Certified Abstract Machines for Skeletal Semantics

CPP 2022

Guillaume Ambal, Sergueï Lenglet, Alan Schmitt

January 17, 2022

Defining a Language on Paper

Example: Call-by-Value λ-calculus

$$
\begin{array}{rl}
\text{Variables} & x \in \mathcal{V} \\
\text{Term} & t ::= \; x \mid t \; t \mid \lambda x.t \\
\text{Closure} & c ::= \; (x, t, s) \\
\text{Environment} & s ::= \; [(x_1 \mapsto c_1), \ldots, (x_n \mapsto c_n)]
\end{array}
$$

$$
\frac{s(x) = c}{s, x \Downarrow c} \qquad \frac{}{s, \lambda x.t \Downarrow (x, t, s)}
$$

$$
\frac{s, t_1 \Downarrow (x, t, s') \qquad s, t_2 \Downarrow c' \qquad (s' + \{x \mapsto c'\}), t \Downarrow c}{s, (t_1 \; t_2) \Downarrow c}
$$

Defining a Language with a Computer

In a proof assistant, from scratch

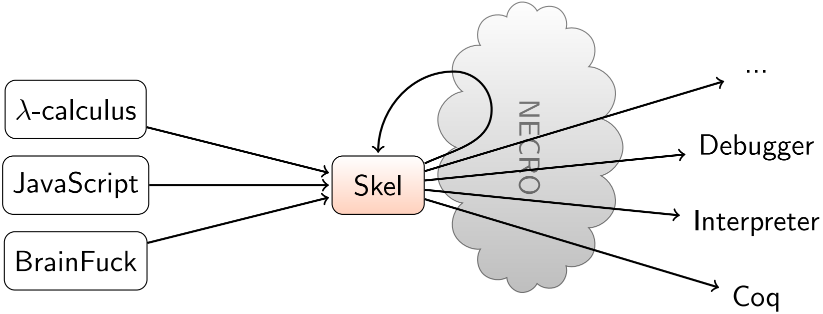
- Coq
- Isabelle/HOL
- Agda, Twelf, ...

In a convenient Framework

- Ott, Lem
- $\mathbb{K}$
- Skeletal Semantics

Skeletal Semantics

- Recent framework (first definition: POPL 2019)
- Meta-language (Skel) to define programming languages
- Toolbox to manipulate semantics: Necro.



Skeletal Semantics for CbV λ-calculus

```
type ident

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)

type clos =
| Clos (ident, lterm, env)

type env

term extEnv: (env,ident,clos) → env
term getEnv: (ident,env) → clos
```

```
term eval (s:env) (l:lterm): clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
```

Skeletal Semantics for CbV λ-calculus

```
type ident

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)

type clos =
| Clos (ident, lterm, env)

type env

term extEnv: (env,ident,clos) → env
term getEnv: (ident,env) → clos
```

Unspecified Types

We do not explicit what the elements look like.

E.g., there exist variables.

Skeletal Semantics for CbV λ-calculus

```
type ident

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)

type clos =
| Clos (ident, lterm, env)

type env

term extEnv: (env,ident,clos) → env
term getEnv: (ident,env) → clos
```

Specified Types

Defined as algebraic data-types with constructors.

Skeletal Semantics for CbV λ-calculus

```
type ident

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)

type clos =
| Clos (ident, lterm, env)

type env

term extEnv: (env,ident,clos) → env
term getEnv: (ident,env) → clos
```

Unspecified Terms

For when the actual implementation is not important.

E.g., we can extend an environment, and we can read the mapping of a variable.

Skeletal Semantics for CbV λ-calculus

Specified Term

Evaluation functions we want to describe.

There are associated with a given definition.

```
term eval (s:env) (l:lterm): clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
```

Skeletal Semantics for CbV λ-calculus

Branching

Construction of the meta-language to list several possible behaviors.

Can be used to represent pattern-machings (like here), conditional statements, non-deterministic choices, etc.

```
term eval (s:env) (l:lterm): clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
```

Skeletal Semantics for CbV λ-calculus

```
type ident

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)

type clos =
| Clos (ident, lterm, env)

type env

term extEnv: (env,ident,clos) → env
term getEnv: (ident,env) → clos
```

```
term eval (s:env) (l:lterm): clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
```

Syntax of Skel

$$
\begin{aligned}
\text{Identifier} \quad & x \in \mathcal{V} \\
\text{Term} \quad & t ::= x \mid C\ t \mid (t, \ldots, t) \mid \lambda p.S \\
\text{Skeleton} \quad & S ::= t_0\ t_1 \ldots t_n \mid \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2 \\
& \quad\quad \mid \texttt{Branching}(S, \ldots, S) \mid t \\
\text{Pattern} \quad & p ::= \_ \mid x \mid C\ p \mid (p, \ldots, p)
\end{aligned}
$$

Semantics of Skel?

Main semantics of Skel is Big-Step.

Wish for a different format of semantics: Abstract Machines.
Notably, would like an executable semantics.

For this, known technique by Danvy et al.:

- CPS Transform
- Defunctionalization

Non-Deterministic Abstract Machine

$$\langle \texttt{Branching}(I), \Sigma, \kappa \rangle_{\mathsf{sk}} \rightarrow \langle S, \Sigma, \kappa \rangle_{\mathsf{sk}} \quad \texttt{for} \ (S \in I)$$

$$\langle \texttt{let} \ p = S_1 \ \texttt{in} \ S_2, \Sigma, \kappa \rangle_{\mathsf{sk}} \rightarrow \langle S_1, \Sigma, \lceil \texttt{let} \ p = \square \ \texttt{in} \ S_2, \Sigma \rfloor :: \kappa \rangle_{\mathsf{sk}}$$

$$\cdots \rightarrow \cdots$$

$$\langle \lceil \texttt{let} \ p = \square \ \texttt{in} \ S, \Sigma \rfloor :: \kappa, r \rangle_{\mathsf{k}} \rightarrow \langle p, r, \Sigma, \lceil S, \square \rfloor :: \kappa \rangle_{\mathsf{pat}}$$

$$\langle \lceil S, \square \rfloor :: \kappa, \Sigma \rangle_{\mathsf{k}} \rightarrow \langle S, \Sigma, \kappa \rangle_{\mathsf{sk}}$$

Non-Deterministic Abstract Machine

$$\langle \texttt{Branching}(l), \Sigma, \kappa \rangle_{\mathsf{sk}} \to \langle S, \Sigma, \kappa \rangle_{\mathsf{sk}} \quad \texttt{for } (S \in l)$$
$$\langle \texttt{let } p = S_1 \texttt{ in } S_2, \Sigma, \kappa \rangle_{\mathsf{sk}} \to \langle S_1, \Sigma, \lceil \texttt{let } p = \Box \texttt{ in } S_2, \Sigma \rfloor :: \kappa \rangle_{\mathsf{sk}}$$
$$\cdots \to \cdots$$
$$\langle \lceil \texttt{let } p = \Box \texttt{ in } S, \Sigma \rfloor :: \kappa, r \rangle_{\mathsf{k}} \to \langle p, r, \Sigma, \lceil S, \Box \rfloor :: \kappa \rangle_{\mathsf{pat}}$$
$$\langle \lceil S, \Box \rfloor :: \kappa, \Sigma \rangle_{\mathsf{k}} \to \langle S, \Sigma, \kappa \rangle_{\mathsf{sk}}$$

Problem: still non-deterministic, so not really computable...
Next: deterministic AM, with backtracking.

Deterministic Abstract Machine

$$\langle \texttt{Branching}(S :: l), \Sigma, \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle S, \Sigma, \kappa, \llbracket \texttt{Branching}(l), \Sigma, \kappa \rrbracket :: f \rangle_{\mathsf{sk}}$$

$$\langle \texttt{Branching}([]), \Sigma, \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle f \rangle_{\mathsf{fk}}$$

$$\langle \texttt{let}\ \ p = S_1\ \ \texttt{in}\ \ S_2, \Sigma, \kappa, f \rangle_{\mathsf{sk}} \rightarrow \langle S_1, \Sigma, \lceil \texttt{let}\ \ p = \square\ \ \texttt{in}\ \ S_2, \Sigma \rfloor :: \kappa, f \rangle_{\mathsf{sk}}$$

$$\cdots \rightarrow \cdots$$

$$\langle \lceil \texttt{let}\ \ p = \square\ \ \texttt{in}\ \ S, \Sigma \rfloor :: \kappa, r, f \rangle_{\mathsf{k}} \rightarrow \langle p, r, \Sigma, \lceil S, \square \rfloor :: \kappa, f \rangle_{\mathsf{pat}}$$

$$\langle \lceil S, \square \rfloor :: \kappa, \Sigma, f \rangle_{\mathsf{k}} \rightarrow \langle S, \Sigma, \kappa, f \rangle_{\mathsf{sk}}$$

$$\cdots \rightarrow \cdots$$

$$\langle \llbracket S, \Sigma, \kappa \rrbracket :: f \rangle_{\mathsf{fk}} \rightarrow \langle S, \Sigma, \kappa, f \rangle_{\mathsf{sk}}$$

Equivalence Certification

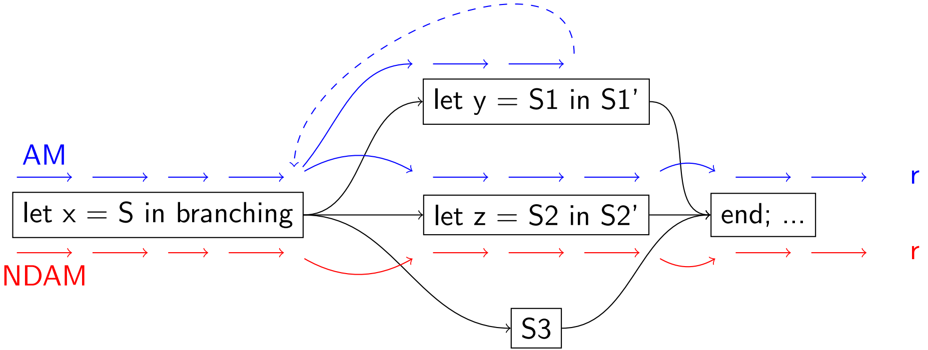
Definitions in Coq:

- Big-Step semantics already defined
- We define the Non-Deterministic Abstract Machine
  ```
  Inductive step: state -> state -> Prop
  ```
- We define the Deterministic Abstract Machine
  ```
  Definition step (a: state) : option state
  ```

Certification:

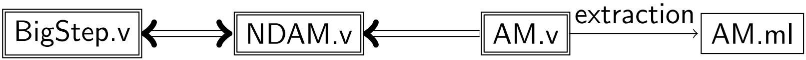
- We prove Big-Step and NDAM are equivalent (standard proof)
- We prove AM is sound w.r.t. NDAM

AM ⇒ NDAM



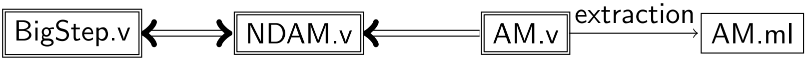
Certified Interpreter

Now we have different semantics for Skel:

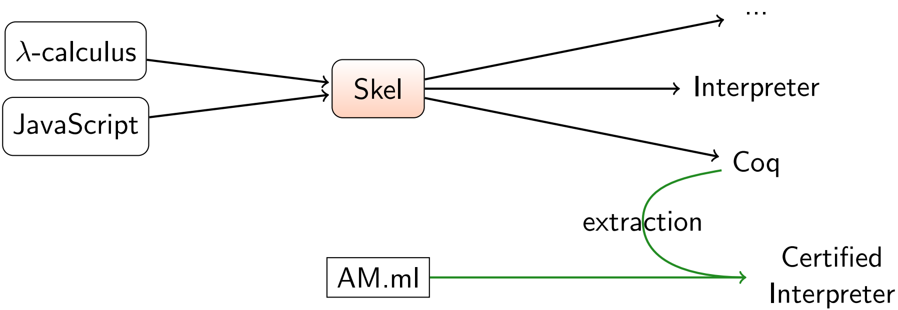


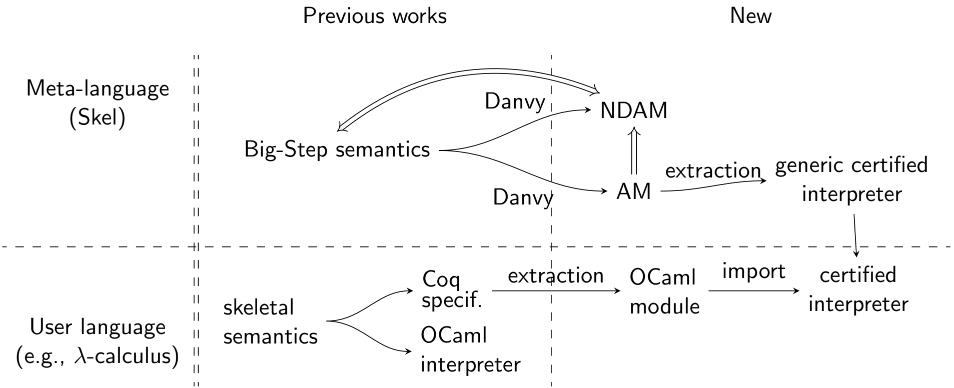
Certified Interpreter

Now we have different semantics for Skel:



For the user, we can produce a certified interpreter:




Previous works
New
Meta-language
(Skel)
User language
(e.g., λ-calculus)
Big-Step semantics
Danvy
NDAM
Danvy
AM
extraction
generic certified
interpreter
skeletal
semantics
Coq
specif.
OCaml
interpreter
extraction
OCaml
module
import
certified
interpreter