Algebraic Specifications
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
- \( domN: FN \to SN^* \) domain
- \( ranN: FN \to SN \) 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
- \( dom: F \to S^* \) domain
- \( ran: F \to S \) range
Denotation (mapping syntactic $\rightarrow$ semantic domain)

$\delta : \Sigma \rightarrow A$

- $\delta: sn_i \rightarrow S_j$ ($\delta$ maps each sort name into a sort)
- $\delta: fn_i \rightarrow F_j$ (analogously for function names)
- $\text{dom}(\delta(fn_i)) = \delta(\text{domN}(fn_i))$, $\text{ran}(\delta(fn_i)) = \delta(\text{ranN}(fn_i))$

(domain and range “are preserved”)

Denotation
syntactic domain

sort Nat;
operations
  zero : Nat;
  succ : Nat -> Nat;

semantic domain

Nat

denotation

succ(n) = n+1
0 ... 2
... 1

zero

succ(n) = n+1
0 ... -2
-1 2

successor

succ(n) = 0
0
Terms

Term language
Let $\Sigma = <SN, FN, domN, ranN>$ 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 $fn: sn_1 \times \cdots \times sn_k \rightarrow sn$ ($k \geq 1$), $t_1, \ldots, t_k$ 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

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

strings

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

**Example:** \( \delta(push)("newstack", "zero") = "push(newstack, zero)" \)
Substitution

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

\[
\sigma : X \rightarrow T(\Sigma), \text{ where } x \text{ and } \sigma(x) \text{ must have the same type}
\]

**Ground substitution**
Substitution of variables by variable-free terms

\[
push(x, y) \quad y \rightarrow succ(zero) \\
push(x, succ(zero)) \quad x \rightarrow push(newstack, zero) \\
push(push(newstack, zero), succ(zero))
\]
Properties of abstract data types

Presentation
A presentation \((\Sigma, E)\) is a signature \(\Sigma\), combined with a set of equations \(E\).
Each equation \(e \in E\) is built up as follows:
\(t_1 \equiv t_2 \ (t_1, t_2 \text{ terms})\)

Satisfies-Relation
Let \((\Sigma, E)\) be a presentation, \(A\) be an algebra and \(\delta: \Sigma \to A\) be a denotation. \(A\) satisfies the presentation \((\Sigma, E)\) if and only if:
\(t_1 \equiv t_2 \implies \delta(t_1) = \delta(t_2)\) for all ground substitutions of variables

Variety
Let \((\Sigma, E)\) 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;
```

```
Relationships between algebras

Homomorphism

Let $A$ and $B$ be algebras for the signature $\Sigma$, 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 $h_1, \ldots, h_m$ with the following properties:

- $h_i : \delta_A(sn_i) \rightarrow \delta_B(sn_i)$ (for all sort names)
- Let $f_n : \rightarrow sn_i$ be a name of a nullary function. Then:
  
  \[ h_i(\delta_A(fn)) = \delta_B(fn) \]

- Let $f_n : sn_{j1} x sn_{jk} \rightarrow sn_i$, $k > 0$. The following condition must hold for all suitably typed $s_{j1}, \ldots, s_{jk}$:
  
  \[ h_i(\delta_A(fn)(s_{j1}, \ldots, s_{jk})) = \delta_B(fn)(h_{j1}(s_{j1}), \ldots, h_{jk}(s_{jk})) \]

Isomorphism

An isomorphism is a bijective homomorphism.
Illustration by a commutative diagram

\[ A \xrightarrow{\delta_A(fn)} s_{iA} \xrightarrow{h_i} s_{iB} \xrightarrow{\delta_B(fn)} B \]

\[ (s_{j_1}, \ldots, s_{j_k}) \xrightarrow{h, \ldots, h} (h_{j_1}(s_{j_1}), \ldots, h_{j_k}(s_{j_k})) \]
Example: sets and multi-sets

sorts S, Nat, Bool;
operations
  ...
  empty : -> S;
  insert : Nat x S -> S;
  isin : Nat x S -> Bool;
decare 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:
- $h_1 : A_1 \rightarrow A_2 \land h_2 : A_2 \rightarrow A_3 \Rightarrow h_1 \circ h_2 : A_1 \rightarrow A_3$ 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:
- $I \in C \land A \in C \Rightarrow \exists h : I \rightarrow A$

Final algebra
Coarsest algebra of a category:
- $F \in C \land A \in C \Rightarrow \exists h : A \rightarrow F$

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

equivalence class for the empty stack

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

equivalence class for the stack which contains the single element 0

```
"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)))zero)"
...
```
Equation-based reasoning

Reflexivity

declare <declaration part>
axiom
t == t;

Substitutability

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

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

Symmetry

declare <declaration part>
axiom
t1 == t2;

declare <declaration part>
axiom
t2 == t1;

Transitivity

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

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

Reflexivity

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

Symmetry

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

Substitutability

```plaintext
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 $f$
Spécifications Formelles

Example

Presentation

```plaintext
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

```plaintext
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 $i \Rightarrow \text{succ}(i)$)

Induction assumption

\[ i = \text{add}(i, \text{zero}) \]

Reflexivity

\[ \text{succ}(j) = \text{succ}(j) \]

Substitution of $j$

\[ \text{succ}(i) = \text{succ}(\text{add}(i, \text{zero})) \]

Axiom 4

\[ \text{succ}(i) = \text{add}(\text{succ}(i), \text{zero}) \]
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

```plaintext
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

```
Stack
   /|
  / |
Bool Nat
```
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 already present in $H$, then there is a term $t'$ in $H$ with $t = t'$. 
Example of a consistent and complete addition

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

```ebnf
<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

\[ \text{instantiate StackScheme;} \]
\[ \quad \text{with Item as Nat,} \]
\[ \quad \text{error as zero;} \]
\[ \quad \text{end instantiate StackScheme;} \]

Instantiated specification

\[ \text{module Stack;} \]
\[ \quad \text{import Bool, true, false from Bool;} \]
\[ \quad \text{all from Nat;} \]
\[ \quad \text{export all;} \]
\[ \quad \text{sort Stack;} \]
\[ \quad \text{operations} \]
\[ \quad \quad \text{newstack : } \to \text{ Stack;} \]
\[ \quad \quad \text{push: Stack } \times \text{ Nat } \to \text{ Stack;} \]
\[ \quad \quad \text{isnewstack : Stack } \to \text{ Bool;} \]
\[ \quad \quad \text{pop : Stack } \to \text{ Stack;} \]
\[ \quad \quad \text{top : Stack } \to \text{ Nat;} \]
\[ \quad \text{declare } \text{s : Stack;} \text{ it : Nat;} \]
\[ \quad \text{axioms} \]
\[ \quad \quad \text{isnewstack(newstack) } = \text{ true;} \]
\[ \quad \quad \text{isnewstack(push(s, it)) } = \text{ false;} \]
\[ \quad \quad \text{pop(newstack) } = \text{ newstack;} \]
\[ \quad \quad \text{pop(push(s, it)) } = \text{ s;} \]
\[ \quad \quad \text{top(newstack) } = \text{ zero;} \]
\[ \quad \quad \text{top(push(s, it)) } = \text{ it;} \]
\[ \quad \text{end module Stack;} \]
Another example of a parameterized specification (1)

```plaintext
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;
```
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

```plaintext
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) == s;  
  top(push(s,it)) == error;  
  top(push(s,it)) == it;  
end module Stack;
```
Examples of term rewriting

\[
\begin{align*}
\text{pop}(\text{push}(\text{pop}(\text{push}(\text{push}(\text{newstack},5),7)),9)) &= \\
&\quad \text{pop}(\text{push}(s,n)) = s \\
\text{pop}(\text{push}(\text{push}(\text{newstack},5)),9)) &= \\
&\quad \text{pop}(\text{push}(s,n)) = s \\
\text{push}(\text{newstack},5) \\
\text{isnewstack}(\text{pop}(\text{push}(\text{pop}(\text{push}(\text{newstack},5)),7))) &= \\
&\quad \text{pop}(\text{push}(s,n)) = s \\
\text{isnewstack}(\text{pop}(\text{push}(\text{newstack}),7))) &= \\
&\quad \text{pop}(\text{push}(s,n)) = s \\
\text{isnewstack}(\text{newstack}) &= \\
&\quad \text{isnewstack}(\text{newstack}) = \text{true} \\
\text{true} \\
\text{pop}(\text{push}(\text{newstack},\text{top}(\text{push}(\text{newstack},8)))) &= \\
&\quad \text{top}(\text{push}(s,n)) = n \\
\text{pop}(\text{push}(\text{newstack},8)) &= \\
&\quad \text{pop}(\text{push}(s,n)) = s \\
\text{newstack}
\end{align*}
\]
**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.
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;
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.
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);
Example: proof of equality by constructor axioms

Problem: Are the following sets equal?
\[ s_1 = \{0,1,2,3,0\}, \ s_2 = \{3,2,1,0\} \]

\[
\text{insert}(0,\text{insert}(1,\text{insert}(2,\text{insert}(3,\text{insert}(0,\emptyset))))) = \\
\text{insert}(1,\text{insert}(0,\text{insert}(2,\text{insert}(3,\text{insert}(0,\emptyset)))))) = \\
\text{insert}(1,\text{insert}(2,\text{insert}(0,\text{insert}(3,\text{insert}(0,\emptyset)))))) = \\
\text{insert}(1,\text{insert}(2,\text{insert}(3,\text{insert}(0,\text{insert}(0,\emptyset)))))) = \\
\text{insert}(\text{insert}(1,\text{insert}(\text{insert}(1,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(2,\text{insert}(3,\text{insert}(1,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(2,\text{insert}(1,\text{insert}(3,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(2,\text{insert}(3,\text{insert}(1,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(2,\text{insert}(1,\text{insert}(3,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(\text{insert}(2,\text{insert}(1,\text{insert}(3,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(\text{insert}(2,\text{insert}(3,\text{insert}(1,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(\text{insert}(2,\text{insert}(1,\text{insert}(3,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(\text{insert}(2,\text{insert}(3,\text{insert}(1,\text{insert}(0,\emptyset)))) = \\
\text{insert}(\text{insert}(\text{insert}(2,\text{insert}(1,\text{insert}(3,\text{insert}(0,\emptyset)))) = 
\]
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

Diagram:
```
  ↓Symboltable
  ↓Stack
  ↓ArrayNat
  ↓Bool

  ↓Nat

  ↓ArrayNat
  ↓Identifier
  ↓Attribute
```

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
\[
\begin{align*}
\downarrow \text{newstack :} & \quad \rightarrow \quad \text{ArrayNat}; \\
\downarrow \text{push :} & \quad \text{ArrayNat} \times \text{Item} \quad \rightarrow \quad \text{ArrayNat}; \\
\downarrow \text{pop :} & \quad \text{ArrayNat} \quad \rightarrow \quad \text{ArrayNat}; \\
\downarrow \text{top :} & \quad \text{ArrayNat} \quad \rightarrow \quad \text{Item}; \\
\downarrow \text{isnewstack :} & \quad \text{ArrayNat} \quad \rightarrow \quad \text{Bool};
\end{align*}
\]

declare an : ArrayNat; it : Item;
operation axioms
\[
\begin{align*}
\downarrow \text{newstack} & \quad \begin{array}{c}
\text{==} \\
\text{\text{(empty,0);}}
\end{array} \\
\downarrow \text{push(an,it)} & \quad \begin{array}{c}
\text{==} \\
\text{(arrayOf an[it/natOf an],natOf an + 1);}
\end{array} \\
\downarrow \text{pop(an)} & \quad \begin{array}{c}
\text{==} \\
\text{\text{if natOf an = 0 then an else (arrayOf an,natOf an - 1) end if;}}
\end{array} \\
\downarrow \text{top(an)} & \quad \begin{array}{c}
\text{==} \\
\text{\text{if natOf an = 0 then error else read(arrayOf an, natOf an - 1) end if;}}
\end{array} \\
\downarrow \text{isnewstack(an)} & \quad \begin{array}{c}
\text{==} \\
\text{natOfan = 0;}
\end{array}
\end{align*}
\]

end module Stack;
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 $A$ if:

- $A$ 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 $A$ and $\downarrow A$

- 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
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 $d : C \rightarrow O$ such that:

- For each nullary constructor $c : \rightarrow S$:
  $$d(c) : \rightarrow \downarrow S$$

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

(analogous definition for procedure implementation $p : P \rightarrow O$)
Example of data representation and procedure implementation

```plaintext
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;
```
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 $\downarrow A$:

$$r(f(t_1,\ldots,t_n)) =$$
$$d(f)(r(t_1),\ldots, r(t_n)) \text{ if } f \text{ is an } A\text{-constructor}$$
$$p(f)(r(t_1),\ldots, r(t_n)) \text{ if } f \text{ is an } A\text{-procedure}$$
$$f(r(t_1),\ldots, r(t_n)) \text{ otherwise (n } \geq 0)$$
An implementation invariant is a Boolean function $I : \downarrow S \rightarrow \text{Bool}$ which all $\downarrow S$-objects meet which serve as implementations of $S$-objects.

```plaintext
operation $I$ : ArrayNat -> Bool;
declare an : ArrayNat;
operation axiom
  $I(\text{an}) == \text{alldefined(}\text{arrayOf an, natOf an})$;
  (* 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;
```

---

**Note:** The example code snippet is presented in a structured format to enhance readability and comprehension. The operations and axioms are defined with clear variable declaration and function definitions, ensuring that the implementation invariant is a feasible and practical solution for the given context.
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. ($\downarrow$ must be defined for all $\downarrow S$-objects which meet the implementation invariant $I$.)

```plaintext
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 \( \downarrow S \)-objects whether they represent the same abstract \( S \)-object.
(~ must be defined for all \( \downarrow S \)-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 $\downarrow A$ must be closed with respect to the implementation invariant $I$.

```
declare an : ArrayNat; it : Item;
theorem I(an) => I(\downarrow 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)))) ==
@((\downarrow pop(\downarrow push(\downarrow newstack, it)))) ==
@((\downarrow pop(\downarrow push((empty, 0), it)))) ==
@((\downarrow pop((empty[it/0], 1)))) ==
@((empty[it/0], 0)) ==
newstack ==
pop(push(newstack, it))
```
If two $A$-terms are equal, their representations are equivalent.

\[
\text{pop(push(newstack, it)) == newstack} \Rightarrow \\
r(\text{pop(push(newstack, it)))} = \\
... = \\
(\text{empty[it/0], 0}) \sim \\
(\text{empty, 0}) = \\
r(\text{newstack})
\]

If two $\downarrow A$-terms satisfying the implementation invariant are equivalent, then their abstractions are equal.

\[
(\text{empty[it/0], 0}) \sim (\text{empty, 0}) \Rightarrow \\
@((\text{empty[it/0], 0})) = \\
\text{newstack} = \\
@((\text{empty, 0}))
\]
The composition of abstraction function and representation function yields the identity (with respect to the equivalence relation ~).

$$I((\text{empty}[it/0], 0)) \Rightarrow r(@((\text{empty}[it/0], 0))) = r(\text{newstack}) = (\text{empty}, 0) \sim (\text{empty}[it/0], 0)$$

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

$$r(\text{isnewstack}(\text{push}(\text{newstack}, it))) = \downarrow\text{isnewstack}(\downarrow\text{push}(\downarrow\text{newstack}, it)) = \downarrow\text{isnewstack}((\text{empty}[it/0], 1)) = \text{false} = \text{isnewstack}(\text{push}(\text{newstack}, it))$$
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

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.

Fundamental, but very theoretical book on algebraic specifications.

Collection of papers which provides an overview of the current state of the art in research.