A Modular Rewriting Semantics for CML

Fabricio Chalub Barbosa do Rosário
frosario@ic.uff.br

19 de março de 2004

Outline

A closer look at MSOS

Mapping MSOS to MRS

Executing and model checking CML programs

Future Directions

A closer look at MSOS

- Generalized Transition System
- Transition system with configurations: $\Gamma \times A \times \Gamma$
- $\Gamma =$ **value added** syntax tree (syntax + values)
- Configurations $\langle \gamma, \alpha, \gamma' \rangle$ are written $\gamma -\alpha\rightarrow \gamma'$
- (Conditional) rules:

$$\frac{c_1, \ldots, c_n}{c}$$

A closer look at MSOS

- The components model the **information flow** of the computation, affected by the language phrases.

- Components may be: **read-only**, **read-write**, **write-only**

- **Label composition** defines the computation path.

A closer look at MSOS

Label notation

- $\langle \gamma, \alpha, \gamma' \rangle$ / $\gamma -\alpha\rightarrow \gamma'$

- $\{\rho = \rho_0, \ldots\}$ specifies labels α such that $\alpha.\rho = \rho_0$

- $\{\sigma = \sigma_0, \sigma' = \sigma_1, \ldots\}$ specifies labels α such that $\alpha.\sigma = \sigma_0$ and $\alpha.\sigma' = \sigma_1$

- The "..." ensures that unmentioned components of labels are never excluded.

A closer look at MSOS

Read-only component: the environment (ρ)

```
let val x = 1 in x + 1 end
```

$$\frac{\mathrm{D} \multimap\!\!\!\!\rightarrow \mathrm{D}'}{\texttt{let}\ \mathrm{D}\ \texttt{in}\ \mathrm{E}\ \texttt{end} \multimap\!\!\!\!\rightarrow \texttt{let}\ \mathrm{D}'\ \texttt{in}\ \mathrm{E}\ \texttt{end}}$$

$$\frac{\mathrm{E} \multimap\!\!\!\!\rightarrow \{\rho = \rho_1[\rho_0], \ldots\} \rightarrowtail \mathrm{E}'}{\texttt{let}\ \rho_0\ \texttt{in}\ \mathrm{E}\ \texttt{end} \multimap\!\!\!\!\rightarrow \{\rho = \rho_1, \ldots\} \rightarrowtail \texttt{let}\ \rho_0\ \texttt{in}\ \mathrm{E}'\ \texttt{end}}$$

$$\texttt{let}\ \rho_0\ \texttt{in}\ \nu\ \texttt{end} \longrightarrow \nu$$

A closer look at MSOS

Read-write component: the store (σ)

`x := 2`

$$\frac{E_1 \text{ –}X\text{→ } E_1'}{E_1 \text{ := } E_2 \text{ –}X\text{→ } E_1' \text{ := } E_2}$$

$$\frac{E_2 \text{ –}X\text{→ } E_2'}{l \text{ := } E_2 \text{ –}X\text{→ } l \text{ := } E_2'}$$

$l := v \text{ –}\{\sigma = \sigma_0, \sigma' = \sigma_0[l \mapsto v], \ldots\}\rightarrow$ **nil**

Mapping Modular SOS into MRS

- MSOS rules must be in normal form:

1) fields affected by the rule are explicit in the label

2) side conditions do not involve record fields

Bad: $$\frac{R = \{env : \rho, PR\} \qquad v = \rho(x)}{x -R\rightarrow v}$$

Good: $$\frac{v = \rho(x)}{x -\{env : \rho, PR\}\rightarrow v}$$

Mapping Modular SOS into MRS

$\gamma -\alpha\rightarrow \gamma' \mapsto$ **rl** $\langle P_0, R_0\rangle \rightarrow \langle P_1, R_1\rangle$

Read-only components (c)

```
crl < p, {(c : v), pr} > => < p', {(c : v), pr'} >
      if < p, {(c : v), pr} > => < p', {(c : v), pr'} >
```

Read-write components (c, c')

```
crl < p, {(c : v), pr} > => < p', {(c : v'), pr'} >
      if < p, {(c : v), pr} > => < p', {(c : v'), pr'} >
```

Write-only components (c')

```
crl < p, {(c : l), pr} > => < p', {(c : l.v), pr'} >
      if < p, {(c : []), pr} > => < p', {(c : v), pr'} >
```

MRS of SML: overall structure

```
EXPR-SYNTAX .
EXPR-SEMANTICS is including EXPR-SYNTAX .
DECLARATIONS-SYNTAX is extending EXPR-SYNTAX.
DECLARATIONS-SEMANTICS is including DECLARATIONS-SYNTAX
                          extending EXPR-SEMANTICS .
ABSTRACTIONS-SYNTAX
ABSTRACTIONS-SEMANTICS
IMPERATIVES-SYNTAX
IMPERATIVES-SEMANTICS
EXCEPTIONS-SYNTAX
EXCEPTIONS-SEMANTICS
CONCURRENCY-SYNTAX
CONCURRENCY-SEMANTICS
```

MRS of SML: syntax declaration

- From EXPR-SYNTAX

```
sorts Ide Value Exp .    subsorts Ide Value < Exp .
op if_then_else_ : Exp Exp Exp -> Exp [prec 60] .
```

- From DECLARATIONS-SYNTAX

```
sorts Decl ValueBind .   subsort ValueBind < Decl .

op _=_ : Ide Exp -> ValueBind .
op let_in_end : Decl Exp -> Exp [prec 70] .

subsorts Exp Decl < Program .
```

MRS of SML: expressions semantics

```
vars e1 e'1 e2 e3 : Exp .        var r r ' : Record .

 crl { if e1 then e2 else e3, r } =>
     [ if e'1 then e2 else e3, r' ]
  if { e1, r } => [ e'1, r' ] .

  rl { if constr(true) then e2 else e3, r } =>
     [ e2, r ] .
  rl { if constr(false) then e2 else e3, r } =>
     [ e3, r ] .
```

MRS of SML: declarations

```
var b : Env .      vars e e'   : Exp .          var d : Decl .
var r : Record .   vars pr pr' : PreRecord .

 crl { let d in e end, r } => [ let d' in e end, r' ]
   if { d, r } => [ d', r' ] .

 crl { let b in e end, {(env : rho), pr} } =>
     [ let b in e' end, {(env : rho), pr'} ]
   if rho' := override-env (rho, b)  /\
     { e, {(env : rho'), pr} } =>
     [ e', {(env : rho'), pr'} ] .

 rl { let b in v end, r } => [ v, r ] .
```

MRS of CML: imperatives

```
crl { ref v, {(st : s), pr} } => [ l, {(st : s'), pr} ]
    if l := newLoc(s) /\ s' := update (s, l, v) .

crl { e1 := e2, r } => [ e'1 := e2, r' ]
    if { e1, r } => [ e'1, r' ] .
crl { l := e2, r } => [ l := e'2, r' ]
    if { e2, r } => [ e'2, r' ] .
crl { l := v, {(st : sigma), pr}  } =>
    [ tuple(), {(st : sigma'), pr} ]
    if sigma' := update (sigma, l, v) .
```

MRS of CML: concurrency

```
sorts Procs Pid .

op _||_ : Procs Procs -> Procs [assoc comm] .
op prc : Pid Exp -> Procs .

crl { p1 || p2, r } => [ p'1 || p2, r' ]
  if { p1, r } => [ p'1, r' ] .

crl { prc(pi1, e1:Exp), {(ac : a), pr} } =>
    [ prc(pi1, e'1:Exp), {(ac : a), pr' } ]
    if { e1,  {(ac : a), pr} } =>
       [ e'1, {(ac : a), pr' } ] .
```

MRS of CML: concurrency

```
crl { spawn v, {(pids : ps), (ac : a), pr} } =>
    [ pi, {(pids : add-pid(ps,pi)),
          (ac : concat(a, act(prc(pi, (v tuple())))), pr} ]
  if pi := new-pid(ps) .

crl { prc(pi1, e1:Exp), {(ac : a), pr} } =>
    [ (prc(pi1, e'1:Exp) || p:Procs), {(ac : a), pr'} ]
  if { e1,  {(ac : a), pr} } =>
     [ e'1, {(ac : a'), pr' } ] /\ act(p:Procs) := last(a') .
```

MRS: concrete components

```
 op find : Env Ide -> [BVal] .
---
 sort Bind CEnv .
 subsort Bind < CEnv .
 subsorts Loc Tuple List Rec NumberConstant < BVal .

 op [_,_] : Ide BVal -> Bind [ctor] .
 op __ : CEnv CEnv -> CEnv [ctor assoc comm id: mt-env] .
 op <_> : [CEnv] -> [Env] .

 eq find(< [i, bv] ce >,i) = bv .
 eq find(e,i) = no-value [owise] .
```

Executing

```
rewrite
< let fun f(x) = if x '< rat(2)
                 then x else x '* f (x '- rat(1))
  in f rat(30) end,
 { (env : < mt-env >), (st : < mt-st >) } > .

---
rewrites: 100684 in 1771ms cpu (1811ms real)
                     (56828 rewrites/second)
result Conf: < rat(2...0),
               {(env : < mt-env >),st : < mt-st >} >
```

Model Checker

Linear Temporal Logic: basic formulae

True: $\top$

Atomic propositions

Until: $\varphi \, \mathcal{U} \, \psi$

Boolean connectives: $\neg, \vee, \wedge, \ldots$

Linear Temporal Logic: other formulae

Eventually: $\Diamond \varphi = \top \, \mathcal{U} \, \varphi$

Henceforth: $\Box \varphi = \neg \Diamond \neg \varphi$

Model Checker: Dekker's Algorithm

Dekker's algorithm: a solution to the mutual exclusion problem

Two processes, P_1 and P_2 try to access a shared resource using Dekker's solution.

If the solution is correct, a race condition should never occur (safety).

Better yet: when one or more processes have expressed their intentions to enter CS, one of them eventually enters (liveness).

Desirable: any process that expresses its intention to enter CS will be able to do so in finite time (freedom from starvation).

We'll test the **safety** property.

Dekker algorithm for process i

```
i = {1,2}      other = 3 - i      turn = 1

status[i] := competing;
while (status[other] = competing) {
      if (turn = other) {
            status[i] := out;
            wait until (turn = i)
            status[i] := competing;
      }
}
CS;
turn := other;
status[i] := out;
```

SML code for process 1

```
cz1, cz2, c1, c2, turn | competing = 1  out = 0 nothing = ()

while true do
   c1 := competing ;
   while (!c2) = competing do
       if (!turn) = 2 then
           c1 := out
           while (!turn) = 2 do nothing
           c1 := competing)
       else nothing
   cz1 := 1 ; cz1 := 0 ;
   turn := 2 ;
   c1 := out
```

Dekker algorithm: main process

```
op dekker : -> Conf .
op init : -> Record .
eq init = { (pids : < p[pid(1)] >), (ac : < mt-ac >),
            (env : < mt-env >), (st : < mt-st >) } .
eq dekker = < prc(pid(1),
  let val pat(cz1) = ref 0 and pat(cz2) = ref 0 and
          pat(ir1) = ref 0 and pat(ir2) = ref 0 and
          pat(c1) = ref 0 and  pat(c2) = ref 0 and
          pat(turn) = ref 1
  in let val pat(p1) = (fn p() => <...>) seq
             pat(p2) = (fn p() => <...>)
     in (spawn p1 ; spawn p2) end
  end), init > .
```

Model Checker: Dekker's Algorithm

LTL specification of Dekker's Solution

Proposition: $enterCrit_i$ is **true** only when process P_i is in its critical zone.

Race condition: $enterCrit_1 \wedge enterCrit_2$

Race condition will never occur: $\Box\neg(enterCrit_1 \wedge enterCrit_2)$

LTL formulae range over a set of **states** that will be specified soon.

Model Checker: Dekker's Algorithm

```
fmod LTL is
  sorts Prop Formula .
  subsort Prop < Formula .
  op _|=_ : State Formula ~> Bool .
endfm

subsort Conf < State .
op enterCrit : Int -> Prop .
eq < P:Program,
     {(st : < [[loc(1),1]] C:CStore >), PR:PreRecord} >
     |= enterCrit(1) = true .
```

Model Checker: Dekker's Algorithm

```
op modelCheck : State Formula ~> ModelCheckResult
                                          [special (...)] .


red modelCheck(dekker, []~ (enterCrit(1) /\ enterCrit(2))) .
rewrites: 169732634 in 21745617ms cpu (25384348ms real)
                                  (7805 rewrites/second)
result Bool: (true).Bool
```

Running time: 21745617ms (6 hours!)

Developments and future work

Use parser-generators to parse SML programs

Develop a Big-step / equational semantics

Automatic MSOS $\mapsto$ MRS

New Small-step / rewriting semantics ("true-concurrency")

Reduction semantics

Enhancing model-checking capabilities

Definitive Semantics

Using parser-generators

Problem: odd SML syntax in current spec: `x '< rat(2)`

Solution: use parser-generators to convert from SML to a syntax tree.

```
let val x = 1 in x end
```

vs.

```
let-in-end (val (valbind (pat(x), scon(1))), x, end)
```

Working `yacc` (Bison, actually) prototype.

Big-step

Advantage of using big-step: huge increase in efficiency (1000! in aprox. one second.).

How to do exception handling and concurrent processing with it.