A modular rewriting semantics for CML

Fabricio Chalub and Christiano Braga

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

Universidade Federal Fluminense

Acknowledgements

- Peter Mosses for comments on our specification;
- CNPq and EPGE-FGV for partial support;

Outline

- Formal semantics of programming languages
- Modularity in specifications
- Rewriting Logic
- Concurrent ML
- Verification of Concurrent ML programs
- Developments and future work

Formal Semantics

A formal semantics for some programming language $\mathcal{L}$ provides:

- An unambiguous definition of what $\mathcal{L}$ means;
- The ability to formally reason about $\mathcal{L}$ and prove desired properties;
- If the specification is executable, the formal reasoning can be computer aided;

Modularity in the context of formal semantics

The specification process (that is, writing down the formal semantics) is inherently creative and can be extremely complex. Modularity comes into play:

- Software engineering: a methodology to build complex systems (specifications in this case).
- Ease of extension: new functionality is “easily” added (no need to change previous modules). Related to software engineering.
- Didactic way of formally present something (programming languages semantics, in this case).

Rewriting Logic (RWL)

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

Modularity in Rewriting Logic

- Modular Rewriting Semantics (MRS): Braga and Meseguer defined a technique that brings modularity into rewriting logic programming languages semantics;
- Is the continuation of the joint work of Braga, Meseguer, Mosses, and Hermann. It is influenced by Peter Mosses' *Modular Structural Operational Semantics* (MSOS) and shares with MSOS the technique of *record inheritance* (to be discussed later);
- There is a bissimulation between MSOS and MRS;

$$\text{MRS Configuration} = \langle\ \text{Program}, \text{Semantic record}\ \rangle$$

Modularity in Rewriting Logic

- Record inheritance. "The less we specify, the more general the record is.";
- Use of the variable that captures the "rest of the record" in the context of rewriting modulo ACI;

```
R:Record
{ (env : e:Env), PR:PreRecord }
```

- Abstract functions over components in rules. Expose only the interface and hide the (concrete) implementation. Abstract functions aren't tied to a particular implementation of a component (e.g., a store).

```
{ (env:[x,loc(1)]), (store:[[loc(1),1]]) }
```

Concurrent ML

We specified a modular rewriting logic semantics of (a significant subset of) CML and proved some properties of CML programs. Reasons for using CML:

- Formal from the beginning. Milner, et al. gave an operational semantics for Standard ML.
- Reppy formally defined Concurrent ML, also in operational semantics style.
- Mosses gave a *modular structural operational semantics (MSOS)* for CML.
- Several implementations (SML/NJ, Moscow ML, Poly/ML, ML Kit) and applications (Isabelle, HOL, older JAPE versions).

MRS of SML: declarations

```
fmod DECLARATIONS-SYNTAX is
  extending EXPRESSIONS-SYNTAX .

 sorts Decl ValueBind .
 subsort ValueBind < Decl .

 op _=_ : Ide Exp -> ValueBind .
 op let_in_end : Decl Exp -> Exp .
endfm
```

Example:

```
let val x = 1 in e(x) end
```

MRS of SML: declarations

```
let val x = 1 in e(x) end
```

- Semantics of let-in-end

```
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: concurrency

- Semantics of concurrency (overview)

```
sorts   Proc Procs Pid .
subsort Proc < Procs .

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

crl { p1 || PS2, r } => [ PS1 || PS2, r' ]
 if { p1, r } => [ PS1, r' ] .
```

- Matching modulo AC guarantees the nondeterministic choice of which process to step at a given time, giving an interleaving model of concurrency.

Formal Verification

- Two processes, P_1 and P_2 try to access a shared resource using some sort of mutual exclusion algorithm.
- One of the properties of the solution should be safety, that is, no race condition should occur.
- Other is freedom from starvation, that is, if one P_i is competing for the shared resource, it will eventually get access it.
- We'll test both safety and a freedom from starvation properties of Dekker's solution.

Model Checker: Dekker's Algorithm

Proving the freedom of starvation of Dekker's solution.

Maude's model checker will return a counterexample for "It is always true that when both P_1 and P_2 are competing, the turn will always be with P_1, that is, memory location l_7 will always be 1."

$$\Box(competing \rightarrow (\Box turn(1)))$$

The counterexample:

```
{< ...,{(env : < mt-env >),(st : < [[loc(1),rat(0)]]
[[loc(2),rat(0)]] [[loc(3),rat(0)]] [[loc(4),rat(0)]]
[[loc(5),rat(1)]] [[loc(6),rat(1)]] [[loc(7),rat(2)]] >),
(val : < mt-val >),(pids : < pval[pid(1)] x pval[pid(2)] x
pval[pid(3)] >),(ac : < mt-ac >),tr : < mt-tr>} >,'step}
```

Model Checker: Dekker's Algorithm

How to prove the safety of Dekker's solution

- On our CML implementation, the critical section of process P_i consists of two instructions: $l_i \leftarrow 1; l_i \leftarrow 0$, where l_i is a memory location bound to a variable on process P_i.
- Let c_i be the proposition that is true iff $l_i = 1$. Notice that c_i will only be true when P_i is inside its critical section.
- The LTL formula for "race condition will never occur" is then $\Box\neg(c_1 \wedge c_2)$

Model Checker: Dekker's Algorithm

```
mod CHECK is including CONCURRENCY-TEST .
 including MODEL-CHECKER .
 subsort Conf < State .

 op mutex-violation : -> Prop .
 eq < P:Program,{(st : <[[loc(1),rat(1)]]
                        [[loc(2),rat(1)]] C:CStore>),
      PR:PreRecord } > |= mutex-violation = true .
endm
reduce modelCheck(dekker, []~ mutex-violation) .
rewrites: 58380093 in 2315950ms cpu (2362140ms real)
          (25207 rewrites/second)
result Bool: ({true).Bool
```

Developments and future work

- Although the mapping was applied manually, we are working on an automatic translator;
- New specification with the following characteristics:
  - True concurrency;
  - Reduction semantics + CPS
  - Mosses' Definitive Semantics (basic library of semantic constructors that can be reused);
  - The use of parser-generators to translate SML programs into Definitive Semantics constructions;