Formal Component-Based Semantics

Ken Madlener,
Sjaak Smetsers,
Marko van Eekelen

Radboud University Nijmegen, the Netherlands

September 7, 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.

- Combines basic language constructs contained in an open-ended repository to develop languages.
- Components can be defined using Action Semantics or Modular SOS.
- Think Intentional Programming (Microsoft)

This talk presents a first formalization, developed in CoQ.

My longer-term goal is support for verification of concrete programs.
A While-Loop with Break

\[
\begin{align*}
\text{Cmd[while (E) C]} &= \text{catch(\text{cond-loop(Exp[E], Cmd[C])}},\text{eq(”breaking”, skip)}) \\
\text{Cmd[break]} &= \text{throw(”breaking”)}
\end{align*}
\]
A While-Loop with Break and Continue

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

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

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

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

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

\[\text{throw}(E) \frac{\{\epsilon' = E, -\}}{\epsilon = E} \rightarrow \text{skip}\]

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

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

\[\epsilon = E \]

\[\text{eq}(E, C) \frac{\{\epsilon, -\}}{\epsilon = E} \rightarrow C\]

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

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

\[\text{catch}(C_1, C_2) \frac{\{\epsilon, X\}}{\epsilon = (\)} \rightarrow C_1\]

\[\text{catch}(C_1, C_2) \frac{\{\epsilon, X\}}{\epsilon \neq (\)} \rightarrow C_2\]

\[\text{catch}(\text{skip}, C) \frac{\{\_\}}{} \rightarrow C\]
Labels

The "state" is encoded by labels on the transition relation. These labels are the arrows of a suitable product category $\mathbb{A} = \prod_{i \in I} A_i$:

- Composition of arrows is needed for consecutive transitions:

  $S \xrightarrow{a \rightarrow b} T \xrightarrow{b' \rightarrow c} R$ only if $b = b'$.

- Identity arrows express silent transitions "$\rightarrow$".

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

  $\text{print("foo") print("bar")}$ $\rightarrow$ $\ast$ $\rightarrow$ skip; $\text{print("bar")}$ $\rightarrow$ $\ast$ $\rightarrow$ $\text{print("bar")}$ $\rightarrow$ $\ast$ $\rightarrow$ skip
Labels

The "state" is encoded by labels on the transition relation. These labels are the *arrows* of a suitable product category $\mathbb{A} = \prod_{i \in I} \mathbb{A}_i$:

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

- Identity arrows express silent transitions "−".

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

$$\text{print("foo"); print("bar") \to \ast \to \text{skip; print("bar") \to \ast \to \text{print("bar") \to \ast \to \text{skip}}.$$

Three different label categories $\mathbb{A}_i$ are used to model RO/RW/WO permissions of each label entity (more later).
The "state" is encoded by labels on the transition relation. These labels are the arrows of a suitable product category $\mathbb{A} = \prod_{i \in I} \mathbb{A}_i$:

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

  ```
  print("foo"); print("bar") \xrightarrow{\ast} \xrightarrow{\ast \ "foo"} skip; print("bar")
  \xrightarrow{\ast \ ()} \xrightarrow{\ast} \xrightarrow{\ast \ "bar"} print("bar")
  \xrightarrow{\ast \ "bar"} \xrightarrow{\ast} \xrightarrow{\ast} skip
  ```

Three different label categories $\mathbb{A}_i$ are used to model RO/RW/WO permissions of each label entity (more later).
The "state" is encoded by labels on the transition relation. These labels are the *arrows* of a suitable product category \( \mathbb{A} = \prod_{i \in I} \mathbb{A}_i \):

- Composition of arrows is needed for consecutive transitions:
  \[ S \xrightarrow{a \rightarrow b} T \xrightarrow{b' \rightarrow c} R \] only if \( b = b' \).
- Identity arrows express silent transitions "−".
- Write-only components require multiple arrows, e.g.:
  \[
  \text{print("foo"); print("bar")} \quad \xrightarrow{* \rightarrow *} \quad \text{skip; print("bar")}
  \]
  \[
  \quad \xrightarrow{* \rightarrow *} \quad \text{print("bar")}
  \]
  \[
  \quad \xrightarrow{* \rightarrow *} \quad \text{skip}
  \]
- Three different label categories \( \mathbb{A}_i \) are used to model RO/RW/WO permissions of each label entity (more later).
Projections

The label category $\mathbb{A}$ is split into

- a *transparent* part $\xrightarrow{P_M} \prod_{i \in M} \mathbb{A}_i$ for the *mentioned* entities;
- an *opaque* part $\xrightarrow{P_U} \mathbb{U}$ for the *unmentioned* entities.

That is, the components *specify* the label entities $\mathbb{A}_i$ for $i \in M$, while it is parameterized on $\mathbb{U}$ (more on next slide).

In our formalization $(\alpha : x \rightarrow y)$:

\[
\begin{align*}
\text{throw}(E) \xrightarrow{\{\epsilon' = E, -\}} \text{skip} &\quad \leadsto \quad \frac{(\pi_\epsilon \circ P_M) y = E}{\text{throw}(E) \xrightarrow{\alpha} \text{skip}} \\
C_1 \xrightarrow{\{\epsilon, X\}} C'_1 &\quad \epsilon = () \\
\text{catch}(C_1, C_2) \xrightarrow{\{\epsilon, X\}} \text{catch}(C'_1, C_2) &\quad \leadsto \quad \frac{(\pi_\epsilon \circ P_M) x = ()}{\text{catch}(C_1, C_2) \xrightarrow{\alpha} \text{catch}(C'_1, C_2)}
\end{align*}
\]
Projections

The label category $\mathbb{A}$ is split into

- a *transparant* part $\mathbb{A} \xrightarrow{P_M} \prod_{i \in M} \mathbb{A}_i$ for the *mentioned* entities;
- an *opaque* part $\mathbb{A} \xrightarrow{P_U} \mathbb{U}$ for the *unmentioned* entities.

That is, the components *specify* the label entities $\mathbb{A}_i$ for $i \in M$, while it is parameterized on $\mathbb{U}$ (more on next slide).

In our formalization ($\alpha : x \rightarrow y$):

$$
\text{throw}(E) \xrightarrow{\{\epsilon' = E, -\}} \text{skip} \quad \leadsto \quad (\pi_\epsilon \circ P_M)y = E \quad P_U\alpha = 1
$$

$$
\text{throw}(E) \xrightarrow{\alpha} \text{skip}
$$

$$
\text{catch}(C_1, C_2) \xrightarrow{\{\epsilon, X\}} \text{catch}(C_1', C_2) \quad \leadsto \quad C_1 \xrightarrow{\alpha} C_1' \quad (\pi_\epsilon \circ P_M)x = ()
$$

$$
\text{catch}(C_1, C_2) \xrightarrow{\alpha} \text{catch}(C_1', C_2)
$$
Full Transition Relations

Consider the following components for Skip and Seq:

\[
\text{Cmd ::= skip | seq (Cmd, Cmd)}
\]

\[
\text{Label ::= \{\ldots\}}
\]

\[
\text{seq (skip, c) } \xrightarrow{\{-\}} \text{seq } c
\]

\[
\text{c} \xrightarrow{\{X\}} \text{Cmd } c'
\]

\[
\text{seq (c_1, c_2) } \xrightarrow{\{X\}} \text{seq } \text{seq (c'_1, c_2)}
\]

A *full* transition relation is built from local transition relations, e.g.:

\[
\text{c } \xrightarrow{\alpha} \text{skip } c' \quad (c \in \Gamma_{\text{skip}})
\]

\[
\text{c } \xrightarrow{\alpha} \text{Cmd } c' \quad (c \in \Gamma_{\text{seq}})
\]
Example: Type Class Instances for Skip-Seq

```
Type Class Instances for Skip-Seq

S_Cmd
  \downarrow
LS_Seq
  \downarrow
Skip  label_none
     \downarrow
      ...
LS_Skip
  \downarrow
Skip  Seq  label_none
     \downarrow
      ...
```

Madlener, Smetsers, van Eekelen (RU)
Classes for Transition Relations

Context
\((O : \textbf{Type}) \{\text{Ar: Arrows } O\} (\Gamma : \textbf{Type})\).

Class \textbf{Step} :=
\(\text{step} : \forall \{x, y : O\}, (x \rightarrow y) \rightarrow \Gamma \rightarrow \Gamma \rightarrow \textbf{Prop}\).

Class \textbf{Constructor} (name: string) A B
\((\text{inj: Inject } A B) (\text{prj: Project } A B) (\text{ip: IP_Pair } A B \text{ inj prj}) :=\)
placeholder: unit.

Class \textbf{LocalStep} \{'C : Constructor\} :=
\(\text{localstep} : \forall \{x, y : O\}, (x \rightarrow y) \rightarrow \text{restr } C \rightarrow \Gamma \rightarrow \textbf{Prop}\).

(restr \(C\) is the \textsc{Coq}-equivalent of \(\Gamma_C\) )
Classes for Transition Relations

Context

\((O : \text{Type}) \{\text{Ar: Arrows } O\} (\Gamma : \text{Type})\).

Class Step :=

step : \(\forall \{x y: O\}, (x \rightarrow y) \rightarrow \Gamma \rightarrow \Gamma \rightarrow \text{Prop} \).

Class Constructor (name: string) A B

(inj: Inject A B) (prj: Project A B) (ip: IP_Pair A B inj prj) :=

placeholder: unit.

Class LocalStep \(\{C : \text{Constructor}\} :=

localstep : \(\forall \{x y: O\}, (x \rightarrow y) \rightarrow \text{restr } C \rightarrow \Gamma \rightarrow \text{Prop} \).

(restr C is the Coq-equivalent of \(\Gamma_C\))
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 :=
(* step rules here *)

Global Instance LS_seq: LocalStep Obj := @lstep.

Lemma det_seq (c_1 c_2: Cmd):
(* a property saying that seq is deterministic, discussed later *)

End seq.
**Example: **\textbf{Cmd Encoded in CoQ}

\textbf{Inductive} \textit{Cmd} \textbf{:=} \textit{skip} \mid \textit{seq}: \textit{Cmd} \rightarrow \textit{Cmd} \rightarrow \textit{Cmd}.

\textit{(* generation of Constructor instances here *)}

\textbf{Inductive} \textit{s.Cmd} \{\textit{x y: O}\} (\textit{ar: x \rightarrow y}): \textit{Cmd} \rightarrow \textit{Cmd} \rightarrow \text{Prop} \textbf{:=} \textit{\mid s.Cmd.skip:}
\begin{align*}
\forall (c: \text{restr Skip}) (c': \text{Cmd}), \\
\text{localstep} \text{ ar } c \text{ c'} \rightarrow \textit{s.Cmd} \text{ ar (l c) c'}
\end{align*}
\textit{\mid s.Cmd.seq:}
\begin{align*}
\forall (c: \text{restr Seq}) (c': \text{Cmd}), \\
\text{let } _ := \text{@s.Cmd: Step O Cmd} \text{ in} \\
\text{localstep} \text{ ar } c \text{ c'} \rightarrow \textit{s.Cmd} \text{ ar (l c) c'}.
\end{align*}

\textbf{Instance} \textit{S.Cmd: Step O Cmd} \textbf{:= } \textit{@s.Cmd}. 
**Arrows**

Class Arrows (O: Type): Type := Arrow: O → O → Type.

Arrow instances for the RO/RW/WO entity categories:

- **Instance arrows_ro** (O: Type): Arrows O := \( \lambda x y, x = y \).
- **Instance arrows_rw** (O: Type): Arrows O := \( \lambda _, unit \).
- **Instance arrows_wo** (O: Type): Arrows unit := \( \lambda x y, list O \).

Arrow instance for products:

- **Instance arrows_prod** (I: Type) (O: I → Type): Arrows (\( \forall i, O i \)) := \( \lambda x y, \forall i, x i \rightarrow y i \).

A category defines an equivalence relation on its corresponding Arrows instance (omitted here).
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) }.
We consider a class of well-behaved properties that permit modular proof:

**Definition** admissible $\Gamma \ (P : \text{Step } O \Gamma \to \Gamma \to \text{Prop}) :=$

$\forall \ (C : \text{Constructor name } A \Gamma) \ (S : \text{Step } O \Gamma) \ (b : \text{restr } C),$

$P \ (\text{globalize} \ (\text{localize } C \ S)) \ (I \ b) \to P \ S \ (I \ b).$

“$P \ S \ (I \ b)$ does not depend on anything but the localized version of $S$”

**Lemma**

**Determinism is admissible.**

It is shown in the paper that determinism of Skip-Seq can be shown by providing a proof for each component, e.g. for Seq:

**Lemma** $\text{det_seq} \ (c_1 \ c_2 : \text{Cmd})$:  
localize Skip $\text{Step_Cmd} = \text{LS_skip} \to$

det_global $\text{Step_Cmd} \ c_1 \to \text{det_local } \text{LS_seq} \ (\text{Seq} \cdot (c_1, c_2)).$
Modular Proof

We consider a class of well-behaved properties that permit modular proof:

**Definition** admissible \( \Gamma \) (\( P: \text{Step } O \; \Gamma \rightarrow \Gamma \rightarrow \text{Prop} \)) :=
\[
\forall \; \forall ' \left( C: \text{Constructor name } A \; \Gamma \right) \left( S: \text{Step } O \; \Gamma \right) \left( b: \text{restr } C \right),
\]
\[
P \left( \text{globalize } \left( \text{localize } C \; S \right) \right) \left( I \; b \right) \rightarrow P \; S \; (I \; b).
\]

“\( P \; S \; (I \; b) \) does not depend on anything but the localized version of \( S \)”

**Lemma**

*Determinism is admissible.*

It is shown in the paper that determinism of Skip-Seq can be shown by providing a proof for each component, e.g. for Seq:

**Lemma** \( \text{det} \; \text{seq} \; (c_1 \; c_2: \text{Cmd}) \):

\[
\text{localize } \text{Skip} \; \text{Step}\; \text{Cmd} = \text{LS}\; \text{skip} \rightarrow
\]
\[
\text{det}\; \text{global} \; \text{Step}\; \text{Cmd} \; c_1 \rightarrow \text{det}\; \text{local} \; \text{LS}\; \text{seq} \; (\text{Seq} \cdot (c_1, \; c_2)).
\]
Future Work

- Support for software verification.
  - I’m working on MSOS constructs for design contracts.
  - Currently developing a VCG.
  - Slicing of semantics.
- Can we support reasoning about language 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.