Formal Component-Based Semantics

Ken Madlener,
Sjaak Smetsers,
Marko van Eekelen

Radboud University Nijmegen

August 28, 2011
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

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

\[\text{Cmd}[\text{break}] = \text{throw("breaking")}\]
A While-Loop with Break and Continue

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

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

\[
Cmd[\text{continue}] = \text{throw("continuing")}
\]
Exceptions in Modular SOS

\[
\text{Cmd} ::= \text{skip} \mid \text{throw(String)}
\]

\[
\text{Label} := \{ \epsilon : \text{String}, \ldots \}
\]

\[
\text{throw}(E) \xrightarrow{\{\epsilon' = E, \cdot\}} \text{skip}
\]

\[
\text{Cmd} ::= \text{eq(String, Cmd)}
\]

\[
\text{Label} := \{ \epsilon : \text{String}, \ldots \}
\]

\[
\epsilon = E
\]

\[
\text{eq}(E, C) \xrightarrow{\{\epsilon, \cdot\}} C
\]

\[
\text{Cmd} ::= \text{catch(Cmd, Cmd)}
\]

\[
\text{Label} := \{ \epsilon : \text{String}, \ldots \}
\]

\[
\begin{align*}
C_1 \xrightarrow{\{\epsilon, X\}} C_1' & \quad \epsilon = () \\
\text{catch}(C_1, C_2) \xrightarrow{\{\epsilon, X\}} \text{catch}(C_1', C_2)
\end{align*}
\]

\[
\begin{align*}
C_1 \xrightarrow{\{\epsilon, X\}} C_1' & \quad \epsilon \neq () \\
\text{catch}(C_1, C_2) \xrightarrow{\{\epsilon, X\}} C_2
\end{align*}
\]
Command Sequencing in Modular SOS

\[
\begin{align*}
\text{Cmd} & ::= \text{skip} \mid \text{seq (Cmd, Cmd)} \\
\text{Label} & ::= \{\ldots\}
\end{align*}
\]

\[
\begin{align*}
\text{seq (skip, } c) & \xrightarrow{\{-\}} c \\
\begin{array}{c}
\quad \\c_1 & \xrightarrow{\{X\}} c_1' \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{seq (c_1, c_2)} & \xrightarrow{\{X\}} \text{seq (c_1', c_2)}
\end{align*}
\]
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:

  \[
  S \xrightarrow{a \rightarrow b} T \xrightarrow{b' \rightarrow c} R \text{ only if } b = b'.
  \]
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:
  \[ S \xrightarrow{a\rightarrow b} T \xrightarrow{b'\rightarrow c} R \text{ only if } b = b'. \]
- Identity arrows express silent transitions \{−\}. 

Write-only components require multiple arrows, e.g.:

\[ \text{print("foo"); print("bar") } \]
\[ \xrightarrow{\ast} \xrightarrow{\ast} \text{skip; print("bar") } \]
\[ \xrightarrow{\ast} \xrightarrow{\ast} \text{print("bar") } \]
\[ \xrightarrow{\ast} \text{skip} \]
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:
  \[ S \overset{a \rightarrow b}{\rightarrow} T \overset{b' \rightarrow c}{\rightarrow} R \text{ only if } b = b'. \]

- Identity arrows express silent transitions \{\_\}.\]

- Write-only components require multiple arrows, e.g.:

\[
\begin{align*}
\text{print("foo")}; \text{print("bar")} & \quad \overset{\text{"foo"}}{\rightarrow} \overset{*}{\rightarrow} \text{skip; print("bar")} \\
\overset{\text{()}}{\rightarrow} \overset{*}{\rightarrow} \text{print("bar")} \\
\overset{\text{"bar"}}{\rightarrow} \overset{*}{\rightarrow} \text{skip}
\end{align*}
\]
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: \forall x: A, prj (inj x) = Some x; 
   H_p: \forall b: B, match project b with 
       | None \Rightarrow True 
       | Some x \Rightarrow 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

\[(M: \text{Type}) \ (O_M: M \rightarrow \text{Type}) \ (A_M: \forall m, \text{Arrows}(O_M m))\]
\['\{\text{ip: IP\_Pair} \ M \ L\}\]
\{O_L: L \rightarrow \text{Type}\} \ \{A_L: \forall l, \text{Arrows}(O_L l)\}\}.

Class Label := \{ \]
\text{cover}_O: \forall m: M, O_M m = O_L (’ m);
\text{cover}_A: \forall m: M, A_M m =
\langle\langle \text{fun} \ T \Rightarrow \text{Arrows} \ T \ # \ \text{eq\_sym} \ (\text{cover}_O m) \rangle \rangle \ 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 \rightarrow y): restr Seq \rightarrow Cmd \rightarrow Prop :=

Global Instance LS_seq: LocalStep Obj := @lstep.

Lemma det_seq (c_1 c_2: Cmd):

End seq.
Example: Cmd Encoded in Coq

\textbf{Inductive} s\_Cmd \{x y: O\} (ar: x \rightarrow y): Cmd \rightarrow Cmd \rightarrow \text{Prop} :=
\begin{align*}
| \text{s\_Cmd\_skip:} \\
& \forall (c: \text{restr Skip}) (c': \text{Cmd}), \\
& \quad \text{localstep} \ ar \ c \ c' \rightarrow s\_Cmd \ ar \ (I \ c) \ c' \\
| \text{s\_Cmd\_seq:} \\
& \forall (c: \text{restr Seq}) (c': \text{Cmd}), \\
& \quad \text{let } _:= (@s\_Cmd: \text{Step O Cmd}) \text{ in} \\
& \quad \text{localstep} \ ar \ c \ c' \rightarrow s\_Cmd \ ar \ (I \ c) \ c'.
\end{align*}

\textbf{Instance} S\_Cmd: \text{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.