Modular Structural Operational Semantics with Maude MSOS

Eszter Tasi
ISI-student, Hagenberg

Content

- Formal semantics of programming languages
- Structured Operational Semantics
- Modular Structured Operational Semantics
- Defining a programming language with the Maude MSOS Tool

Formal Semantics

The “classical way” to define a programming language: writing a compiler which translates a high-level programming language to a lower level one.

The formal semantics is concerned with the rigorous mathematical study of the meaning of programming languages.

- Denotational Semantics
- Operational Semantics

Motivation

Why to use operational semantics for defining programming languages?

- If we want a non-imperative language, writing a compiler is difficult.
- We can specify our concepts, behaviors, systems in an abstract manner, without concerning about how it is internally realized.
  - programming languages,
  - verifications,
  - specification of concurrent systems.

Denotational vs. Operational Semantics

Denotational semantics

- each phrase in the language is *translated* into a denotation;
- compilation
- target language is a mathematical formalism
- Ex.: functional languages → domain theory

Operational Semantics

- the execution of the language is *described* directly;
- interpretation;
- target language is a mathematical formalism;
- defines an abstract machine: gives meaning to phrases by describing the transitions they induce on states of the machine

Operational Semantics

- Describes how a valid program is interpreted as sequences of computational steps.
- The sequences are the meaning of the programs.
- The result of the last step is the value of the program.
- The concept was used for the first time in defining the semantics of Algol 68.
- The use of the term with present meaning was introduced by Dana Scott.

Different Approaches for Operational Semantics

- Structured Operational Semantics (SOS):

  The behavior of a program is defined in terms of the behavior of its parts, in a syntax oriented way.

  Introduced by Gordon D. Plotkin, 1981

- Modular Structured Operational Semantics (MSOS):

  The transition rules for each construct are completely independent of the presence or absence of other constructs in the described language.

  Introduced by Peter D. Mosses,

SOS - Abstract Syntax

Symbols for syntactic sets must be defined:

- Numbers N
- Truth values: T = {true, false}
- Identifiers Id
- Arithmetic expressions Aexp
- Commands Com
- Declarations Dec

Metavariables are ranging over the given sets.

Constructor functions must be defined:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

$$d ::= \text{const } X = a \mid \text{var } X := a \mid d_0 ; d_1$$

$$c ::= \text{skip} \mid X := a \mid c_0 ; c_1 \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \mid \text{while } b \text{ do } c$$

SOS – Computed Values

The SOS of most constructs of programming languages involves computations which, on termination, result in a value of some kind.

- Expression values: $\mathbf{N} \cup \mathbf{T}$
- Command values: {nil}
- Declaration values

SOS – Auxiliary Entities

- Locations – independent memory cells
- Storable values – $\mathbf{N} \cup \mathbf{T}$
- Denotable values – $\mathbf{N} \cup \mathbf{T} \cup \mathbf{Loc}$
- Stores – Location $\rightarrow$ Storable value
- Environments (**Env**) – Id $\rightarrow$ Denotable value

Declaration values: **Env**

SOS – Configurations

- Configurations: states of transition systems
- Computation of a part of a program: sequence of transitions between configurations, starting from an initial configuration and terminating in a final configuration
- Initial configuration : syntax + auxiliary components
- Final configuration : same structure but with computed value instead of the original syntax
- Value-added syntax: the sets of configurations are generalized by adding computed values to the syntactic sets:

$$d ::= \rho,\ \rho \in Env$$

$$\Gamma = (\mathbf{Aexp} \cup \mathbf{Com} \cup \mathbf{Dec}) \times \mathbf{Env} \times \mathbf{Store}$$

$$T = (\mathbf{N} \cup \mathbf{T} \cup \{\text{nil}\} \cup \mathbf{Env}) \times \mathbf{Env} \times \mathbf{Store}$$

Labelled Terminal Transition Systems (LTTS)

Definition: $\langle \Gamma, A, \longrightarrow, T \rangle$

- Γ set of configurations γ
- A set of labels α
- $\longrightarrow \subseteq \Gamma \times A \times \Gamma$ ternary relation (notation $\gamma \xrightarrow{\alpha} \gamma'$)
- $T \subseteq \Gamma$ set of terminal configurations, such that $\gamma \xrightarrow{\alpha} \gamma'$ implies $\gamma \notin T$

A computation is a finite or infinite sequence of successive transitions, such that for the last configuration at the end of the sequence we have $\gamma_n \in T$.

SOS - Rules

SOS rules define the transitions in the LTTS.

Structure: $\frac{c_0 c_1 \ldots}{c}$, conditions/conclusion

Example: evaluation of sums

- identifiers are directly bound to constant values, hence stores omitted;
- when the environment is the same the notation $\rho \vdash$ is used;

$$\frac{\rho \vdash e_0 \longrightarrow n_0, \ \rho \vdash e_1 \longrightarrow n_1,}{\rho \vdash e_0 + e_1 \ \longrightarrow n_0 + n_1}$$

$$\frac{\rho(x) = con}{\rho \vdash x \longrightarrow con}$$

SOS – Styles

- Big step rules (usually used for expressions)
- Small step rules (usually used for commands):

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

$$\frac{\rho \vdash e_1 \longrightarrow {e_1}'}{\rho \vdash n + e_1 \longrightarrow n + {e_1}'}$$

$$\frac{n = n_0 + n_1}{\rho \vdash n_0 + n_1 \longrightarrow n}$$

Problem

- Extendibility: if we want to introduce stores the store must be included in configurations and the rules must be reformulated.
- Reusability: we need to make changes, the existing code is not reused.

Solution: MSOS

MSOS - Configurations

- Γ: restricted to value-added syntax, no auxiliary components

  $$\Gamma = \mathbf{Aexp} \cup \mathbf{Com} \cup \mathbf{Dec}$$

- T: restricted to computed values

  $$T = \mathbf{N} \cup \mathbf{T} \cup \{nil\} \cup \mathbf{Env}$$

Generalized Transition Systems (GTS)

Definition: $\langle \Gamma, \mathbb{A}, \longrightarrow, T \rangle$

$\mathbb{A}$ is a category with morphisms A such that $\langle \Gamma, A, \longrightarrow, T \rangle$ is an LTTS.

Category: an abstract way to describe mathematical entities and their relationships.

It consists of:

- a set of objects O,
- a set of morphisms (arrows) A, whose source and target are objects,
- a partial function for composing morphisms ($A \times A \rightarrow A$),
- a function giving an identity morphism for each object ($O \rightarrow A$).

A computation in the GTS is a computation in the underlying LTTS, such that the consecutive transition labels must be composable in $\mathbb{A}$.

MSOS – Configurations and Labels

Configuration

- the part of the program which remains to be executed.

Labels on transitions

- The state of processed information at the beginning (first part) and at the end of transition (second part).
- Stores the information contained by auxiliary components.

MSOS – Types of Labels

Read-only

- Holds an information , which does not change in a transition.
- Ex.: environments

Read-write

- The information can be inspected and changed.
- Declared as pairs: information *before* and *after* the transition.
- Ex.: stores

Write-only

- Can be updated in the transition but cannot be inspected by subsequent transitions.
- Refer to information *after* the transition.
- Ex.: output signals

Example of Label

The category which models **Env**:

- object set = **Env**
- morphisms = **Env**
- A single identity morphism for each object.
- Composable if they are equal.

The category which models **Store**:

- object set = **Store**
- morphisms = **Store** x **Store**
- Identity morphism's form: $\langle \sigma, \sigma \rangle$
- Composable if the target of the first morphism equals with the source of the second one.

The labels of a GTS can be composed from different types of labels.

Example:

- object set $\subseteq$ **Env** x **Store**
- morphism set $\subseteq$ **Env** x **Store** x **Store**

MSOS - Rules

$$\gamma \xrightarrow{\langle \rho, \sigma, \sigma', t \rangle} \gamma'$$

corresponds to

$$\rho \vdash \langle \gamma, \sigma \rangle \xrightarrow{t} \langle \gamma', \sigma' \rangle$$

More examples and notations later.

The Maude Framework

- Developed by Stanford Research Institute (SRI) International.
- Highly extensible.
- Based on rewriting logic.
- Supports the work with formalisms.
- Used to create executable environments for different logics, theorem provers, languages and models of computation.

The MSOS Maude Tool

- Execution environment for MSOS specifications.
- Developed by Fabricio Chalub and Christiano Braga.
- Specification of programming language semantics and concurrent systems.
- Is an implementation based on mapping from MSOS to rewriting logic.
- The Modular SOS Definition Formalism (MSDF) specification language is supported:
  - extended-BNF notation for the definition of abstract grammar,
  - a textual representation for MSOS transitions

Example – Defining MINI-LANGUAGE

Everything is encapsulated in separate modules:

- EXPRESSIONS:
  - Defines the evaluation of arithmetical expressions.
  - No auxiliary entities needed.
- IDENTIFIERS:
  - Introduces the “let-in-end“ expression.
  - The use of environments is needed.
- COMMANDS:
  - Introduces commands.
- ASSIGNMENTS:
  - Variable declarations and assignments.
  - Stores are needed.
- MINI-LANGUAGE:
  - Uses the defined modules to define a small programming language.

EXPRESSIONS

```
(msos EXPRESSIONS is
 Exp .
 Op .

 Op ::= sum
     | sub .

 Exp ::= Exp Op Exp
       | Int .
--- transition rules ---
sosm)
```

EXPRESSIONS – Rules

```
                Exp1 -{...}-> Exp'1
-- ----------------------------------------
(Exp1 Op Exp2) : Exp -{...}-> Exp'1 Op Exp2 .

                Exp2 -{...}-> Exp'2
-- --------------------------------------
(Int Op Exp2) : Exp -{...}-> Int Op Exp'2 .

     Op := sum, Int3 := Int1 + Int2
-- ----------------------------------
(Int1 Op Int2) : Exp --> Int3 .

     Op := sub, Int3 := Int1 - Int2
-- ----------------------------------
(Int1 Op Int2) : Exp --> Int3 .
```

IDENTIFIERS

```
(msos IDENTIFIERS is
 Id .
 Env = (Id, Int) Map .
 Exp ::= let Id = Int in Exp end
       | Id .
 Label = { env : Env, ... } .

    Env' := (Id |-> Int) / Env,
                       Exp -{env = Env', ...}-> Exp'
 -- ------------------------------------------------
 (let Id = Int in Exp end) : Exp -{env = Env, ...}->
   (let Id = Int in Exp' end) .

 (let Id = Int in Int' end) : Exp --> Int' .

    Int := lookup (Id, Env)
 -- -----------------------------
 Id : Exp -{env = Env, -}-> Int .
sosm)
```

MINI-LANGUAGE, ver.1

```
(msos MINI-LANGUAGE is
 see EXPRESSIONS, IDENTIFIERS .
sosm)
```

The Test module:

```
(mod TEST is
 including MINI-LANGUAGE .

 ops x y z : -> Id .
 endm)
```

Execution of a simple program:

```
 (rew < (let x = 5 in
         let y = 6 in
          let z = 8 in
             x sub (y sum z)
           end
          end
         end ) ::: 'Exp, { env = void } > .)
```

IDENTIFIERS - modified

```
(msos IDENTIFIERS is
 Id .
 Denotable .
 Env = (Id, Denotable) Map .
 Denotable ::= Int .
 Exp ::= let Id = Int in Exp end
       | Id .
 Label = { env : Env, ... } .

    Denotable := Int,  Env' := (Id |-> Denotable) / Env,
                       Exp -{env = Env', ...}-> Exp'
 -- --------------------------------------------------------
 (let Id = Int in Exp end) : Exp -{env = Env, ...}->
   (let Id = Int in Exp' end) .

 (let Id = Int in Int' end) : Exp --> Int' .

    Denotable := Int, Denotable := lookup (Id, Env)
 -- ---------------------------------------------------
                      Id : Exp -{env = Env}-> Int .
sosm)
```

COMMANDS

```
(msos COMMANDS is
 Cmd .
 Cmd ::= skip
      | Cmd ; Cmd .

      Cmd1 -{...}-> Cmd'1
 -- ----------------------------------------
 (Cmd1 ; Cmd2) : Cmd -{...}-> Cmd'1 ; Cmd2 .

      Cmd0 -{...}-> skip
 -- -----------------------------------
 (Cmd0 ; Cmd1) : Cmd -{...}-> Cmd1 .

 sosm)
```

ASSIGNMENTS

```
(msos ASSIGNMENTS is
 Loc .
 Store = (Loc, Int) Map .
 Denotable ::= Loc .
 Cmd ::= Id = Exp
     | var Loc Id = Exp .

 Label = { st : Store, st' : Store, ... } .

--- transition rules for simple assignments
--- transition rules for declaration and assignment

sosm)
```

ASSIGNMENTS –
Rules for declarations and assignments

```
              Exp0 -{...}-> Exp'0
 -- ------------------------------------------
  (var Loc Id = Exp0) : Cmd -{...}->
                           var Loc Id = Exp'0 .



       Env' := (Id |-> Loc) / Env,
              Store' := (Loc |-> Int) / Store
 -- --------------------------------------------------
  (var Loc Id = Int) : Cmd -{env = Env',
              st = Store, st' = Store', -}-> skip .
```

ASSIGNMENTS - Rules for simple assignment

```
             Exp0 -{...}-> Exp'0
-- ----------------------------------------
(Id = Exp0) : Cmd -{...}-> Id = Exp'0 .



  Loc := lookup (Id, Env),
             Store' := (Loc |-> Int) / Store
-- -------------------------------------------
 (Id = Int) : Cmd -{env = Env,
      st = Store, st' = Store', -}-> skip .
```

ASSIGNMENTS

```
      Loc := lookup (Id, Env),
                   Int := lookup (Loc, Store)
-- ----------------------------------------
 Id : Exp -{env = Env,
                   st = Store, st' = Store, -}-> Int .
```

MINI-LANGUAGE – ver.2

```
(msos MINI-LANGUAGE is
 see EXPRESSIONS, IDENTIFIERS .
 see COMMANDS, ASSIGNMENTS .
sosm)
```

The Test module:

```
(mod TEST is
 including MINI-LANGUAGE .
 ops a1 a2 a3 : -> Loc .
 ops s q r x y z : -> Id .
 endm)
```

Execution of a simple program:

```
 (rew < ( var a1 s = 7 sum 8;
          var a2 q =
              let x = 5 in
               let y = 6 in
                   x sub (y sum s)
                end
               end;
           s = s sub q
           ) ::: ‘Cmd, { env = void, st = void, st’ = void } > .)
```

Maude MSOS Tool Case Studies

- Specification of a subset of ML and a subset of Java language using Constructive MSOS (every construct of a language must be in a separate module).
- Specification of a pure functional programming language called Mini-Freja.
- Specification and verification of distributed algorithms.

Conclusions

MSOS is a powerful framework for formal definition of programming languages.

Maude MSOS is an implementation of MSOS => tool for teaching operational semantics.

Advanced users can make real use of the power of Maude MSOS:

- easily define domain specific languages by reusing existing modules;
- specifying concurrent systems.

BUT:

Not enough examples and the existing ones are not clearly explained => difficulties for beginners.

Resources

- http://en.wikipedia.org
- http://maude.cs.uiuc.edu/
- http://maude-msos-tool.sourceforge.net/
- Glynn Winskel

  The Formal Semantics of Programming Languages,

  MIT Press, 1993
- Peter D. Mosses

  Modular Structural Operational Semantics,

  BRICS Report Series, 2005
  http://www.brics.dk/RS/05/7/index.html
- Fabricio Chalub, Christiano Braga

  Maude MSOS Tool, 2005

  http://maude-msos-tool.sourceforge.net/mmt-manual.pdf

Questions?

Thank You!