Modular Rewriting Semantics in Practice

Christiano Braga

cbraga@ic.uff.br

Universidade Federal Fluminense

José Meseguer

meseguer@cs.uiuc.edu

University of Illinois at Urbana-Champaign

Context

- Rewriting logic (RWL) is a widely used semantic framework with many formal analysis tools developed based on the rewriting semantics of programming languages.
- Modularity is an important property of a language specification $\mathcal{S}$, meaning that $\mathcal{S}$ does not have to be redefined when the language is extended.

Context

- Modular structural operational semantics (MSOS) is an extension to structural operational semantics (SOS), proposed by Peter Mosses, which solves the modularity problem of SOS specifications.
- **Modular rewriting semantics** (MRS) is the result from the application and *extension* of lessons learned from previous work, on the mapping from MSOS to RWL, to the development of rewriting semantics of programming languages.

Objectives of this Talk

1. To present three techniques to help the development of modular specifications of programming languages' semantics in rewriting logic. (Or the *modular rewriting semantics* of programming languages.)
2. To illustrate the practical usefulness of our approach by means of examples taken from the MRS of Milner's CCS, one of the case studies presented in the paper.

Strategy for this Talk

1. Modularity requirements
2. Modular rewriting semantics specifications
   (a) Configurations
   (b) Techniques
   (c) Small-step specifications
3. CCS
4. Weak transition semantics of CCS
5. (Comment on) GNU bc
6. Final remarks

Modularity Requirements

MRS is defined in the context of *incremental specifications*: Syntax and corresponding semantic axioms are introduced for groups of related features.

- Incremental presentation of the syntax of a language $\mathcal{L}$: $\{\mathcal{L}_i\}_{i \in I}$, with I a *poset* with a top element $\top$, such that (i) if $i < j$ then $\mathcal{L}_i \subseteq \mathcal{L}_j$, and (ii) $\mathcal{L}_\top = \mathcal{L}$.
- Incremental rewriting semantics for $\mathcal{L}$ is an indexed family of rewrite theories $\{\mathcal{R}_{\mathcal{L}_i}\}_{i \in I}$ with $\mathcal{R}_{\mathcal{L}_i}$ defining the semantics of $\mathcal{L}_i$.

Modularity Requirements

- *Monotonicity*: if $i \leq j$ then, there is a theory inclusion $\mathcal{R}_{\mathcal{L}_i} \subseteq \mathcal{R}_{\mathcal{L}_j}$.
- *Extensibility*: rewrite rules should be defined in the most abstract and general way possible, that is, the semantics of each language feature is defined once and for all.

MRS Specifications/Configurations

A *configuration* is a state of a particular program execution. Configurations in MRS specifications are organized as pairs of program syntax and a record of semantic entities, such as the memory store or the declarations environment.

MRS Specifications/Techniques

- The first technique to achieve modularity, shared with MSOS, is *record inheritance*: it means that one can always consider a record with more fields as a special case of one with fewer fields.
- Features added later to a language may necessitate adding new semantic components to the record; but the axioms of older features can be given once and for all in full generality: they will apply just the same with new components to the record.

MRS Specifications/Techniques

```
fmod RECORD is
...
op _,_ : PreRecord PreRecord -> PreRecord
                    [ctor assoc comm id: null] .
op _:_ : [Index] [Component] -> [Field] [ctor] .
op {_} : [PreRecord] -> [Record] [ctor] .
op duplicated : [PreRecord] -> [Truth] .
...
eq duplicated((I : C),(I : C′), PR) = tt .
cmb {PR} : Record if duplicated(PR) =/= tt .
endfm
```

MRS Specifications/Techniques

- Record inheritance is accomplished through pattern matching *modulo* associativity, commutativity, and identity.
- For example, a record with an environment component indexed by `env` and a store component indexed by `st` can be viewed as a special case of a record with just the environment component. A function `get-env` extracting the environment could be defined by `eq get-env({env : E, PR}) = E .`

MRS Specifications/Techniques

- The second technique is the systematic use of *abstract datatypes* to represent syntactic and semantic entities.
- In a language specification no *concrete* syntactic or semantic sorts are ever identified with abstract sorts: they are always either specified as *subsorts* of corresponding abstract sorts, or mapped to abstract sorts by *coercion*. Axioms are given only at the level of concrete sorts.

MRS Specifications/Techniques

The third technique regards the form of the rules. The only new rewrite rules in the i^{th} rewrite theory $\mathcal{R}_{\mathcal{L}_i}$ are semantic rules of the form

$$\langle f(t_1, \ldots, t_n), u\rangle \longrightarrow \langle t', u'\rangle \ \mathit{if}\ C$$

where f is a new language feature, e.g., `if-then-else`, introduced in $\mathcal{L}_i$, u and u' are record expressions and u contains a variable PR of sort `PreRecord` which allows record inheritance.

MRS Specifications/Techniques

- The following information hiding discipline should also be followed in u, u' and in any record expression appearing in C: besides basic record syntax only abstract functions symbols are allowed.
- This allows for extensible change of internal representations of semantic entities, used, for instance, in the weak transition semantics of CCS.

MRS Specifications/Small-step

- When representing SOS specifications in RWL it is necessary to control the number of rewrites in the conditions due to the fact that rewrites are reflexive and transitive in RWL.
- Therefore, in order to specify explicitly one (or more) rewrite(s) in the conditions of conditional rules the configuration theory should be extended as follows.

MRS Specifications/Small-step

```
mod RCONF is
 inc RECORD .
 sorts Program Conf .
 op <_,_> : Program Record -> Conf [ctor] .
 op {_,_} : [Program] [Record] -> [Conf] [ctor] .
 op [_,_] : [Program] [Record] -> [Conf] [ctor] .
 vars p p' : Program .  vars r r' : Record .
 crl [step] : < p , r > => < p' , r' >
  if { p , r } => [ p' , r' ] .
endm
```

MRS Specifications/Small-step

- Any application of the `step` rule mimics a one-step rewrite, given that semantic rules have the form

$$\{t, u\} \longrightarrow [t', u']\ \mathit{if}$$
$$\{v_1, w_1\} \longrightarrow [v_1', w_1'] \wedge \ldots \wedge \{v_n, w_n\} \longrightarrow [v_n', w_n'] \wedge C.$$

- Therefore the proofs of rewrites are the finitary *computations*.

CCS

We illustrate the MRS of CCS with the semantic rules for processes composition and process definition.

CCS

```
*** Composition
crl {p | q, {(tr : t), pr}} =>
    [p' | q, {(tr : t ; a), pr}]
 if {p, {(tr : nil), pr}} =>
    [p', {(tr : nil ; a), pr}] .

crl {p | q, {(tr : t), pr}} =>
    [p' | q',{(tr : t ; tau), pr}]
 if {p,{(tr : nil), pr}} =>
    [p',{(tr : nil ; l), pr}] /\
    {q,{(tr : nil), pr}} =>
    [q',{(tr : nil ; (~ l)), pr}] .
```

CCS

```
*** Definition
crl {x,{(tr : t), (env : e), pr}} =>
    [p,{(tr : t ; a), (env : e), pr'}]
 if p := def(x, e) /\
    {p, {tr : nil), (env : e), pr}} =>
    [p', {(tr : nil ; a), (env : e), pr'}]
```

CCS

1. *Record inheritance* is used to define the two record indices: the action trace field, indexed by `tr`, and the environment of processes names, indexed by `env`.
2. Traces and the environment are defined as *abstract datatypes*, with action composition (`_;_`) and process definition (`def`) defined as abstract functions.

CCS

3. The *information hiding* discipline is applied in the conditions of the rules in order to make the specification extensible. No concrete functions are applied to traces and for processes definition.

CCS Weak Transition Semantics

- In the weak transition semantics τ steps become unobservable.
- The extension is quite simple: a *new constructor* from list of actions (`ActList`) to a trace (`Trace`) is defined where τ steps are ignored.

CCS Weak Transition Semantics

```
mod WEAK-CCS-SEMANTICS is
  extending CCS-SEMANTICS .

  op    <_> : ActList -> Trace [ctor] .
        *** concrete sort coercion

  vars al al' : ActList .
  var a : Act .

  eq < al tau al' > = < al al' > .
  eq < al > ; a = < al a > .
endm
```

GNU bc

- GNU bc is an arbitrary precision calculator imperative language, with syntax that resembles the C language.
- We have given an MRS semantics for bc, both in rule- and equational-based. In the former transitions in the conditions are represented by “matching equations”, that is, equations defined using the match ($:=$) operator.

GNU bc

- A prototype tool that checks compatibility of the physical units of variables was defined as an extension to the MRS of bc. This extension follows the idea of the weak transition semantics of CCS.

Final Remarks

- We have proposed a general method for developing modular semantic definitions of programming languages in rewriting logic.
- Two case studies were developed, for Milner's CCS and GNU bc.
- Current work is on the experimentation of these techniques on more case studies involving bigger languages; the application of formal verification techniques; and the extension to a truly concurrent semantics.