Maude MSOS Tool

Fabricio Chalub and Christiano Braga

fchalub@ic.uff.br / cbraga@ic.uff.br

Universidade Federal Fluminense

Acknowledgements

- CNPq and EPGE-FGV for support;

Outline

- Quick introduction to SOS and MSOS
- Overview of MSOS-SL and Maude MSOS Tool
- Rewriting logic
- MSOS-SL
- Example of simulation with MMT
- Behind the scenes
- Evolution of the language
- Developments and future work
- Conclusion

Modularity in SOS (i)

- SOS (Plotkin, 1981) is a simple yet mathematically rigorous generic semantic framework.
- 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 computations and configurations are only value-added abstract syntax trees.

$$\frac{e_0 \xrightarrow{X} e'_0}{e_0 \bullet e_1 \xrightarrow{X} e'_0 \bullet e_1}$$

- Components and are accessed through indices

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

Modularity in operational semantics (iii)

More about labels and configurations in MSOS.

- Indexed components in labels are of three different types (more can be defined, actually)
  - read only (e.g. environments of bindings)
  - read write (e.g. stores)
  - write only (e.g. output)
- Transitions are composable when their labels are composable (which is defined based on the information in the label)
- Unobservable transitions are the transitions where read-write components don't change, and write-only components emit no new information.

Maude MSOS Tool and MSOS-SL

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

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

The Maude MSOS Tool provides an executable environment for MSOS specifications where, by giving the semantics of a language $\mathcal{L}$ in MSOS, we get the ability to execute programs, and to perform formal analysis of programs in $\mathcal{L}$.

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 specification 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.
- transition rules: where the dynamic semantics of the language is specified.

MSOS-SL modules

MSOS-SL modules are written as:

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

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

```
(msos A is
 including B .
 including C .
 [...]
sosm)
```

MSOS-SL: syntax definition

The (abstract/concrete) syntax definition in MSOS-SL comes directly from Maude. Constructions: sort, subsort, op (for the declaration of operators).

Maude has a number of builtin datatypes available for use, such as the naturals, rationals, floating-point numbers, strings, etc.

MSOS-SL: syntax definition

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

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

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

```
sorts Dec Exp .
```

The `let` expression is declared in mixfix form:

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

ctor means that this operation is a constructor of terms.

MSOS-SL: syntax definition

Identifiers are terms of the sort `Id`.

```
sort Id .
```

Declarations include bindings from identifiers to values (`ValueBind`s) obtained from the evaluation of expressions.

```
sort ValueBind .

op val_ : ValueBind -> Dec [ctor] .

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

MSOS-SL: syntax definition

Value is the sort of the values expressible in our language.
Expressions evaluate to values, so we subsort Value to Exp.
Identifiers can appear in expressions also.

```
sort Value .
subsort Value < Exp .
subsort Id < Exp .
```

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

```
subsort Nat < Value .
```

We may now write:

```
op x : -> Id .
let val x = 10 in x end .
let_in_end(val_(_=_(x, 10)), x)
```

MSOS-SL: syntax definition

Operators may have associativity (`assoc`), commutativity (`comm`) and identity (`id`) attributes.

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

Other possibilities: frozen arguments, gather patterns, 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 (e, 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 a monoid: identity element $(\mathbf{e})$ and binary operation $(\mathbf{bop})$.

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

MSOS-SL: label components

Label 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 _|->_ : Id BVal -> Env [ctor] .
op __    : Env Env -> [Env] .   --- disjoint union
op find  : Env Id  -> [BVal] .
op _/_   : Env Env -> Env .     --- overriding
```

Writing [S] as the image sort of an operator makes this a partial function.

MSOS-SL: transitions

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

γ: the value-added syntax tree. α: the label expression. $\langle condition \rangle$: consists of a conjunction of transitions, written in the general form γ `=`α`=>` γ', together with membership assertions and equational conditions, separated by '`/\`'.

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

Unobservable transitions: γ `==>` γ'.

MSOS-SL: label expressions

Labels (sort `Label`) are formed by a set of *fields* of the form:

`(`i` : `C`).`

The sort `Fields` 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'), FS}`,

the variable `FS`, of sort `Fields`, matches against any unspecified set of fields.

Unobservable labels are identity labels of the sort ILabel, a subsort of `Label`, and their subsets are of the sort IFields, a subsort of `Fields`.

MSOS-SL: transitions

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

```
var  X      : Label .             var  FS           : Fields .
var  v      : Value .             vars D D'         : Dec .
vars E E'   : Exp .               vars b rho rho'   : Env .

ctr let D in E end = X => let D' in E end
if D = X => D' .
```

MSOS-SL: transitions

As an example, let us give the semantics of the `let` expression defined earlier (cont.):

```
var  X      : Label .            var  FS           : Fields .
var  v      : Value .            vars D D'         : Dec .
vars E E'   : Exp .              vars b rho rho' : Env .
```

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

MSOS-SL: transitions

As an example, let us give the semantics of the `let` expression defined earlier (cont.):

```
var  X      : Label .            var  FS           : Fields .
var  v      : Value .            vars D D'         : Dec .
vars E E'   : Exp .              vars b rho rho'   : Env .
```

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

MSOS-SL: concurrency example

Simulation. 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

Simulation. 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, Haeusler, 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.

Evolution of the language

The development of MSOS-SL coincided with the development of Mosses' own MSDF and tool in Prolog. MSDF was influenced by ASDF, created on colaboration with Jørgen Iversen.

Our visit to Aarhus focused on the usability of the Maude MSOS Tool and its MSOS-SL language, based on Mosses' MSDF experience.

The idea is to bring MSOS-SL closer to the domain of MSDF/MSOS specifications than the domain of Maude/algebraic specifications.

Our aim is to use the same language on both the Prolog and Maude tools.

BNF syntax

Instead of something like:

```
op local : Dec Exp -> Exp .
```

we should use, as in MSDF:

```
Exp ::= local(Dec, Exp) .
```

More flexible productions are possible:

```
Exp ::= if Exp then Exp else Exp .
```

Implicit importation of modules

```
Exp ::= local(Dec, Exp) .
```

From that production, we can also assume that the user needs to access the modules that declare the sets `Dec` and `Exp`.

Automatic metavariables and derived types

From the creation of a set, say `Exp`, we would have any metavariable implicitly declared that begins with the name of the set. Example: `Exp, Exp', Exp1, Exp2 ....`

This prevents the re-declaration of the same metavariables on every module that is needed and also make sure that we are consistent on the use of metavariable names.

Also, we should get derived types: `Exp+`, `Exp*`, etc.

Complete example

```
msos EXP/LOCAL is

 Exp ::= local Dec Exp .

 Label = {env : Env, ...} .

 ...
```

Complete example

```
 ...

                    Dec -{...}-> Dec'
-- ---------------------------------------------
(local Dec Exp):Exp -{...}-> local Dec' Exp .


  Env' := Env / Env0,   Exp -{env = Env', ...}-> Exp'
-- ------------------------------------------------------
(local Env Exp):Exp -{env = Env0, ...}-> local Env Exp' .


(local Env Value):Exp --> Value .
sosm
```

Developments and future work

- We are in the process of implementing the new language.
- Huge state space problem, due to small-step semantics.
- Verification problem. In rewriting logic, transitions on the conditions are "scratch pad" transitions.
- We will investigate if reduction semantics and evaluation contexts can offer in this respect. Also related is the work in the conversion of conditional to unconditional rewrite rules.

Conclusion

- Now, specifications can be written in a language closer to MSOS than Maude.
- This ease of use is combined with a high-performance engine (soon a compiler) gives us a efficient executable environment for languages defined with MSOS.
- Several formal tools available with Maude via the Maude MSOS Tool

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] .
```

comm and assoc create a multiset of processes.

Label declaration

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

MSOS-SL: concurrency example

Creation of processes.

```
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 !()))) .

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

MSOS-SL: concurrency example

Interleaving of processes.

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

MSOS-SL: concurrency example

Creation of channels.

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

MSOS-SL: concurrency example

Sending/receiving information.

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

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

ctr recv ch ={(offer' : O), IIS}=> recv-ph (ch)
if O := new-offer (rcv (ch)) .
```

MSOS-SL: concurrency example

Sending/receiving information

```
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) .
```

MSOS-SL: concurrency example

Filtering unmatched offers.

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

From concrete to abstract syntax

Concrete syntax: `if then else`, `let in end`, "application of expressions"

Abstract syntax: `cond()`, `local()`, `app()`

```
...
eq convert (if E1 then E2 else E3)
= cond (convert (E1), convert (E2), convert (E3)) .

eq convert (let D in E end)
= local (convert (D), convert (E)) .

eq convert (E1 E2)
= app (convert (E1), convert (E2)) .
...
```