Maude MSOS Tool

Fabricio Chalub and Christiano Braga

`frosario@ic.uff.br / cbraga@ic.uff.br`

Universidade Federal Fluminense

Outline

- Quick introduction to SOS and MSOS
- Overview of MSOS-SL and Maude MSOS Tool
- Rewriting logic
- MSOS-SL
- Example of formal verification with MMT
- Behind the scenes
- Developments and future work

Modularity in operational semantics (i)

- SOS = structural operational semantics, also known as small-step operational semantics
- Some practical problems with SOS: retracting previous rules.

  Semantics of $\bullet$ with an environment (ρ)

$$\frac{\rho \vdash e_0 \longrightarrow e_0'}{\rho \vdash e_0 \bullet e_1 \longrightarrow e_0' \bullet e_1}$$

  Semantics of $\bullet$ with an environment (ρ) and a store (σ, σ')

$$\frac{\rho \vdash \langle e_0, \sigma \rangle \longrightarrow \langle e_0', \sigma' \rangle}{\rho \vdash \langle e_0 \bullet e_1, \sigma \rangle \longrightarrow \langle e_0' \bullet e_1, \sigma' \rangle}$$

Modularity in operational semantics (ii)

Mosses' MSOS solves the modularity problem in structural operational semantics.

- Transition labels carry the semantic information associated with the computation.

$$\frac{e_0 \xrightarrow{x} e_0'}{e_0 \bullet e_1 \xrightarrow{x} e_0' \bullet e_1}$$

- Record components are environments, stores, etc., and are access using indices.

$$\frac{e \xrightarrow{\{\rho = \rho_1[\rho_0], \ldots\}} e'}{\textbf{let } \rho_0 \textbf{ in } e \textbf{ end} \xrightarrow{\{\rho = \rho_1, \ldots\}} \textbf{let } \rho_0 \textbf{ in } e' \textbf{ end}}$$

Maude MSOS Tool and MSOS-SL

MSOS-SL: the MSOS specification language, a conservative extension of Maude system modules.

MSOS-Tool: the MSOS-SL executable environment, written in Maude.

With the Maude MSOS Tool it is possible to give formally verifiable specifications for programming languages.

Maude and Rewriting Logic (RWL)

- A logical framework which can represent in a natural way many different logics, languages, operational formalisms, and models of computation;
- Parameterized by an equational logic, membership equational logic;
- Specifications in rewriting logic are executable with CafeOBJ, ELAN, and Maude;
- Formal verification tools available in Maude: model checker, breadth-first search, theorem prover, Church-Rosser checker, and termination checker;

MSOS-SL

The MSOS-SL semantics of a language $\mathcal{L}$ has three distinct parts:

- syntax definition: where we specify the (abstract/concrete) syntax of $\mathcal{L}$
- label declaration: where we specify the label composition.
- dynamic rules: where the dynamics of the language constructions are specified.

MSOS-SL modules

MSOS-SL modules are written as:

```
(msos MODULE is [...]  sosm)
```

MSOS-SL modules include other modules by the `including` keyword, such as:

```
(msos A is
 including B .
 including C .

 [...]
sosm)
```

MSOS-SL: syntax definition

Order-sorted logic is able to represent a context-free grammar (Goguen et al.)

The syntax definition in MSOS-SL is given in the algebraic way:

- Sorts are the counterpart of non-terminals.
- Term constructors are the counterpart of terminals.

(Infinite sets of terminals are, of course, represented by terms of some sort. For example, the natural numbers in Peano notation `s(s(s(s(0))))`, of sort `Nat`. Maude conveniently converts the Peano notation into decimal numbers.)

MSOS-SL: syntax definition

Syntax definition uses the Maude constructions: sort, subsort, op (for the declaration of operators).

Let us specify a simple ML-like **let-in-end**, as in:

```
let val x = 10 in x end
```

The **let** expression has two distinct parts: the declaration of bindings and the expression to be executed.

$$Exp ::= \texttt{let}\ \langle Dec\rangle\ \texttt{in}\ \langle Exp\rangle\ \texttt{end}$$

$\langle Dec\rangle$ and $\langle Exp\rangle$ will become sorts, and the `let-in-end` will became an operator.

MSOS-SL: syntax definition

We create a sort `Exp` for expressions and `Dec` for declarations in general.

```
sorts Dec Exp .
```

A **let** expression is declared as follows:

```
op let_in_end :  Dec Exp -> Exp [ctor] .
```

ctor means that this operation is not a function, but a constructor of terms. prec is the precedence we may assign to this operator.

Mixfix syntax (underscores).

MSOS-SL: syntax definition

Identifiers are terms of the sort Id.

```
sort Id .
```

Declarations are defined as bindings from identifiers to constants, obtained from the evaluation of expressions.

```
op val_ :  ValueBind -> Dec [ctor] .
```

Bindings are expressed as:

```
op _=_ :  Id Exp -> ValueBind [ctor] .
```

MSOS-SL: syntax definition

We declare the sort `Value` of the values expressible in our language. Since a value is also an expression, we have to subsort `Value` to `Exp`.

```
subsort Value < Exp .
```

By subsorting `Nat` to `Value` we make the naturals a primitive value of our programming language.

```
sort Value .
subsort Nat < Value .
```

We may now write:

```
op x : -> Id .
let val x = 100 in x end .
```

MSOS-SL: syntax definition

We may give equational attributes to operators, such as associativity, commutativity and identity to further enhance our syntax definition.

```
op _;_ :  Exp Exp -> Exp [ctor assoc prec 100] .
```

assoc indicates that this operation is associative
prec indicates the precedence level of this operation

More complex constructions are possible using frozen arguments, gather patterns (for example to create left-associative constructions), evaluation strategies (for example to create lazy-evaluation operations), and so on.

MSOS-SL: label declaration

Label indices are declared using the following keywords:

read-only i : τ .
read-write i : τ .
write-only i : τ $(\mathbf{e}, \mathbf{bop})$.

i is the index name, and τ the sort of the values indexed by i, referred to as *components*.

For WO indices, we must describe the monoid: identity element $(\mathbf{e})$ and binary operation $(\mathbf{bop})$.

```
read-only env : Env .
write-only out : Output (nil, append) .
```

MSOS-SL: components

Components are also specified as algebraic data types. In this example Env is the sort of environments, and BVal is the sort of "bindable values", along with associated operations.

```
sorts Env BVal .

op _U_ : Env Env -> Env .
op find : Env Ide -> [BVal] .
op _->_ : Ide BVal -> Env [ctor] .
op _//_ : Env Env -> Env .
...
```

Writing `[BVal]` as the image sort of `find` makes this a partial function.

MSOS-SL: transitions

MSOS transitions are declared with syntax `ctr`, as follows:

`ctr` γ `=` α `=>` γ' `if` $\langle condition \rangle$ `.`

γ is the value-added syntax tree. α is the label expression.

Unconditional transitions: `tr` γ `=` α `=>` γ' `.`

Unobservable transitions: γ `==>` γ'.

$\langle condition \rangle$: consists of a conjunction of transitions, written in the general form γ `=` α `=>` γ', together with the usual conditions from Maude system modules.

MSOS-SL: label expressions

Labels are formed by a set of *fields* of the form `(`i `:` C`)`.

The sort `IndexSet` is defined as a subsort of a `Label`. This opens the possibility to create *label expressions* as in MSOS.

`{(env : rho), (st : sigma), (st' : sigma'), IS}`, the variable `IS`, of sort `IndexSet`, matches against any unspecified component.

Unobservable labels are *identity labels* of the sort `ILabel`, a subsort of `Label`, and their subsets are of the sort `IIndexSet`, a subsort of `IndexSet`.

MSOS-SL: transitions

As an example, let us give the semantics of the **let** expression defined earlier:

```
var X        : Label .              var  IS           : IndexSet .
var v        : Value .              vars D D'         : Dec .
vars E1 E'1 E2 E'2 : Exp .          vars b rho rho' : Env .

ctr let D in E2 end = X => let D' in E2 end
if D = X => D' .

ctr let b in E end ={(env : rho), IS}=> let b in E' end
if rho' := rho // b /\ E ={(env : rho'), IS}=> E' .

tr let b in v end ==> v .
```

MSOS-SL: concurrency example

Syntax definition

```
sorts Prog Procs .
op  cml_ : Procs -> Prog [ctor] .
op  _||_ : Procs Procs -> Procs [ctor comm assoc] .
op  proc : PIde Exp -> Procs [ctor] .
ops spawn channel send recv : -> Value [ctor] .
```

The use of comm and assoc create an equivalent of a multiset.

Label declaration

```
read-write pides : PIdes .
write-only create : Create (nilc, appendc) .
read-write chans : Channels .
write-only offer : Offers (nilo, appendo) .
```

MSOS-SL: concurrency example

Transition rules

```
ctr (spawn f) = {(create' : C), (pides : PDS),
                 (pides' : PDS'), IIS} => PI
if PI := newPIde (PDS) /\
   PDS' := addPIde (PDS, PI) /\
   C := new-create (proc (PI, (f empty-tuple))) .

ctr proc (PI1, E1) ={(create' : nilc), IS }=>
    proc (PI1, E'1) || P
if E1 ={(create' : C), IS}=> E'1 /\ P := get1 (C) .

ctr P1 || P2 = X => P'1 || P2
if P1 = X => P'1 .
```

MSOS-SL: concurrency example

Transition rules

```
ctr channel empty-tuple
     ={ (chans : chs), (chans' : chs'), IIS}=> ch
if ch := newChannel (chs) /\
   chs' := addChannel (chs, ch) .

op snd : Channel Value -> Offer [ctor] .
op rcv : Channel -> Offer [ctor] .

ctr send tuple (ch, v) ={(offer' : O), IIS}=>
    empty-tuple
if O := new-offer (snd (ch, v)) .

op recv-ph : Channel -> Value [ctor] .
```

MSOS-SL: concurrency example

Transition rules

```
ctr P1 || P2 ={(offer' : nilo), IIS}=>
     P'1 || update-recv (P'2, v)
if P1 ={(offer' : O1), IIS}=> P'1 /\
   P2 ={(offer' : O2), IIS}=> P'2 /\
   o1 := get-offer (O1) /\
   o2 := get-offer (O2) /\
   agree (o1, o2) /\
   v := agree-value (o1, o2) .

ctr cml P ={(offer' : nilo), IS}=> cml P'
if P ={(offer' : O), IS}=> P' /\ O == nilo .
```

MSOS-SL: concurrency example

Formal verification. This search must find two final states (concurrent access to a memory location).

```
(search exec (let val x "=" (ref $(1))
                in spawn (fn y "=>" (x ":=" $(2))) ;
                   spawn (fn y "=>" (x ":=" $(3)))
                end) =>! C:Conf .)
```

MSOS-SL: concurrency example

```
Solution 1
C:Conf <- < cml(proc(pide(0),pide(2)) ||
                proc(pide(1),empty-tuple)||
                proc(pide(2),empty-tuple)),
   {...(st : <[[loc(1),$(3)]]>)} >

Solution 2
C:Conf <- < cml(proc(pide(0),pide(2))||
                proc(pide(1),empty-tuple)||
                proc(pide(2),empty-tuple)),
   {...(st : <[[loc(1),$(2)]]>)} >

No more solutions.
```

MSOS-SL: concurrency example

Formal verification. Concurrent sending / receiving.

```
(search exec (let val c "=" channel !()
              in  (spawn (fn x "=>" send !(c, $(10))) ;
                   spawn (fn x "=>" send !(c, $(11))) ;
                   recv c)
              end) =>! C:Conf .)
```

MSOS-SL: concurrency example

Again, two final outcomes possible.

```
Solution 1
C:Conf <- < cml(
  proc(pide(0),$(10)) ||
  proc(pide(1),empty-tuple) ||
  proc(pide(2), let ... in send tuple(chn(1),$(11) end),
  {...} >

Solution 2
C:Conf <- < cml(
  proc(pide(0),$(11)) ||
  proc(pide(1),let ... in send tuple(chn(1),$(10)) end)||
  proc(pide(2), empty-tuple)),{...} >
```

Implementing Maude MSOS Tool

Braga and Meseguer created Modular Rewriting Semantics (MRS), a novel method for the modular specification of programming language semantics and defined (and proved correct) a mapping from MSOS to MRS. The work is based on the joint work of Braga, Hæusler, Meseguer, and Mosses.

The Maude MSOS Tool was implemented based on this mapping and also by extending Full Maude, a Maude application that makes heavy use of Maude's reflective capabilities to create executable environments for languages, logics, etc.

Developments and future work

- MSOS-SL and MSDF
- Continuations
- Incremental MSOS specification