Formale Techniken in der Software-Entwicklung

Equational Specification in Maude

Christian Prehofer

Based on material from Martin Wirsing

SS 2011

Goals

- Introduce algebraic specifications
- Write first specifications with Maude
- Order-sorted signatures and specifications
- Membership equational logic

An Initial Algebra Specification: Natural Numbers

```
fmod NAT-PREFIX is
   sort Natural .

   op 0 : -> Natural .
   op s : Natural -> Natural .
   op plus : Natural Natural -> Natural .

   vars N M : Natural .

   eq plus(N,0) = N .
   eq plus(N,s(M)) = s(plus(N,M)) .
endfm
```

Algebraic Specifications

Definition:

Let $\Sigma = (S, F)$ be a signature and E a set of (closed) Σ-formulas.

- SP=<Σ,E> is called an algebraic specification.
- If E is a set of equations, SP is called an equational specification.

Moreover, depending on the semantics we distinguish loose specification and initial algebra specifications:

- The semantics of a loose specification SP is given by the class of all models of SP:

  $$\mathrm{Mod}(SP) =_{def} \{ A \in \mathrm{Alg}(SP) \mid A \models E \}$$

- The semantics of an initial algebra specification SP is given by all initial models of SP:

  $$I(SP) =_{def} \{A \in \mathrm{Mod}(SP) \mid A \text{ is initial in } \mathrm{Mod}(SP)\}$$

Maude

- Maude is an executable specification language for equational specifications and term rewriting.
- Maude is being developed by Jose Meseguer and his group at Univ. of Illinois and by the group of Carolyn Talcott at SRI.
- You can download Maude 2.6 from the Maude web page http://maude.cs.uiuc.edu. Chapter 2 in the Maude 2.6 manual (also in that web page) explains how you start Maude and interact with it.

Josè Meseguer
Prof. UIUC
PhD Zaragoza

Carolyn Talcott
SRI
PhD Stanford

Maude Functional Modules and Theories

In Maude,

- A **loose specification** is called **theory**, declared with the syntax

```
th <name> is (Σ,E) endth
```

  Maude theories are not executable!

- An **initial specification** is called **functional module**, declared with syntax

```
fmod <name> is (Σ,E) endfm
```

Maude Theories (Loose Specifications)

- The **trivial theory** consisting of one sort

```
fth TRIV is
  sort Elt .
endfth
```

- The theory of **partial orderings**

```
fth PARTIAL-ORDER is
  protecting BOOL .
  including TRIV .
  op _<=_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ceq X <= Z = true if X <= Y and Y <= Z [nonexec label transitive] .
  ceq X = Y         if X <= Y /\ Y <= X  [nonexec label antisymmetric] .
   eq X <= X = true                      [nonexec label reflexive] .
endfth
```

Maude Theories (Loose Specifications)

- The theory of **groups**

```
fth GROUP is
  sorts Group .
  op e : -> Group .
  op _o_ : Group Group -> Group .
  op _-1  : Group -> Group .

  vars X Y Z : group .

  eq e o X = X                          [nonexec label identity] .
  eq (X o Y) o Z = X o (Y o Z)          [nonexec label associative] .
  eq X o X-1 = e                        [nonexec label idempotent] .
endfth
```

Maude Theories (Loose Specifications)

Or shorter:

Associativity, commutativity, and identity axioms can be abbreviated in Maude by annotating the signature:

associativity axiom	`assoc`
commutativity axiom	`comm`
identity axiom w.r.t a constant e	`id: e`

```
fth GROUP is
  sorts Group .
  op e : -> Group .
  op _o_ : Group Group -> Group   [assoc id: e] .
  op _-1  : Group -> Group .

  vars X Y Z : group .

  eq X o X-1 = e                  [nonexec label idempotent] .
endfth
```

Natural Numbers (prefix syntax)

```
fmod NAT-PREFIX is
   sort Natural .

   op 0 : -> Natural .
   op s : Natural -> Natural .
   op plus : Natural Natural -> Natural .

   vars N M : Natural .

   eq plus(N,0) = N .
   eq plus(N,s(M)) = s(plus(N,M)) .
endfm

Maude> red plus(s(s(0)),s(s(0))) .
reduce in NAT-PREFIX : plus(s(s(0)), s(s(0))) rewrites: 3 in
-10ms cpu (0ms real) (~rewrites/second)
result Natural: s(s(s(s(0))))
Maude>
```

Natural Numbers (mixfix syntax)

```
fmod NAT-MIXFIX is
    sort Natural .

    op 0 : -> Natural .
    op s_ : Natural -> Natural .
    op _+_ : Natural Natural -> Natural .
    op _*_ : Natural Natural -> Natural .

    vars N M : Natural .

    eq N + 0 = N .
    eq N + s M = s(N + M) .
    eq N * 0 = 0 .
    eq N * s M = N + (N * M) .
endfm

Maude> red (s s 0) + (s s 0) .
reduce in NAT-MIXFIX : s s 0 + s s 0
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Natural: s s s s 0
```

Lists of Natural Numbers

```
fmod NAT-LIST is
   protecting NAT-MIXFIX .
   sort List .

   op nil : -> List .
   op _._ : Natural List -> List .
   op length : List -> Natural .

   var N : Natural .
   var L : List .

   eq length(nil) = 0 .
   eq length(N . L) = s length(L) .
endfm
```

```
Maude> red length(0 . (s 0 . (s s 0 . (0 . nil)))) .
reduce in NAT-LIST : length(0 . s 0 . s s 0 . 0 . nil)
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result Natural: s s s s 0
```

Some Common Mistakes

- not ending declarations for sorts, operators, etc. with a space followed by a period, e.g.,

```
sort Natural
op s : Natural -> Natural.
```

- not leaving spaces between a mixfix operator and its arguments, e.g., ~~`0+0`~~ .

- not putting enough parentheses to disambiguate expressions,
  e.g., `p s s 0 + 0 * s 0 .`

Constructors

- Often not all operations are needed to construct the elements of a data type. A **constructor** is an operation which contributes to the construction of the data elements of an (initial) algebra.
- **Examples:**
  - The operations 0 and s_ are constructors of sort Natural in NAT-MIXFIX
  - The operations nil and _._ are constructors of sort List in NAT-LIST
- **Formally:**

  A set C of operations is called **set of constructors of sort s** if for every element $a \in A_s$, there is an assignment $v : X \rightarrow A$ with $v(x) = a$ (and $x \in X_s$), variables $y_1,\ldots, y_n$ of sorts different from s, and a term $t \in T((S,C), \{y_1,\ldots, y_n\})$ s.t. $A, v \models x = t$
- In Maude a constructor operation is annotated by `[ctor]`.

Maude Specifications with Constructors

```
fmod NAT is
   sort Natural .
   op 0 : -> Natural [ctor].
   op s_ : Natural -> Natural [ctor].
   op _+_ : Natural Natural -> Natural .
   op _*_ : Natural Natural -> Natural .
   ...
endfm

fmod NAT-LIST is
   protecting NAT-MIXFIX .
   sort List .
   op nil : -> List [ctor].
   op _._ : Natural List -> List [ctor].
   op length : List -> Natural .
   ...
endfm
```

Maude Specifications with Constructors

- **Spielkarten**

```
fmod SPIELKARTE is

   sorts Wert Farbe Spielkarte .

   ops As 7 8 9 10 Bube Dame Koenig : -> Wert [ctor] .
   ops Karo Herz Pique Kreuz : -> Farbe [ctor] .

   op _-_ : Farbe Wert -> Spielkarte [ctor] .

   op wert : Spielkarte -> Wert .
   op farbe : Spielkarte -> Farbe .

   var W : Wert .    var F : Farbe .

   eq wert(F - W) = W .
   eq farbe(F - W) = F .

endfm
```

Specifying partial functions in total algebras?

Problem

- How to specify partial functions in a framework of algebras with total functions?
- Consider for example defining a function
  - `first` that takes the first element of a list of natural numbers, or
  - a predecessor function `p` that assigns to each natural number its predecessor.

  What can we do? If we define,

```
op first : List -> Natural .
op p_ : Natural -> Natural .
```

  we have then the awkward problem of having to define the values of `first(nil)` and of `p 0` , which in fact are undefined.

Order-sorted signatures

Solution:

Recognize that these functions are partial, but become total on **appropriate subsorts**

`NeList < List` of **nonempty lists**, and

`NzNatural < Natural` of **nonzero natural numbers**.

If we define,

```
op s_ : Natural -> NzNatural .
op _._ : Natural List -> NeList .
op first : NeList -> Natural .
op p_ : NzNatural -> Natural .
```

everything is fine.

Subsorts also allow us to **overload operator symbols**. For example,

`Natural < Integer` , and

```
op _+_ : Natural Natural -> Natural
op _+_ : Integer Integer -> Integer
```

Order-sorted Natural Numbers

```
fmod NATURAL-NAT3 is
   sorts Natural NzNatural .
   subsorts NzNatural < Natural .
   op 0 : -> Natural .
   op s_ : Natural -> NzNatural .
   op p_ : NzNatural -> Natural .
   op _+_ : Natural Natural -> Natural .
   op _+_ : NzNatural Natural -> NzNatural . --- subsort overloading
   vars N M : Natural .

   eq p s N = N .
   eq N + 0 = N .
   eq N + s M = s(N + M) .

   sort Nat3 .
   ops 0 1 2 : -> Nat3 .
   op _+_ : Nat3 Nat3 -> Nat3 [assoc comm id: 0] . --- ad-hoc
   eq 1 + 1 = 2 .
   eq 1 + 2 = 0 .
endfm
```

Order-sorted Lists

```
fmod NAT-LIST-II is
  protecting NATURAL .
  sorts NeList List .
  subsorts NeList < List .

  op nil : -> List .
  op _._ : Natural List -> NeList .
  op length : List -> Natural .
  op first : NeList -> Natural .

  var N : Natural .
  var L : List .

  eq length(nil) = 0 .
  eq length(N . L) = s length(L) .
  eq first(N . L) = N .
endfm
```

Order-sorted Signature (mathematically)

- An **order-sorted signature ("sortengeordnet")** Σ is a triple

  $\Sigma = ((S, F_{w,s})_{(w,s) \in S* \times S}, <)$,

  where $((S, F_{w,s})_{(w,s) \in S* \times S}, <)$ is an S-sorted signature, and where $<$ is a partial order relation on S called subsort inclusion.

  Note: Unless specified otherwise, by a signature in Maude we will always mean an order-sorted signature.
- Two sorts s and s′ are called **connected** ($s \equiv_{\leq} s'$), if

  $s \equiv s'$ or

  $s < s'$ or $s' < s$ or

  if there is s″ with $s \equiv_{\leq} s''$ and $s'' \equiv_{\leq} s'$
- When we have two operator declarations,

  $f : w \rightarrow s$, and $f : w' \rightarrow s'$,

  with w and w′ strings of equal length, then:

  (1) if $w \equiv_{\leq} w'$ and $s \equiv_{\leq} s'$, we call them **subsort overloaded**;

  (2) otherwise, we call them **ad-hoc overloaded**.

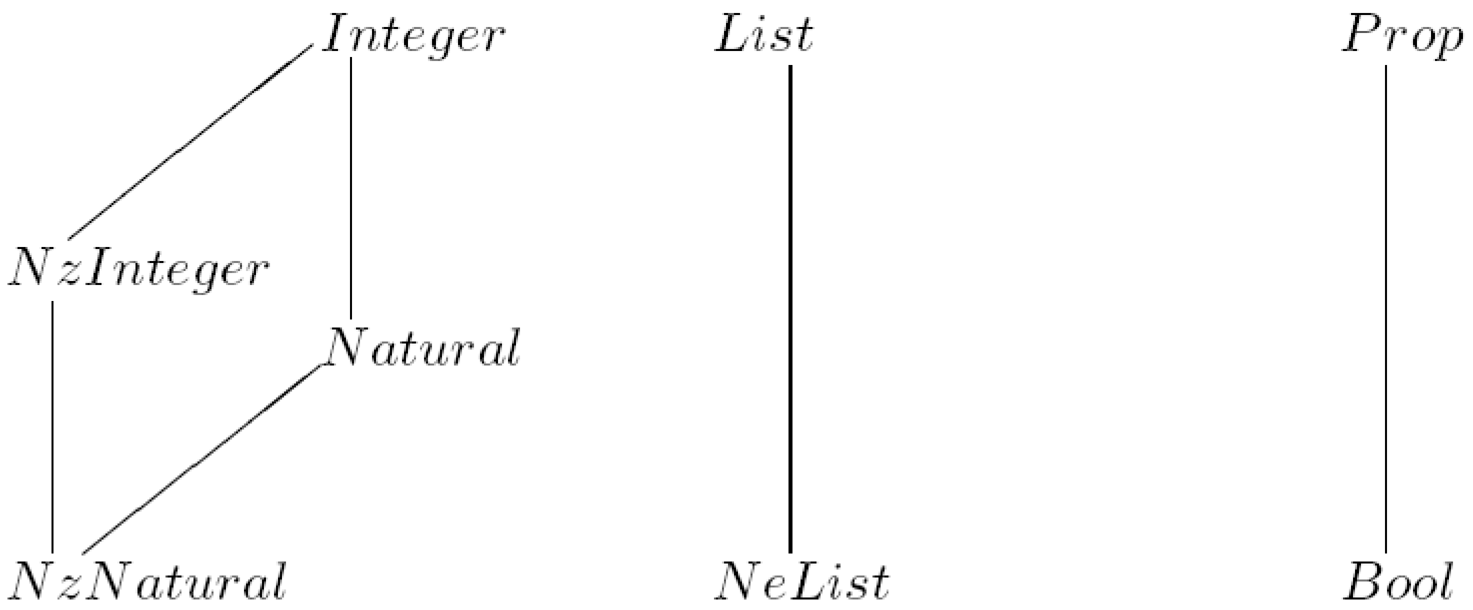
Connected Components

- Given a signature Σ, we can define an equivalence relation

  $\equiv_\leq$ between sorts s, s′ $\in$ S

  as the smallest relation such that:
  - if $s \leq s'$ or $s' \leq s$ then $s \equiv_\leq s'$
  - if $s \equiv_\leq s'$ and $s' \equiv_\leq s''$ then $s \equiv_\leq s''$
- We call the equivalence classes modulo $\equiv_\leq$ the **connected components** ("zusammenhängend") of the poset order $(S, \leq)$.
- Intuitively, when we view the poset as a directed acyclic graph, they are the connected components of the graph.

Connected Components Example



$S/\equiv_{\leq}$ =

{ Integer, NzInteger, Natural, NzNatural}, {NeList, List}, {Bool, Prop} }

Order-Sorted Algebras

Given an order-sorted signature $\Sigma = (S, \{ \mathsf{F}_{w,s} \}_{(w,s) \in S^* \times S}, <)$ an order-sorted Σ-algebra is defined as a many-sorted $(S, \{ \mathsf{F}_{w,s} \}_{(w,s) \in S^* \times S})$-algebra A such that:

- In $A = \{A_s\}_{s \in S}$, if $s < s'$ then $A_s \subseteq A_{s'}$
- if f is subsort overloaded, so that we have, $f : w \longrightarrow s$, and $f : w' \longrightarrow s'$, with w and w' strings of equal length, and with $w \equiv_{\leq} w'$ and $s \equiv_{\leq} s'$, then:
  - if $w = w' = nil$, then f is a constant and we have $f_A^{nil,s} = f_A^{nil,s'}$ (subsort overloaded constants coincide)
  - otherwise, if $(a_1, \ldots, a_n) \in A^w \cap A^{w'}$, then $f_A^{w,s}(a_1, \ldots, a_n) = f_A^{w',s'}(a_1, \ldots, a_n)$ (subsort overloadec operations agree)

Maude: Kind

- Order-sorted signatures are still restrictive:
  - **Example**

    `NzNat < Nat, _div_ : Nat NzNat -> Nat`

    Then
    - `(p s s 0)` is **not** in `NzNatural` and thus
    - `(s 0) div (p s s 0)` is not well-formed!
- **Kind („Art")**
  - A kind describes a connected component and is denoted by

    `[` „the topmost sort(s) of the component" `]`
  - **Examples:**

    [List] Kind of the List connected component

    [Integer] Kind of the Integer connected component

    `(s 0) div (p s s 0)` is of kind `[Nat]`
  - **Remark:**

    Terms that have a kind, but do not have a sort in e.g. [Integer] are thought of as error (or undefined) terms.

    So-called membership equational logic will give us a general way of dealing with partiality within the total context provided by the kinds.

Maude: Membership

- **Membership („Elementbeziehung")**
  - `t : s` asserts for any term `t` of kind `[s]` that the (interpretation of) `t` is an element of (the carrrier set of) sort `s`
- Membership allows one to **define subsorting and many-sorted signatures**:
  - `NzNat < Nat` corresponds to
    `cmb N : Nat if N : NzNat`
  - `_div_ : Nat NzNat -> Nat` corresponds to
    `_div_ : [Nat] [Nat] -> [Nat]`
    `cmb M div N : Nat if M : Nat /\ N : NzNat`
  - For `t :=((s 0) div (p s s 0))`
    `t:[Nat]` holds, but `t:Nat` does not hold.

Membership Equational Specification

- **Element Signature** $\Sigma = (K, S, F)$
  - Many-sorted signature (K, F) with kinds K
  - K-kinded family of sorts $S = (S_k)_{k \in K}$
- **Element Algebra** $A \in Alg()$
  - Many-sorted (K, F)-algebra A
  - Interpretation A_s of a sort s:

    if $s \in S_k$, then $A_s \subseteq A_k$
- **Element Equational Specification**
  - Conditional equational formulas (Horn formulas)

    $(\forall X)\ t = t' \Leftarrow (u1 = u'1 \wedge \ldots \wedge uk = u'k) \wedge (v1 : s1 \wedge \ldots \wedge vm : sm)$

    $(\forall X)\ t : s \Leftarrow (u1 = u'1 \wedge \ldots \wedge uk = u'k) \wedge (v1 : s1 \wedge \ldots \wedge vm : sm)$

Example Palindrome Lists

```
fmod PALINDROME is protecting QID .
  sorts Pal List .
  subsorts Qid < Pal < List .
  op nil : -> Pal [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  ops rev : List -> List .
  vars I : Qid .
  var P : Pal .
  var L : List .
  mb I P I : Pal .
  eq rev(nil) = nil .
  eq rev(I L) = rev(L) I .
endfm
```

- `QID` is the predefined module of „quoted identifiers" where every identifier is represented by an apostrophe followed by a string.
  - **Example**: `'abc` with underlying string `"abc"`

Example Pokerpaar

- Kartenpaar beim Poker

```
fmod KARTENPAAR is

   protecting SPIELKARTE .

   sorts Paar PokerPaar .
   subsort PokerPaar < Paar .

   op <_;_> : Spielkarte Spielkarte -> Paar [comm] .

   var  W : Wert .   var F F1 : Farbe .

    mb < F - W ; F1 - W > : PokerPaar .

endfm
```

Summary

- Maude is an executable language for equational specifications.
- Loose specifications are called theories, initial algebra specifications are called functional modules.
- In Maude partial functions are modelled by total functions on subsorts.
- Subsort overloading vs. ad-hoc overloading of functions.
- Equational membership specifications allow one to model any (equationally specifiable) predicate.