Certified Abstract Machines for Skeletal Semantics

TYPES 2022

Guillaume Ambal, Sergueï Lenglet, Alan Schmitt

June 23, 2022
Defining a Language on Paper

Example: Call-by-Value $\lambda$-calculus

Variables $x \in V$

Term $t ::= x \mid t \ t \mid \lambda x.t$

Closure $c ::= (x, t, s)$

Environment $s ::= [(x_1 \mapsto c_1), \ldots, (x_n \mapsto c_n)]$

$s(x) = c$

$s, x \Downarrow c$

$s, \lambda x.t \Downarrow (x, t, s)$

$s, t_1 \Downarrow (x, t, s')$

$s, t_2 \Downarrow c'$

$(s' + \{x \mapsto c'\}), t \Downarrow c$

$s, (t_1 \ t_2) \Downarrow c$
Defining a Language with a Computer

In a proof assistant, from scratch
- Coq
- Isabelle/HOL
- Agda, Twelf, …

In a convenient Framework
- Ott, Lem
- 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 $\lambda$-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
Skeletal Semantics for CbV $\lambda$-calculus

```ocaml
type ident

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

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

Unspecified Types

We do not explicit what the elements look like.

E.g., there exist variables.
```

```ocaml
type env

term extEnv: (env,ident,clos) \rightarrow env
term getEnv: (ident,env) \rightarrow clos
```
Skeletal Semantics for CbV $\lambda$-calculus

```ocaml
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

```ml
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.
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 $\lambda$-calculus

**Branching**

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

Can be used to represent pattern-matchings (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 $\lambda$-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
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
Abstract Machines

Non-Deterministic Abstract Machine

\[
\langle \text{let } p = S_1 \text{ in } S_2, \kappa \rangle_{sk} \rightarrow \langle S_1, [\text{let } p = \Box \text{ in } S_2] :: \kappa \rangle_{sk}
\]

\[
\langle \text{Branch}(l), \kappa \rangle_{sk} \rightarrow \langle S, \kappa \rangle_{sk} \quad \text{for } (S \in l)
\]

\[
\ldots \rightarrow \ldots
\]

Problem: still non-deterministic, so not really computable...

Next: deterministic AM, with backtracking.

Guillaume Ambal
Certified AM for Skeletal Semantics
June 23, 2022 7 / 11
Non-Deterministic Abstract Machine

\[ \langle \text{let } p = S_1 \text{ in } S_2, \kappa \rangle_{sk} \rightarrow \langle S_1, [\text{let } p = \Box \text{ in } S_2] :: \kappa \rangle_{sk} \]
\[ \langle \text{Branch}(l), \kappa \rangle_{sk} \rightarrow \langle S, \kappa \rangle_{sk} \text{ for } (S \in l) \]
\[ \ldots \rightarrow \ldots \]

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

\[
\langle \text{let } p = S_1 \text{ in } S_2, \kappa, f \rangle_{\text{sk}} \to \langle S_1, \lceil \text{let } p = \square \text{ in } S_2 \rceil, \kappa, f \rangle_{\text{sk}} \\
\langle \text{Branch}(S :: l), \kappa, f \rangle_{\text{sk}} \to \langle S, \kappa, \lceil \text{Branch}(l), \kappa \rceil, f \rangle_{\text{sk}} \\
\langle \text{Branch}([]), \kappa, f \rangle_{\text{sk}} \to \langle f \rangle_{f_k} \\
\ldots \to \ldots \\
\langle \lceil S, \kappa \rceil, f \rangle_{f_k} \to \langle S, \kappa, f \rangle_{\text{sk}}
\]
Equivalence Certification

Definitions in Coq:

- Big-Step semantics already defined
- We define the Non-Deterministic Abstract Machine
  
  \[
  \text{Inductive } \text{step} : \text{state} \rightarrow \text{state} \rightarrow \text{Prop}
  \]
- We define the Deterministic Abstract Machine
  
  \[
  \text{Definition } \text{step} (a : \text{state}) : \text{option state}
  \]

Certification:

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

Now we have different semantics for Skel:

- BigStep.v
- NDAM.v
- AM.v
- AM.ml

For the user, we can produce a certified interpreter:

- λ-calculus
- JavaScript

\[\text{Skel} \rightarrow \text{Interpreter} \rightarrow \text{Coq} \]

\[\text{extraction} \quad \text{AM.ml} \rightarrow \text{Certified Interpreter}\]
Conclusion

Previous works

Meta-language (Skel)

Big-Step semantics

User language (e.g., $\lambda$-calculus)

Meta-language (Skel)

Big-Step semantics

Coq specific.

NDAM

Danvy

AM

extraction

generic certified interpreter

skeletal semantics

OCaml interpreter

Coq specific.

extraction

OCaml module

import

certified interpreter

Danvy

NDAM

AM

extraction

generic certified interpreter

Guillaume Ambal

Certified AM for Skeletal Semantics

June 23, 2022
Syntax of Skel

**Identifier** \( x \in \mathcal{V} \)

**Term** \( t ::= x \mid C \ t \mid (t, \ldots, t) \mid \lambda p.S \)

**Skeleton** \( S ::= t_0 \ t_1 \ldots \ t_n \mid \text{let} \ p = S_1 \ \text{in} \ S_2 \)

\[ \quad \mid \text{Branch}(S, \ldots, S) \mid t \]

**Pattern** \( p ::= \_ \mid x \mid C \ p \mid (p, \ldots, p) \)
Non-Deterministic Abstract Machine

\[
\langle \text{Branch}(l), \Sigma, \kappa \rangle_{sk} \rightarrow \langle S, \Sigma, \kappa \rangle_{sk} \quad \text{for } (S \in l)
\]

\[
\langle \text{let } p = S_1 \text{ in } S_2, \Sigma, \kappa \rangle_{sk} \rightarrow \langle S_1, \Sigma, \lceil \text{let } p = \Box \text{ in } S_2, \Sigma \rceil :: \kappa \rangle_{sk}
\]

\[
\ldots \rightarrow \ldots
\]

\[
\langle \lceil \text{let } p = \Box \text{ in } S, \Sigma \rceil :: \kappa, r \rangle_{kr} \rightarrow \langle p, r, \Sigma, \lceil S, \Box \rceil :: \kappa \rangle_{pat}
\]

\[
\langle \lceil S, \Box \rceil :: \kappa, \Sigma \rangle_{ke} \rightarrow \langle S, \Sigma, \kappa \rangle_{sk}
\]

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

\[
\langle \text{Branch}(S :: l), \Sigma, \kappa, f \rangle_{sk} \rightarrow \langle S, \Sigma, \kappa, \llbracket \text{Branch}(l), \Sigma, \kappa \rrbracket :: f \rangle_{sk}
\]

\[
\langle \text{Branch}([]), \Sigma, \kappa, f \rangle_{sk} \rightarrow \langle f \rangle_{fk}
\]

\[
\langle \text{let } p = S_1 \text{ in } S_2, \Sigma, \kappa, f \rangle_{sk} \rightarrow \langle S_1, \Sigma, \llbracket \text{let } p = \Box \text{ in } S_2, \Sigma \rrbracket :: \kappa, f \rangle_{sk}
\]

\[
\ldots \rightarrow \ldots
\]

\[
\llbracket \text{let } p = \Box \text{ in } S, \Sigma \rrbracket :: \kappa, r, f \rangle_{kr} \rightarrow \langle p, r, \Sigma, \llbracket S, \Box \rrbracket :: \kappa, f \rangle_{pat}
\]

\[
\llbracket S, \Box \rrbracket :: \kappa, \Sigma, f \rangle_{ke} \rightarrow \langle S, \Sigma, \kappa, f \rangle_{sk}
\]

\[
\ldots \rightarrow \ldots
\]

\[
\llbracket \llbracket S, \Sigma, \kappa \rrbracket \rrbracket :: f \rangle_{fk} \rightarrow \langle S, \Sigma, \kappa, f \rangle_{sk}
\]
let x = S in branching

let y = S1 in S1'

let z = S2 in S2'

end; ...

S3