Maude Summer School: Lecture 1

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign and
Leverhulme visiting professor at King's College, London

Formal Specification and Verification in Maude

These lectures will introduce you to Maude. The four main ideas of Maude are that:

1. Programming in Maude is mathematical modeling using a computational logic as the programming language.

2. Computation is formal deduction in the computational logic.

3. The meaning of a program P is a mathematical model $\mathbb{C}_P$ in first-order logic, called the canonical model of P.

4. Saying that program P satisfies a formal property φ exactly means that $\mathbb{C}_P \models \varphi$ in the first-order logic sense.

High Performance, Tools, and Applications

Maude is a high-performance language. It ranked second, shortly after Haskell, in the 2018 INRIA-Grenoble competition among 14 languages.

Maude has a formal tool environment supporting both deductive and model checking verification of Maude programs.

Maude has been used in many applications worldwide. See:

- `http://maude.cs.illinois.edu/w/index.php/Applications`
- And for a (somewhat dated) survey: "J. Meseguer, "Twenty years of rewriting logic," J. Log. Algebr. Program. 81(7-8): 721–781 (2012).

Modeling Deterministic and Concurrent Systems

Maude can naturally model:

1. Deterministic systems in equational logic, in its sublanguage of functional modules.

2. Concurrent systems in rewriting logic, in its full language of system modules.

- Lectures 1 and 2 will focus on equational logic and functional modules.

- Lectures 3 and 4 will mainly focus on rewriting logic and system modules.

Equational Theories

Theories in equational logic are called equational theories. In Computer Science they are sometimes referred to as algebraic specifications.

An equational theory is a pair (Σ, E), where:

- Σ, called the signature, describes the syntax of the theory, that is, what types of data and what (typed) operation symbols (function symbols) are involved;
- E is a set of equations between expressions (called Σ-terms) in the syntax of Σ.

Unsorted, Many-Sorted, and Order-Sorted Signatures

Our syntax Σ can be more or less expressive, depending on how many types (called sorts) of data it allows, and what relationships between types it supports:

- unsorted (or single-sorted) signatures have only one sort, and operation symbols on it;
- many-sorted signatures allow different sorts, such as `Integer`, `Bool`, `List`, etc., and operation symbols relating these sorts;
- order-sorted signatures are many-sorted signatures that, in addition, allow subtype relations between sorts, such as `Natural < Integer < Rational`.

Maude Functional Modules

Maude functional modules are equational theories (Σ, E), declared with syntax

$$\texttt{fmod}\ (\Sigma, E)\ \texttt{endfm}$$

Such theories can be unsorted, many-sorted, or order-sorted.

In what follows we will see examples of unsorted, many-sorted and order-sorted equational theories (Σ, E) expressed as Maude functional modules, and of how such theories are used as functional programs that compute by equational deduction (replacement of equals for equals) with their equations E.

Unsorted Functional Modules

```
*** Natural number addition in prefix syntax

fmod NAT-PREFIX is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op + : Natural Natural -> Natural .
  vars N M : Natural .
  eq +(N,0) = N .
  eq +(N,s(M)) = s(+(N,M)) .
endfm

Maude> red +(s(s(0)),s(s(0))) .
reduce in NAT-PREFIX : +(s(s(0)), s(s(0))) .
result Natural: s(s(s(s(0))))
Maude>
```

Tracing Maude's Reduce Command

We can use Maude's trace facility to see how Maude uses equational deduction (replacement of equals for equals) in a left to right manner to evaluate functional expressions with the `reduce` command:

```
Maude> set trace on .
Maude> red +(s(s(0)),s(s(0))) .
reduce in NAT-PREFIX : +(s(s(0)), s(s(0))) .
*********** equation
eq +(N, s(M)) = s(+(N, M)) .
N --> s(s(0))
M --> s(0)
+(s(s(0)), s(s(0)))
--->
s(+(s(s(0)), s(0)))
*********** equation
```

```
eq +(N, s(M)) = s(+(N, M)) .
N --> s(s(0))
M --> 0
+(s(s(0)), s(0))
--->
s(+(s(s(0)), 0))
*********** equation
eq +(N, 0) = N .
N --> s(s(0))
+(s(s(0)), 0)
--->
s(s(0))

result Natural: s(s(s(s(0))))
```

Unsorted Functional Modules (II)

```
*** Natural's addition and multiplication in mixfix syntax

fmod NAT-MIXFIX is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s_ : Natural -> Natural [ctor] .
  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 .
result Natural: s s s s 0
Maude>
```

Many-Sorted Functional Modules

```
fmod NAT-LIST is
  protecting NAT-MIXFIX .
  sort List .
  op nil : -> List [ctor] .
  op _._ : Natural List -> List [ctor] .
  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) .
result Natural: s s s s 0
Maude>
```

The Need for Order-Sorted Signatures

Many-sorted signatures are still too restrictive. The problem is that some operations are partial, and there is no natural way of defining them in just a many-sorted framework.

Consider, division by 0, or 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? Declaring operators:

```
op _/_ : Rat Rat -> Rat .
op first : List -> Natural .
op p_ : Natural -> Natural .
```

we then have the awkward problem of defining the values of `1 / 0`, `first(nil)` and `p 0`, which in fact are undefined.

The Need for Order-Sorted Signatures (II)

These functions are partial with the typing just given, but become total on appropriate subsorts `NzRat < Rat` of nonzero rationals, `NeList < List` of nonempty lists, and `NzNatural < Natural` of nonzero naturals. If we declare,

```
op _/_ : Rat NzRat -> Rat .
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 Functional Modules

```
fmod NATURAL is
  sorts Natural NzNatural .
  subsorts NzNatural < Natural .
  op 0 : -> Natural [ctor] .
  op s_ : Natural -> NzNatural [ctor] .
  op p_ : NzNatural -> Natural .
  op _+_ : Natural Natural -> Natural .
  op _+_ : NzNatural NzNatural -> NzNatural .
  vars N M : Natural .
  eq p s N = N .
  eq N + 0 = N .
  eq N + s M = s(N + M) .
endfm

Maude> red p((s s 0) + (s s 0)) .
reduce in NATURAL : p (s s 0 + s s 0) .
result NzNatural: s s s 0
```

Order-Sorted Functional Modules (II)

```
fmod NAT-LIST-II is
  protecting NATURAL .
  sorts NeList List .
  subsorts NeList < List .
  op nil : -> List [ctor] .
  op _._ : Natural List -> NeList [ctor] .
  op length : List -> Natural .
  op first : NeList -> Natural .
  op rest : NeList -> List .
  var N : Natural .
  var L : List .
  eq length(nil) = 0 .
  eq length(N . L) = s length(L) .
  eq first(N . L) = N .
  eq rest(N . L) = L .
endfm
```

Rewriting as Efficient Equational Deduction

Maude computes with equations E from left to right. For example, in the second step of our trace:

```
*********** equation
eq +(N, s(M)) = s(+(N, M)) .
N --> s(s(0))
M --> 0
+(s(s(0)), s(0))  --->  s(+(s(s(0)), 0))
```

the subexpression `+(s(s(0)),s(0))` of the first step's result: `s(+(s(s(0)),s(0)))` has been matched as an instance of the equation's lefthand side `+(N,s(M)) = s(+(N,M))` with matching substitution $\{\texttt{N} \mapsto \texttt{s}(\texttt{s}(\texttt{0})), \texttt{M} \mapsto \texttt{0}\}$, which applied to the equation's righthand side yields the resulting subexpression `s(+(s(s(0)),0))` within: `s(s(+(s(s(0)),0)))`.

Rewriting as Efficient Equational Deduction (II)

This efficient form of equational deduction is called term rewriting, and is achieved as follows:

1. The equations E in (Σ, E) are oriented as a term rewriting system $(\Sigma, \vec{E})$, where $\vec{E} = \{u \to v \mid (u = v) \in E\}$ are called its rewrite rules.

2. A functional expression or term t is rewritten or simplified to t' with $\vec{E}$ in one step, written $t \to_{\vec{E}} t'$, iff there is a subterm w in t (notation: $t = t[w]$), a rule $(u \to v) \in \vec{E}$ and a substitution θ such that: (i) $w = u\theta$, (ii) $w' = v\theta$, and (iii) $t' = t[w'] = t[v\theta]$, where, $u\theta$ (resp. $v\theta$) denotes the instantiation of u (resp. v) with substitution θ.

Let us illustrate all this in our trace example.

Rewriting as Efficient Equational Deduction (III)

In our 2nd trace step for equation: `+(N,s(M)) = s(+(N,M))`

1. $t = \mathtt{s}(+(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{s}(\mathtt{0}))) = \mathtt{s}([+(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{s}(\mathtt{0}))])$
2. $\theta = \{\mathtt{N} \mapsto \mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{M} \mapsto \mathtt{0}\}$
3. $w = +(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{s}(\mathtt{0})) = +(\mathtt{N}, \mathtt{s}(\mathtt{M}))\theta$
4. $w' = \mathtt{s}(+(\mathtt{N}, \mathtt{M}))\theta = \mathtt{s}(+(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{0}))$, and
5. $t' = s([w']) = \mathtt{s}([\mathtt{s}(+(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{0}))]) = \mathtt{s}(\mathtt{s}(+(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{0})))$

yielding the one-step rewrite:

$t = \mathtt{s}([+(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{s}(\mathtt{0}))]) \rightarrow_{\vec{E}} \mathtt{s}([\mathtt{s}(+(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{0}))]) = t'$

with rule `+(N,s(M))` $\rightarrow$ `s(+(N,M))`.