Modeling and Analysis in Maude

Lecture 1: Basic ideas and concepts of rewriting logic & Maude

Einar Broch Johnsen

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

DAT355, 6 April 2021

UNIVERSITY OF OSLO

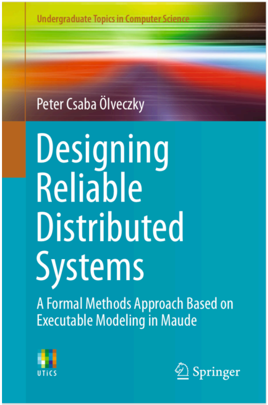
SIRIUS Center for Scalable Data Access in the Oil and Gas Domain

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

http://www.sirius-labs.no

What is Maude (1)

- Maude is a state-of-the-art formal specification language and analysis tool with a built-in notion of concurrency
- Developed at SRI International and Universities of Illinois and Madrid
- Real-Time Maude developed by Peter Ölveczky (UiO)
- Peter has also written a very good introductory book about Maude (recommended)



What is Maude (2)

- Executable modeling language which supports
  - Generality and ease of specification
  - High-level modeling / specification
  - A wide range of analysis techniques
- Solid mathematical foundation: rewriting logic
- User-friendly syntax
  - Makes models easy to understand for students and programmers w/o formal methods experience
  - Avoids unintuitive and error-prone encodings
- Maude is freely available at http://maude.cs.uiuc.edu

Go & get it now!

Plan for the lectures

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

Basic ideas and concepts of rewriting logic & Maude

Rewriting logic: Basic concepts

Rewriting logic combines two ideas in a novel way

- Algebraic specifications
- Term rewriting

Traditionally, both of these ideas aim to explain when terms are *equal*.
Rewriting logic expands on this equational view of term rewriting.

Maude

- Language for describing models in rewriting logic
- Tool for analysing these models

Algebraic specification

Algebraic specification languages

- Formalization of data types and of the operations on the values of these data types
- **Constructor functions:** functions that create or initialize the data elements
- **Additional functions:** functions that operate on the data types, and are defined in terms of the constructor functions

```
fmod NAT−ADD is
    sort Nat .

    op 0 : → Nat [ctor] .
    op s : Nat → Nat [ctor] .
endfm    op _+_ : Nat Nat → Nat .

    vars N M : Nat .

    — Recursive def. of plus
    eq 0 + N = N .
    eq s(N) + M = s (N + M) .
endfm

red s(s(0)) + (s(0) + 0) .
```

Here, Nat is a *sort*, 0 and s are *constructor* functions, $+$ is an additional *function*, and _+_ declares *mixfix*-notation for $+$.

New modules

Let us extend our model with multiplication.

```
fmod NAT−MULT is
  protecting NAT-ADD .
  op _*_ : Nat Nat → Nat .

  vars M N : Nat .

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

- Maude has a **module system**
- fmod = functional modules
- Protecting / including declaration used to import modules into new modules
- Modules and sorts can be parametrised
- **Groundterms** are built from constructors in a sort-correct way (i.e., no variables!)
- **Equational logic**: rules for deciding if $u = v$, given a set of equations

Term Rewriting

Term rewrite systems

- Mathematical model of non-deterministic behavior
- We are given a term and a set of rules
- Rules pattern match on subterms and transform these
- Many rules can match a given term

Example

Consider the TRS we get from NAT-MULT by "orienting" the equations:

Consider the term
$s(0 + 0) * s(0)$

How can we rewrite this term?

$$
\begin{array}{ll}
R1 & 0 + N \Rightarrow N \\
R2 & s(N) + M \Rightarrow s(N + M) \\
R3 & 0 * M \Rightarrow 0 \\
R4 & s(M) * N \Rightarrow N + (M * N)
\end{array}
$$

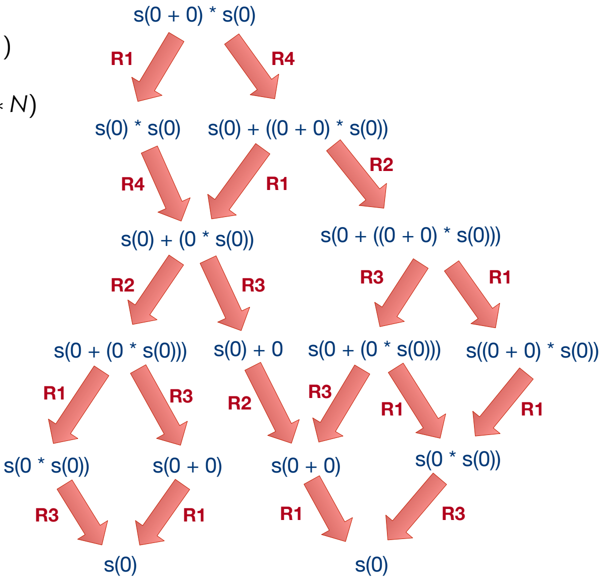
Reductions: Example

$R1 \quad 0 + N \Rightarrow N$

$R2 \quad s(N) + M \Rightarrow s(N + M)$

$R3 \quad 0 * M \Rightarrow 0$

$R4 \quad s(M) * N \Rightarrow N + (M * N)$



Properties of reductions (1)

Termination

- A TRS is *terminating* if it has no infinite sequence of reductions

$$t_0 \to t_1 \to t_2 \to \dots$$

- Termination is generally undecidable (Why?)
- Simple "trick" to have terminating equations:
  Ensure that RHS is "smaller" than LHS

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

- Maude assumes that your equations are terminating
  when ordered left to right

Properties of reductions (2)

- Recall the contructor terms correspond to the values ot our data types.
- When we apply a function, we want to end up with a value!
- The result of a reduction should be a constructor term

Definedness

Functions on a given sort must be defined for all constructor terms of that sort.

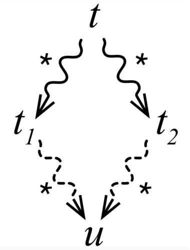
eq 0 + M = M .
eq s(M) + N = s(M + N) .

This ensures that the function can reduce when applied to all constructor terms.

- It is surprisingly easy to overlook a case, which could hinder the reduction of terms to constructor terms.
- Functions are often defined with one equation for each constructor:

op f : ... Nat ... → ...
eq f(..., 0, ...) = ...
eq f(..., s(N), ...) = ...

Properties of reductions (3)



Confluence

If a term t can be reduced to both terms t_1 and t_2, then there is a term u such that t_1 can be reduced to u and t_2 can be reduced to u.

Termination + confluence = unique normal forms
(i.e., unique results when computing functions)

- Maude assumes that your equations are confluent when ordered left to right
- If the equations are terminating and confluent, Maude can decide equality between terms by first reducing them to their unique normal forms.

Exercises for tomorrow

```
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. define a function **op** double : Nat → Nat . which doubles its argument. For example, double(0) should be 0 while double(s(s(s(0)))) should be s(s(s(s(s(s(0)))))). Do not use +, only 0 and s.
2. define a function **op** half : Nat → Nat . which divides a number by 2. For example, "half" of 0 is 0; "half" of 2 is 1; "half" of 3 is also 1; "half" av 4 is 2; "half" of 5 is 2. What is half of 86? of 87?
3. define a function **op** monus : Nat Nat → Nat which computes "minus down to 0," i.e., max(m− n,0).
4. define a function **op** diff : Nat Nat → Nat . which computes the difference between two numbers. For example, the diff between 2 and 7 is 5 and the diff between 8 and 1 is 7.

Summary

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

Tomorrow

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