -> Set;
    _ ∪ _ : Set x Set -> Set;
    _ ∩ _ : Set x Set -> Set;
    isin : Item x Set -> Bool;
  declare
    s, s1, s2 : Set;
    it, it1, it2 : Item;
  constructor axioms
    insert(it1,insert(it2,s)) ==
      insert(it2,insert(it1,s));
    insert(it,insert(it,s)) ==
      insert(it,s);
```

Constructor axioms

```
  operation axioms
    delete(it,Ø) == Ø;
    delete(it1,insert(it2,s)) ==
      if it1 = it2
        then delete(it1,s)
        else insert(it2,delete(it1,s))
      end if;
    {it} == insert(it,Ø);
    s ∪ Ø == s;
    s1 ∪ insert(it,s2) ==
      insert(it,s1 ∪ s2);
    s ∩ Ø == Ø;
    s1 ∩ insert(it,s2) ==
      if isin(it,s1)
        then insert(it,s1 ∩ s2)
        else s1 ∩ s2
      end if;
    isin(it,Ø) == false;
    isin(i1,insert(it2,s)) ==
      if it1 = it2
        then true
        else isin(it1,s)
      end if;
end module Set;
```

Example: proof of equality by constructor axioms

Problem: Are the following sets equal?

`s1 = {0,1,2,3,0}, s2 = {3,2,1,0}`

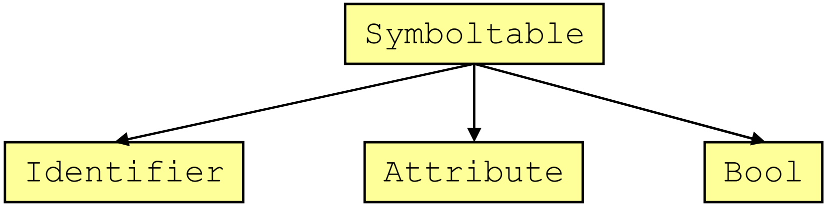
```
insert(0,insert(1,insert(2,insert(3,insert(0,Ø))))) ==
    insert(it1,insert(it2,s)) == insert(it2,insert(it1,s))
insert(1,insert(0,insert(2,insert(3,insert(0,Ø))))) ==
    insert(it1,insert(it2,s)) == insert(it2,insert(it1,s))
insert(1,insert(2,insert(0,insert(3,insert(0,Ø))))) ==
    insert(it1,insert(it2,s)) == insert(it2,insert(it1,s))
insert(1,insert(2,insert(3,insert(0,insert(0,Ø))))) ==
    insert(it,insert(it,s)) == insert(it,s)
insert(1,insert(2,insert(3,insert(0,Ø)))) ==
    insert(it1,insert(it2,s)) == insert(it2,insert(it1,s))
insert(2,insert(1,insert(3,insert(0,Ø)))) ==
    insert(it1,insert(it2,s)) == insert(it2,insert(it1,s))
insert(2,insert(3,insert(1,insert(0,Ø)))) ==
    insert(it1,insert(it2,s)) == insert(it2,insert(it1,s))
insert(3,insert(2,insert(1,insert(0,Ø))))
```

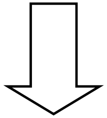
Abstract Implementations

Abstract implementations: goals and approach

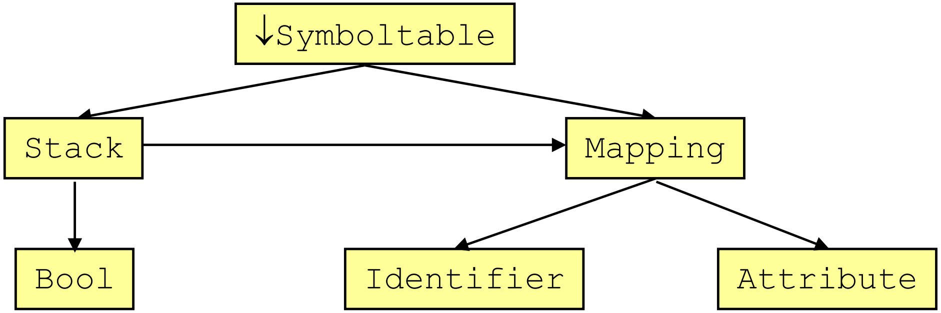
- Starting point: algebraic specifications for abstract data types on a high level of abstraction
- Goal: efficient implementation
- Approach: step-wise **refinement** of specifications, i.e. replacement of abstract with increasingly concrete data types
- Result: abstract implementation (not “real” because base types are only specified)

Example of step-wise refinement (1)



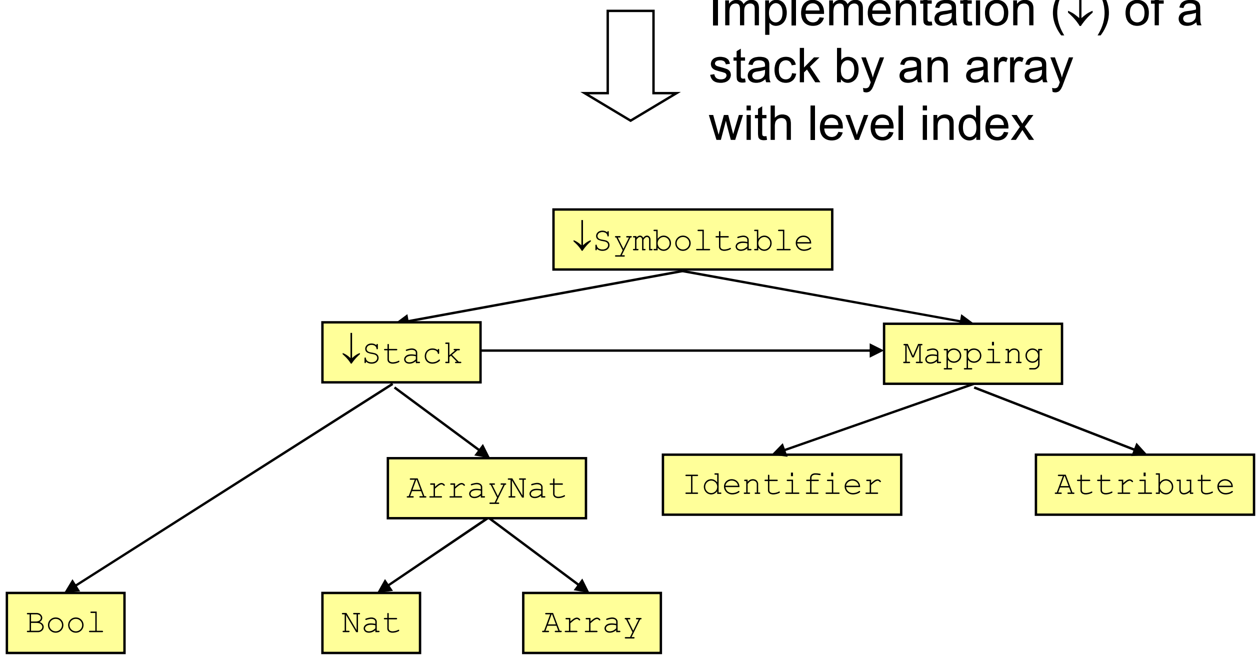


Implementation (↓) of a symbol table by a stack of mappings



Example of step-wise refinement (2)

Implementation (↓) of a stack by an array with level index



Specification of a stack

```
module Stack;
   import Bool, true, false from Bool;
      Nat, zero from Nat rename Nat as Item, zero as error;
   export all;
   sort Stack;
   constructors
      newstack : -> Stack;
      push: Stack x Item -> Stack;
   operations
      isnewstack : Stack -> Bool;
      pop : Stack -> Stack;
      top : Stack -> Item;
   declare s : Stack; it : Item;
   operation axioms
      isnewstack(newstack) == true;
      isnewstack(push(s,it)) == false;
      pop(newstack) == newstack;
      pop(push(s,it)) == s;
      top(newstack) == error;
      top(push(s,it)) == it;
end module Stack;
```

Implementation of a stack

```
module ↓Stack;
  import ArrayNat, (_,_), arrayOf _, natOf _ from ArrayNat;
    Array, empty, _[_/_], read from Array; Bool from Bool;
    Nat, zero, succ, pre, _ = _, _ < _ from Nat
      rename Nat as Item, zero as 0, zero as error, succ as _+1, pre as _-1;
  operations
    ↓newstack : -> ArrayNat;
    ↓push : ArrayNat x Item -> ArrayNat;
    ↓pop : ArrayNat -> ArrayNat;
    ↓top : ArrayNat -> Item;
    ↓isnewstack : ArrayNat -> Bool;
  declare an : ArrayNat; it : Item;
  operation axioms
    ↓newstack == (empty,0);
    ↓push(an,it) == (arrayOf an[it/natOf an],natOf an + 1);
    ↓pop(an) ==
      if natOf an = 0 then an else (arrayOf an,natOf an - 1) end if;
    ↓top(an) ==
      if natOf an = 0 then error else read(arrayOf an, natOf an - 1) end if;
    ↓isnewstack(an) == natOfan = 0;
end module Stack;
```

Array with level index

```
module ArrayNat; (* Record composed of an array and a natural number. *)
   import Array from Array; Nat from Nat;
   export all;
   sort ArrayNat;
   constructor (_,_) : Array x Nat -> ArrayNat;
   operations
      arrayOf _ : ArrayNat -> Array; (* Projection on first component *)
      natOf _ : ArrayNat -> Nat; (* Projection on second component *)
      _[_/array] : ArrayNat x Array -> ArrayNat; (* Replace first comp. *)
      _[_/nat] : ArrayNat x Nat -> ArrayNat; (* Replace second comp. *)
   declare a : Array; n : Nat; an : ArrayNat;
   operation axioms
      arrayOf((a,n)) == a;
      natOf((a,n)) == n;
      an[a/array] == (a,natOf an);
      an[n/nat] == (arrayOf an,n);
end module ArrayNat;
```

Abstract implementation (definition)

Let A and $\downarrow A$ be modules. $\downarrow A$ is an **implementation** of $A \Leftrightarrow$

- A defines a sort S, $\downarrow A$ defines (or imports) a sort $\downarrow S$
- **Data representation**: each A-constructor is mapped into an $\downarrow A$-operation
- **Procedure implementation**: each A-procedure is mapped into an $\downarrow A$-operation
- **Representation function**: each A-Term is mapped into an $\downarrow A$-term
- **Implementation invariant**: condition met by all $\downarrow S$-objects which implement S-objects
- **Abstraction function**: function which maps each $\downarrow S$-object meeting the implementation invariant into the corresponding S-object
- **Equivalence function**: defines $\downarrow S$-objects as equivalent which are mapped onto the same S-object
- Several constraints to be defined later are satisfied

Remarks

- No explicit distinction between module interface and module body (Modula-3 or Ada), but definition of an implementation relation between two modules A und $\downarrow A$
- Data representation and procedure implementation jointly define the representation function
- Multiple $\downarrow S$-objects may be mapped into the same S-object
- The equivalence relation on $\downarrow S$-objects cannot be defined by term equivalence ==, rather in general it is coarser than == and is specifically defined for the implementation relation

Data representation: definition

Let C be the set of constructors in A, O the set of operations in $\downarrow A$.
The **data representation** d is a signature-preserving function
$d : C \to O$ such that:

- For each nullary constructor $c : \to S$:
  $d(c) : \to \downarrow S$
- For each constructor $c : S_1 \times \ldots \times S_n \to S$ $(n \geq 1)$:
  $d(c) : f(S_1) \times \ldots \times f(S_n) \to \downarrow S$, where
  $f(S_i) = \downarrow S$ if $S_i = S$
  $f(S_i) = S_i$ otherwise

(analogous definition for procedure implementation $p : P \to O$)

Example of data representation and procedure implementation

```
module Stack;
   ...
   sort Stack; (* Sort S *)
   constructors
      newstack : -> Stack;
      push: Stack x Item -> Stack;
   operations
      pop : Stack -> Stack;
      top : Stack -> Item;
      isnewstack : Stack -> Bool;
   ...
end module Stack;
```

```
module ↓Stack;
  import ArrayNat ... from ArrayNat;
    (* Imported sort  ↓S *)
  ...
  operations
    ↓newstack : -> ArrayNat;
    ↓push : ArrayNat x Item -> ArrayNat;
    ↓pop : ArrayNat -> ArrayNat;
    ↓top : ArrayNat -> Item;
    ↓isnewstack : ArrayNat -> Bool;
  ...
end module Stack;
```

Representation function: definition and example

Let T be a set of terms, d, p be a data representation and a procedure implementation, respectively. The induced **representation function** is a function $r : T \rightarrow T$ which eventually replaces all operations of A by operations of $\downarrow A$:

- $r(f(t_1,...,t_n)) =$
  $d(f)(r(t_1),..., r(t_n))$ if f is an A-constructor
  $p(f)(r(t_1),..., r(t_n))$ if f is an A-procedure
  $f(r(t_1),..., r(t_n))$ otherwise ($n \geq 0$)

```
r(pop(push(newstack,10))) =
p(pop)(r(push(newstack,10))) =
↓pop(r(push(newstack,10)))
↓pop(d(push)(r(newstack),r(10))) =
↓pop(↓push(r(newstack),r(10))) =
↓pop(↓push(d(newstack),10)) =
↓pop(↓push(↓newstack,10))
```

Implementation invariant: definition and example

An **implementation invariant** is a Boolean function $I : \downarrow S \rightarrow$ Bool which all $\downarrow S$-objects meet which serve as implementations of S-objects.

```
operation I : ArrayNat -> Bool;
declare an : ArrayNat;
operation axiom
   I(an) == alldefined(arrayOf an,natOf an);
   (* All array elements up to the level index must be defined. *)

operation alldefined : ArrayNat x Nat -> Bool;
declare a : Array; n : Nat;
operation axiom
   alldefined(a,n) ==
      if n = 0
         then true
         else
            if isundefined(a,n-1)
               then false
               else alldefined(a,n-1)
            end if
      end if;
```

Abstraction function: definition and example

An **abstraction function** is a function $@ : \downarrow S \rightarrow S$ which maps each $\downarrow S$-object into the S-object which it represents.
(@ must be defined for all $\downarrow S$-objects which meet the implementation invariant I.)

```
operation @ : ArrayNat -> Stack;
declare a : Array; n : Nat;
operation axiom
   @((a,n)) ==
      if n = 0
         then newstack
         else push(@((a,n-1)),read(a,n-1))
      end if;
```

Equivalence relation: definition and example

An **equivalence relation** is a reflexive, transitive, and symmetric relation ~ which determines for two ↓*S*-objects whether they represent the same abstract *S*-object.
(~ must be defined for all ↓*S*-objects which satisfy the implementation invariant *I*.)

```
operation _~_ : ArrayNat x ArrayNat -> Bool;
declare
   an, an1, an2, an3 : ArrayNat; a1, a2 : Array; n1, n2 : Nat;
operation axiom
   (a1,n1) ~ (a2,n2) ==
      if n1 = n2 then
         if n1 = 0 then true
         else (read(a1,n1-1) = read(a2,n2-1)) and (a1,n1-1) ~ (a2,n2-1)
         end if
      else false
      end if;
theorems
      an ~ an == true; (* Reflexivity *)
      an1 ~ an2 == an2 ~ an1; (* Symmetry *)
      an1 ~ an2 and an2 ~ an3 => an1 ~ an3 == true; (* Transitivity *)
```

Implementation constraints (1)

- The implementation operations of ↓*A* must be closed with respect to the implementation invariant *I*.

```
declare an : ArrayNat; it : Item;
theorem I(an) => I(↓push(an,it)) == true;
```

- The composition of representation function and abstraction function yields the identity (with respect to term equivalence ==).

```
@(r(pop(push(newstack,it))) =
@(↓pop(↓push(↓newstack,it))) =
@(↓pop(↓push((empty,0),it))) =
@(↓pop((empty[it/0],1))) =
@((empty[it/0],0)) =
newstack ==
pop(push(newstack,it))
```

Implementation constraints (2)

- If two *A*-terms are equal, their representations are equivalent.

```
pop(push(newstack,it)) == newstack ⇒
r(pop(push(newstack,it))) =
... =
(empty[it/0],0) ~
(empty,0) =
r(newstack)
```

- If two ↓*A*-terms satisfying the implementation invariant are equivalent, then their abstractions are equal.

```
(empty[it/0],0) ~ (empty,0) ⇒
@((empty[it/0],0)) =
newstack =
@((empty,0))
```

Implementation constraints (3)

- The composition of abstraction function and representation function yields the identity (with respect to the equivalence relation ~).

```
I((empty[it/0],0)) ⇒
r(@((empty[it/0],0))) =
r(newstack) =
(empty,0) ~
(empty[it/0],0)
```

- An *A*-term which does not have the sort *S* delivers the same value as its representation.

```
r(isnewstack(push(newstack,it))) =
↓isnewstack(↓push(↓newstack,it)) =
↓isnewstack((empty[it/0],1)) ==
false ==
isnewstack(push(newstack,it))
```

Conclusion

Advantages of algebraic specifications

- Very general approach to the specification of abstract data types
- Behavioral specification which completely abstracts from the implementation
- Formal proofs of properties of abstract data types may be conducted
- Support of rapid prototyping for constructive specifications
- Step-wise refinement from a high-level specification down to the implementation

Disadvantages of algebraic specifications

- For the specification of equations an operational mental model is usually required
- Proofs are laborious, error-prone and can be automated only partially
- Application to large software systems difficult, lack of scalability
- No built-in type constructors (arrays, records, etc. must be specified explicitly)
- No connection to a programming language (code generation)
- Complicated theory (see e.g. refinements)

Literature

- I. van Horebeek, J. Lewi: **Algebraic Specifications in Software Engineering**, Springer-Verlag (1991)
  Book on which this chapter is based. To the best of my knowledge, this is the only book which treats algebraic specifications from the perspective of software engineering.
- H. Ehrig, B. Mahr: **Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics**, EATCS Monographs on Theoretical Computer Science, Springer-Verlag (1985)
  Fundamental, but very theoretical book on algebraic specifications.
- E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner (Hrsg.): **Algebraic Foundations of Systems Specification**, IFIP State-of-the-Art-Report, Springer-Verlag, 1-12 (1999)
  Collection of papers which provides an overview of the current state of the art in research.