Formal Component-Based Semantics

Ken Madlener,
Sjaak Smetsers,
Marko van Eekelen

Radboud University Nijmegen

August 28, 2011

Introduction

- A common problem with language formalizations is the lack of reusability.
  - E.g.: command sequencing is a part of virtually every imperative language.
- Solution: Component-Based Semantics, proposed by Peter D. Mosses in 2009.
  - Think Intentional Programming (Microsoft)
  - Combines *basic* language constructs contained in an *open-ended* repository to develop languages.
  - Components can be defined using Action Semantics or Modular SOS.
- This talk presents a first formalization, developed in Coq.

A While-Loop with Break

$$\mathit{Cmd}[\![\text{while } (E)\ C]\!] = \text{catch}(\text{cond-loop}(\mathit{Exp}[\![E]\!],\ \mathit{Cmd}[\![C]\!]),\ \text{eq}(\text{"breaking"},\ \text{skip}))$$

$$\mathit{Cmd}[\![\text{break}]\!] = \text{throw}(\text{"breaking"})$$

A While-Loop with Break and Continue

$$
\begin{aligned}
\mathit{Cmd}[\![\text{while } (E)\ C]\!] \;=\; & \text{catch}(\text{cond-loop}(\mathit{Exp}[\![E]\!],\\
& \qquad\qquad \text{catch}(\mathit{Cmd}[\![C]\!],\\
& \qquad\qquad\qquad \text{eq}(\text{"continuing"},\ \text{skip})),\\
& \qquad \text{eq}(\text{"breaking"},\ \text{skip}))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Cmd}[\![\text{break}]\!] &= \text{throw}(\text{"breaking"})\\
\mathit{Cmd}[\![\text{continue}]\!] &= \text{throw}(\text{"continuing"})
\end{aligned}
$$

Exceptions in Modular SOS

$$\textbf{Cmd} ::= \mathsf{skip} \mid \mathsf{throw}(\textbf{String})$$

$$\mathsf{Label} := \{\epsilon : \textbf{String}, \ldots\}$$

$$\mathsf{throw}(E) \xrightarrow{\{\epsilon'=E,-\}} \mathsf{skip}$$

$$\textbf{Cmd} ::= \mathsf{eq}(\textbf{String}, \textbf{Cmd})$$

$$\mathsf{Label} := \{\epsilon : \textbf{String}, \ldots\}$$

$$\frac{\epsilon = E}{\mathsf{eq}\,(E, C) \xrightarrow{\{\epsilon,-\}} C}$$

$$\textbf{Cmd} ::= \mathsf{catch}\,(\textbf{Cmd},\ \textbf{Cmd})$$

$$\mathsf{Label} := \{\epsilon : \textbf{String}, \ldots\}$$

$$\frac{C_1 \xrightarrow{\{\epsilon,X\}} C_1' \qquad \epsilon = ()}{\mathsf{catch}\,(C_1,\ C_2) \xrightarrow{\{\epsilon,X\}} \mathsf{catch}\,(C_1',\ C_2)} \qquad \frac{C_1 \xrightarrow{\{\epsilon,X\}} C_1' \qquad \epsilon \neq ()}{\mathsf{catch}\,(C_1,\ C_2) \xrightarrow{\{\epsilon,X\}} C_2}$$

Command Sequencing in Modular SOS

$$\boxed{\mathbf{Cmd} ::= \mathsf{skip}}$$

$$\boxed{\mathsf{Label} := \{\ldots\}}$$

$$\boxed{\mathbf{Cmd} ::= \mathsf{skip} \mid \mathsf{seq}\,(\mathbf{Cmd},\ \mathbf{Cmd})}$$

$$\boxed{\mathsf{Label} := \{\ldots\}}$$

$$\mathsf{seq}\,(\mathsf{skip},\ c) \xrightarrow{\{-\}} c$$

$$\frac{c_1 \xrightarrow{\{X\}} c_1'}{\mathsf{seq}\,(c_1,\ c_2) \xrightarrow{\{X\}} \mathsf{seq}\,(c_1',\ c_2)}$$

Labels

The "state" is encoded by labels on the transition relation. These labels are the *arrows* of a suitable product category:

- Composition of arrows is needed for consecutive transitions:
  $S \xrightarrow{a \to b} T \xrightarrow{b' \to c} R$ only if $b = b'$.

Labels

The "state" is encoded by labels on the transition relation. These labels are the *arrows* of a suitable product category:

- Composition of arrows is needed for consecutive transitions: $S \xrightarrow{a \to b} T \xrightarrow{b' \to c} R$ only if $b = b'$.
- Identity arrows express silent transitions $\{-\}$.

Labels

The "state" is encoded by labels on the transition relation. These labels are the *arrows* of a suitable product category:

- Composition of arrows is needed for consecutive transitions: $S \xrightarrow{a \to b} T \xrightarrow{b' \to c} R$ only if $b = b'$.
- Identity arrows express silent transitions $\{-\}$.
- Write-only components require multiple arrows, e.g.:

$$
\begin{aligned}
\text{print}(\text{"}\mathit{foo}\text{"});\text{print}(\text{"}\mathit{bar}\text{"}) &\xrightarrow{* \xrightarrow{\text{"}\mathit{foo}\text{"}} *} \text{skip};\text{print}(\text{"}\mathit{bar}\text{"}) \\
&\xrightarrow{* \xrightarrow{()} *} \text{print}(\text{"}\mathit{bar}\text{"}) \\
&\xrightarrow{* \xrightarrow{\text{"}\mathit{bar}\text{"}} *} \text{skip}
\end{aligned}
$$

Classes for Transition Relations

```
Class Arrows (O: Type): Type := Arrow: O → O → Type.
Infix "⟶" := Arrow (at level 90, right associativity).
```

```
Context
 (O : Type) {Ar: Arrows O} B.

Class Step :=
 step : ∀ {x y: O}, (x ⟶ y) → B → B → Prop.

Class LocalStep `{C : Constructor} :=
 localstep : ∀ {x y: O}, (x ⟶ y) → restr C → B → Prop.
```

Classes for Language Constructors

```
Class IP_Pair A B (inj: Inject A B) (prj: Project A B) := {
  H_i: ∀ x: A, prj (inj x) = Some x;
  H_p: ∀ b: B, match project b with
        | None ⇒ True
        | Some x ⇒ inj x = b
        end
}.

Class Constructor (name: string) A B
        (inj: Inject A B) (prj: Project A B) (ip: IP_Pair A B inj prj) :=
  placeholder: unit.
```

Class for Labels

```
Context
 (M: Type) (O_M: M → Type) (A_M: ∀ m, Arrows (O_M m))
‘{ip: IP_Pair M L}
 {O_L: L → Type} {A_L: ∀ l, Arrows (O_L l)}.

Class Label := {
 cover_O: ∀ m: M, O_M m = O_L (’ m);
 cover_A: ∀ m: M, A_M m =
  ⟪ fun T ⇒ Arrows T # eq_sym (cover_O m) ⟫ A_L (’ m)
}.
```

Example: Seq Encoded in Coq

```
Section seq.
 Context
  `{Seq: @Constructor "seq" (Cmd*Cmd) Cmd seq p_seq ip_seq}
  `{Skip: @Constructor "skip" unit Cmd skip p_skip ip_skip}
  `{label: Label M_none O_M_none A_M_none}
   {Step_Cmd: Step Obj Cmd}.

 Inductive lstep {x y: Obj} (ar: x ⟶ y): restr Seq → Cmd → Prop :=
  <...>
 Global Instance LS_seq: LocalStep Obj := @lstep.

 Lemma det_seq (c1 c2: Cmd):
  <...>
End seq.
```

Example: Cmd Encoded in Coq

```
Inductive s_Cmd {x y: O} (ar: x ⟶ y): Cmd → Cmd → Prop :=
| s_Cmd_skip:
   ∀ (c: restr Skip) (c': Cmd),
    localstep ar c c' → s_Cmd ar (I c) c'
| s_Cmd_seq:
   ∀ (c: restr Seq) (c': Cmd),
    let _ := (@s_Cmd: Step O Cmd) in
    localstep ar c c' → s_Cmd ar (I c) c'.

Instance S_Cmd: Step O Cmd := @s_Cmd.
```

Future Work

- Support for software verification.
  - I'm working on MSOS constructs for design contracts.
  - Currently developing a VCG.
  - Slicing of semantics.
- A notion of bisimulation for MSOS to support for reasoning about extensions. E.g. under which extensions does $S; (T; R) = (S; T); R$ hold?
  - The presented formalization is based on *first-order* inductive types.
  - Probably need a representation of inductive types à la System F.
- Relation between MSOS and monads.
  - Can't execute the semantics; would like to have an equivalent functional encoding.