Modeling and Analysis in Maude

Lecture 2: From equations to rules

Einar Broch Johnsen

University of Oslo, Norway
einarj@ifi.uio.no

DAT355, 07 April 2021

UNIVERSITY OF OSLO

SIRIUS Center for Scalable Data Access in the Oil and Gas Domain

sfi Centre for Research-based Innovation
The Research Council of Norway

http://www.sirius-labs.no

Plan for the lectures

- **Lecture 1:** Basic ideas and concepts of rewriting logic & Maude
- **Lecture 2:** From equations to rules
- **Lecture 3:** Modeling parallel & distributed systems
- **Lecture 4:** Analyzing models

From equations to rules

Summary from yesterday

Functional modules in Maude

- Define Sorts with constructors
- Other functions defined over the constructors
- How do we decide if two terms are equal?
- Termination, confluence, (unique) normal forms

Today

- Collections in Maude: lists, sets, multisets
- Rewriting logic
- Operational semantics
- Guarded commands

Exercises from yesterday

```
fmod NAT−ADD is
  sort Nat .
  op 0 : → Nat [ctor] .
  op s : Nat → Nat [ctor] .
  op _+_ : Nat Nat → Nat .

  vars M N : Nat .

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

Make a new module which extends NAT−ADD and

1. **op** double : Nat → Nat .
   double(0) $\leadsto$ 0
   double(s(s(s(0)))) $\leadsto$ s(s(s(s(s(s(0)))))).
   Do not use +, only 0 and s.
2. **op** half : Nat → Nat . half(0) $\leadsto$ 0
   half(2) $\leadsto$ 1
   half(5) $\leadsto$ 2
3. **op** monus : Nat Nat → Nat
   "minus down to 0," i.e., max(m−n,0).
4. **op** diff : Nat Nat → Nat .
   diff(2,7) $\leadsto$ 5, diff(8,1) $\leadsto$ 7.

Exercise

- **Prelude:** Maude's library of predefined sorts is automatically included in all modules.
- Let us use the module Nat from Prelude, which provides some prettyprinting for us (e.g., s(s(s(0))) is written as 3)

- Define a module LIST1 which includes Nat.
- Define a sort List1 for lists of natural numbers.
- Define length on List1
- Define concatenation on List1
- Define sum on List1

Lists with append

```
fmod LIST1 is protecting NAT .
  sort List .
  op nil : → List [ctor] .
  op app : List Nat → List [ctor] .
  op length : List → Nat .

  vars L L' : List . var N : Nat .

  eq length(nil) = 0 .
  eq length(app(L, N)) = 1 + length(L) .
endfm
```

- **Concatenation**

```
op concat : List List → List .
eq concat(nil,L) = L .
eq concat(app(L,N),L') =
     app(concat(L,L'),N) .
```

- **Sum**

```
op sum : List → Nat .
eq sum(nil) = 0 .
eq sum(app(L,N)) = N + sum(L) .
```

Lists with subsorts

```
fmod LIST2 is protecting NAT .
  sorts NeList List .
  subsort NeList < List .

  op nil :  → List [ctor] .
  op app : List Nat → NeList [ctor] .

  var L : List . var N : Nat .

  op length : List → Nat .
  eq length(nil) = 0 .
  eq length(app(L, N)) = 1 + length(L) .

  op head : NeList → Nat .
  op tail : NeList → List .
  eq head(app(L,N)) = N .
  eq tail(app(L,N)) = L .
endfm
```

- **Subsorts** allow us to define partial functions
- **Exercise:** Define head and tail

How Maude does lists

```
fmod LIST3 is protecting NAT .
sorts NeList List .
  subsort Nat < NeList < List .

 op nil : -> List [ctor] .
 op __ : List List -> List [ctor assoc id: nil] .
 op __ : NeList List -> NeList [ctor ditto] .
 op __ : List NeList -> NeList [ctor ditto] .

 vars L L' : List . var N M : Nat .
 op length : List -> Nat .
 eq length(nil) = 0 .
 eq length(N) = 1 .
 eq length(L L') = length(L) + length(L') .

 ops head tail : NeList -> Nat .
 eq head(N L) = N .
 eq tail (N L) = L .
endfm
```

- We can directly declare Nat as a subsort of NeList!
- We then use concat instead of app as the constructor for lists
- Use an invisible symbol (whitespace) for concat, to reduce visual noise
- Help Maude get more precise sorts
- Now try

```
red nil nil .
red length(4 5 6 8 9 10) .
red head(4 5 6 8 9) .
```

What went wrong?

- We need to tell Maude that
  - concat is associative and
  - concat has identity element nil !

Other Collections: Multiset and Set

What is the difference between a list, a multiset and a set?

Multisets

- the concat constructor is commutative

Sets

- Additionally, remove duplicate elements

```
fmod MSET is protecting NAT .
  sorts NeMSet MSet .
  subsort Nat < NeMSet < MSet .

  op none : -> MSet [ctor] .
  op __ : MSet MSet -> MSet [ctor assoc comm id: none] .
  op __ : NeMSet MSet -> NeMSet [ctor ditto] .
  op __ : MSet NeMSet -> NeMSet [ctor ditto] .
endfm

fmod SET is protecting NAT .
  sorts NeSet Set .
  subsort Nat < NeSet < Set .

  op none : -> Set [ctor] .
  op __ : Set Set -> Set [ctor assoc comm id: none] .
  op __ : NeSet Set -> NeSet [ctor ditto] .
  op __ : Set NeSet -> NeSet [ctor ditto] .

  var I : Nat . eq I I = I .
endfm
```

Rewriting Logic

- **Static model:** Algebraic specs / functional modules in Maude describe equivalence
- **Dynamic model:** Rewriting rules describe change

Example

- We want to model a person who gets older every year
- This problem is not about equivalence (age $\neq$ age + 1)
- We use rewrite rules to model ageing

```
fmod PEOPLE is
  protecting NAT . protecting STRING .

 sort Person .
 op person : String String Nat → Person [ctor] .

 *** red person("einar", "prof", 17) .
endfm
```

Rewrite Rules

```
mod BIRTHDAY is protecting PEOPLE .

  vars X Y : String . var N : Nat .

  rlcrl [birthday] :
      person(X, Y, N) ⟹ person(X, Y, N +1) if N < 1000 .

  crl [becoming-a-legend] :
      person(X, Y, N) => person(X, "legend", N) if Y =/= "legend" .

  *** rew person("einar", "prof", 17) .

endm
```

Configurations

Let us now create a multiset of People.

```
fmod PERSON is
  protecting NAT .
  protecting STRING .

  sorts Person NeMSet MSet .
  subsort Person < NeMSet < MSet .

  op person : String String Nat → Person [ctor] .

  op none : → MSet [ctor] .
  op __ : MSet MSet → MSet [ctor assoc comm id: none].
  op __ : NeMSet MSet → NeMSet [ctor ditto] .
  op __ : MSet NeMSet → NeMSet [ctor ditto] .

  op init : → MSet .
  eq init = person("einar", "prof", 17) person("volker", "prof", 21) .
endfm
```

Rewriting over multisets

- The rewrite rules can match with any term(s) in the multiset
- **AC-matching**: Maude reorders the elements in the multiset to match rules
- **Fair rewriting**: **frew** [n] init .

A Model of Parallel Programs

- How can we model a parallel program in Maude? Exercise for next week!
- To prepare the ground, let us look at how we can model the semantics of a simple imperative programming language in Maude.

What are the components of the semantics for imperative programs?

- Execution $\langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle \rightarrow \ldots$
- States σ map program variables to values
- Programs s consist of statements
- Statements may contain expressions (e.g., x = e, if e then s else s, …)

GCL

$$
\begin{array}{rcl}
Prog & ::= & \sigma \; \{ \; g_1 \; || \; \ldots \; || \; g_n \; \} \\
\sigma \in State & ::= & \epsilon \mid \sigma[x \mapsto v] \\
g \in GrdStm & ::= & \mathtt{skip} \mid g ; g \\
& \mid & e \rhd s \mid s \lhd e \rhd s \\
s \in Stm & ::= & x := e \mid \mathtt{spawn}(g) \\
e \in Exp & ::= & \mathtt{true} \mid \mathtt{false} \mid x \mid v \mid e + e \\
& \mid & e * e \mid e < e \mid e =:= e \\
& \mid & e \; and \; e \mid e \; or \; e \mid not(e)
\end{array}
$$

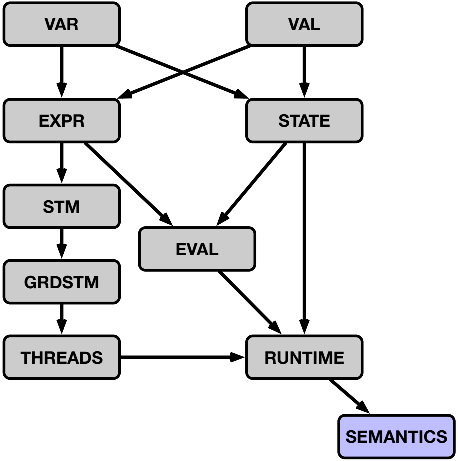
The Guarded Command Language (GCL), invented by Edsger Dijkstra, presents programming concepts in a compact way, before the program is written in some practical programming language. Its simplicity makes reasoning about programs easier.

Informal semantics of GCL

Goal: to make a Maude model of GCL

- We consider two types of values, defined by the sorts Bool and Int
- A runtime state $\sigma \; \{ \; g_1 \; || \; \ldots \; || \; g_n \; \}$ consists of a state σ and a "thread pool" with the threads $g_1, \ldots, g_n$
- We can execute a "thread" g_i from the thread pool if it is enabled in the current state (the guard evaluates to `true`)
  - The guarded statement **`skip`**: the thread can be terminated
  - The guarded statement $e \triangleright s$: if the guard e is true in σ, execute s
  - The guarded statement $s_1 \triangleleft e \triangleright s_2$: if the guard e is true in σ, execute s_1, otherwise execute s_2
  - The guarded statement $g_1 ; g_2$ schedules and executes g_1 as above, then adds g_2 to the thread pool
  - The statement $x := e$ evaluates e to some value v in the state σ, then updates x to v to produce the next state
  - The statement **`spawn`**(g) adds g to the thread pool

Structure of the model



- Gray box = fmod (equations)
- Blue box = mod (rules)

- **We shall now discuss**
  VAR, VAL, EXPR, STATE, EVAL
- **Assignment for next week**
  STM, THREADS, RUNTIME, SEMANTICS

Summary

Rewriting logic: Formal theory combining equivalence and change

- **Algebraic specification:** equivalence between terms
- **Term rewriting:** termination, confluence, (unique) normal forms
- **Modeling data types**: sort, subsorts, constructors, other functions
- **Attributes:** ctor, assoc, com, id: _
- **Collections:** Lists, sets, multisets

Maude: Language and tool for RL models

- **Modules:** fmod, mod, protecting, ...
- **Reduction** of terms to normal form: **red**
- **Rewriting** of terms: **rew, frew**
- **Prelude:** Maude's library