MPRI 2.4

From operational semantics to (verified) interpreter

François Pottier

2018

A naïve interpreter

An interpreter executes a program (represented by its AST).

Let us write one, in OCaml, by paraphrasing the small-step semantics.

Abstract syntax

This is the abstract syntax of the λ-calculus:

```
type var = int (* a de Bruijn index *)
type term =
  | Var of var
  | Lam of (* bind: *) term
  | App of term * term
```

For example, the term $\lambda x.x$ is represented as follows:

```
let id =
  Lam (Var 0)
```

Renaming

`lift_ i k` represents the renaming $\Uparrow^i (+k)$.

```
let rec lift_ i k (t : term) : term =
  match t with
  | Var x ->
      if x < i then t else Var (x + k)
  | Lam t ->
      Lam (lift_ (i + 1) k t)
  | App (t1, t2) ->
      App (lift_ i k t1, lift_ i k t2)

let lift k t =
  lift_ 0 k t
```

Thus, `lift k` represents $+k$. (This renaming adds k to every variable.)

It is used when one moves the term t down into k binders. (Next slide.)

Substitution

`subst_ i sigma` represents the substitution $\Uparrow^i \sigma$.

```
let rec subst_ i (sigma : var -> term) (t : term) : term =
  match t with
  | Var x ->
      if x < i then t else lift i (sigma (x - i))
  | Lam t ->
      Lam (subst_ (i + 1) sigma t)
  | App (t1, t2) ->
      App (subst_ i sigma t1, subst_ i sigma t2)

let subst sigma t =
  subst_ 0 sigma t
```

Thus, `subst sigma` represents σ.

Substitution

A substitution is encoded as a total function of variables to terms.

```
let singleton (u : term) : var -> term =
  function 0 -> u | x -> Var (x - 1)
```

`singleton u` represents the substitution $u \cdot id$.

Recognizing values

It is easy to test whether a term is a value:

```
let is_value = function
  | Var _
  | Lam _ ->
      true
  | App _ ->
     false
```

Performing one step of reduction

A direct transcription of Plotkin's definition of call-by-value reduction:

```
let rec step (t : term) : term option =
  match t with
  | Lam _ | Var _ -> None
  (* Plotkin's BetaV *)
  | App (Lam t, v) when is_value v ->
      Some (subst (singleton v) t)
  (* Plotkin's AppL *)
  | App (t, u) when not (is_value t) ->
      in_context (fun t' -> App (t', u)) (step t)
  (* Plotkin's AppVR *)
  | App (v, u) when is_value v ->
      in_context (fun u' -> App (v, u')) (step u)
  (* All cases covered already, but OCaml cannot see it. *)
  | App (_, _) ->
      assert false
```

We have guarded AppL so that AppL and AppVR are mutually exclusive.

Performing one step of reduction

`in_context` is just the `map` combinator of the type `_ option`.

```
let in_context f ox =
  match ox with
  | None ->   None
  | Some x -> Some (f x)
```

Performing many steps of reduction

To evaluate a term, one performs as many reduction steps as possible:

```
let rec eval (t : term) : term =
  match step t with
  | None ->
      t
  | Some t' ->
      eval t'
```

The function call `eval t` either diverges or returns an irreducible term, which must be either a value or stuck.

Sources of inefficiency

Unfortunately, this is a terribly inefficient way of interpreting programs.

At each reduction step, one must:

- Find the next redex, that is, decompose the term t as $E[\lambda(x.u)\ v]$.
  Time: $O(depth(E))$, that is, $O(height(t))$.
- Perform the substitution $u[v/x]$.
  Time: $O(size(u) \times size(v))$.
- Construct the term $E[u[v/x]]$.
  Time: $O(depth(E))$, that is, $O(height(t))$.

Thus, one reduction step requires much more than constant time!

There seem to be two main sources of inefficiency:

- We keep forgetting the current evaluation context,
  only to discover it again at the next reduction step.
- We perform costly substitutions.

Towards an alternative to small steps

A reduction sequence from an application $t_1\ t_2$ to a final value v always has the form:

$$t_1\ t_2 \longrightarrow^\star_{\text{cbv}} (\lambda x.u_1)\ t_2 \longrightarrow^\star_{\text{cbv}} (\lambda x.u_1)\ v_2 \longrightarrow_{\text{cbv}} u_1[v_2/x] \longrightarrow^\star_{\text{cbv}} v$$

where $t_1 \longrightarrow^\star_{\text{cbv}} \lambda x.u_1$ and $t_2 \longrightarrow^\star_{\text{cbv}} v_2$. That is,

Evaluate operator; evaluate operand; call; continue execution.

Idea: define a "big-step" relation $t \downarrow_{\text{cbv}} v$,
which relates a term directly with the final outcome v of its evaluation, and whose definition reflects the above structure.

Natural semantics, a.k.a. big-step semantics

The relation $t \downarrow_{\text{cbv}} v$ means that evaluating t terminates and produces v.

Here is its definition, for call-by-value:

$$\textsc{BigCbvValue}\quad \frac{}{v \downarrow_{\text{cbv}} v} \qquad\qquad \textsc{BigCbvApp}\quad \frac{t_1 \downarrow_{\text{cbv}} \lambda x.u_1 \qquad t_2 \downarrow_{\text{cbv}} v_2 \qquad u_1[v_2/x] \downarrow_{\text{cbv}} v}{t_1\ t_2 \downarrow_{\text{cbv}} v}$$

Exercise: define $\downarrow_{\text{cbn}}$.

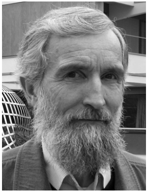
Example

Let us write $\downarrow$ for $\downarrow_{\text{cbv}}$, and “$v \downarrow \cdot$ ” for “$v \downarrow v$”.

$$
\dfrac{
  \dfrac{
    \lambda x.\lambda y.y\ x \downarrow \cdot
    \qquad
    \dfrac{\begin{array}{c}\lambda x.x \downarrow \cdot \\ 1 \downarrow \cdot \\ 1 \downarrow \cdot\end{array}}{(\lambda x.x)\ 1 \downarrow 1}
    \qquad
    \lambda y.y\ 1 \downarrow \cdot
  }{
    (\lambda x.\lambda y.y\ x)\ ((\lambda x.x)\ 1) \downarrow \lambda y.y\ 1
  }
  \qquad
  \lambda x.x \downarrow \cdot
  \qquad
  \dfrac{\begin{array}{c}\lambda x.x \downarrow \cdot \\ 1 \downarrow \cdot \\ 1 \downarrow \cdot\end{array}}{(\lambda x.x)\ 1 \downarrow 1}
}{
  (\lambda x.\lambda y.y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x) \downarrow 1
}
$$

Whereas a proof of $t \longrightarrow_{\text{cbv}} t'$ has linear structure,
a proof of $t \downarrow_{\text{cbv}} v$ has tree structure.

Some history



Martin-Löf uses big-step semantics, in English:

> To execute c(a), first execute c. If you get (λx) b as result, then continue by executing b(a/x). Thus c(a) has value d if c has value (λx) b and b(a/x) has value d.

He proposes type theory (1975) as a very high-level programming language in which both programs and specifications can be written.

Which is what we are doing today, in this lecture!

Per Martin-Löf,
Constructive Mathematics and Computer Programming, 1984.

Some history

Kahn promotes big-step operational semantics:

$$\rho \vdash \text{number}\ \mathrm{N} \Rightarrow \mathrm{N} \tag{1}$$

$$\rho \vdash \text{true} \Rightarrow \mathit{true} \tag{2}$$

$$\rho \vdash \text{false} \Rightarrow \mathit{false} \tag{3}$$

$$\rho \vdash \lambda \mathrm{P}.\mathrm{E} \Rightarrow [\![\lambda \mathrm{P}.\mathrm{E}, \rho]\!] \tag{4}$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}\ \mathrm{I} \mapsto \alpha}{\rho \vdash \text{ident}\ \mathrm{I} \Rightarrow \alpha} \tag{5}$$

$$\frac{\rho \vdash \mathrm{E}_1 \Rightarrow \mathit{true} \qquad \rho \vdash \mathrm{E}_2 \Rightarrow \alpha}{\rho \vdash \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3 \Rightarrow \alpha} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 \Rightarrow \mathit{false} \qquad \rho \vdash \mathrm{E}_3 \Rightarrow \alpha}{\rho \vdash \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3 \Rightarrow \alpha} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 \Rightarrow \alpha \qquad \rho \vdash \mathrm{E}_2 \Rightarrow \beta}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) \Rightarrow (\alpha, \beta)} \tag{8}$$

$$\frac{\rho \vdash \mathrm{E}_1 \Rightarrow [\![\lambda \mathrm{P}.\mathrm{E}, \rho_1]\!] \qquad \rho \vdash \mathrm{E}_2 \Rightarrow \alpha \qquad \rho_1 \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E} \Rightarrow \beta}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 \Rightarrow \beta} \tag{9}$$

$$\frac{\rho \vdash \mathrm{E}_2 \Rightarrow \alpha \qquad \rho \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E}_1 \Rightarrow \beta}{\rho \vdash \text{let}\ \mathrm{P} = \mathrm{E}_2\ \text{in}\ \mathrm{E}_1 \Rightarrow \beta} \tag{10}$$

$$\frac{\rho \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E}_2 \Rightarrow \alpha \qquad \rho \cdot \mathrm{P} \mapsto \alpha \vdash \mathrm{E}_1 \Rightarrow \beta}{\rho \vdash \text{letrec}\ \mathrm{P} = \mathrm{E}_2\ \text{in}\ \mathrm{E}_1 \Rightarrow \beta} \tag{11}$$

Figure 2. The dynamic semantics of mini-ML

He gives a big-step operational semantics of MiniML, a static type system, and a compilation scheme towards the CAM.

Gilles Kahn, Natural semantics, 1987.

A big-step interpreter

The call `eval t` attempts to compute a value v such that $t \downarrow_{\text{cbv}} v$ holds.

```
exception RuntimeError
let rec eval (t : term) : term =
  match t with
  | Lam _ | Var _ -> t
  | App (t1, t2) ->
      let v1 = eval t1 in
      let v2 = eval t2 in
      match v1 with
      | Lam u1 -> eval (subst (singleton v2) u1)
      | _      -> raise RuntimeError
```

If `eval` terminates normally, then it obviously returns a value;
but it can also fail to terminate or terminate with a runtime error. (Why?)

This interpreter does not forget and rediscover the evaluation context.
The context is now implicit in the interpreter's stack!

We could prove this interpreter correct, but will first optimize it further.

Equivalence between small-step and big-step semantics

Lemma (From big-step to small-step)

If $t \downarrow_{cbv} v$, then $t \longrightarrow^{\star}_{cbv} v$.

Proof.

By induction on the derivation of $t \downarrow_{\text{cbv}} v$.
Case BigCbvValue. We have $t = v$. The result is immediate.
Case BigCbvApp. t is $t_1\ t_2$, and we have three subderivations:

$$t_1 \downarrow_{\text{cbv}} \lambda x.u_1 \qquad t_2 \downarrow_{\text{cbv}} v_2 \qquad u_1[v_2/x] \downarrow_{\text{cbv}} v$$

Applying the ind. hyp. to them yields three reduction sequences:

$$t_1 \longrightarrow^{\star}_{\text{cbv}} \lambda x.u_1 \qquad t_2 \longrightarrow^{\star}_{\text{cbv}} v_2 \qquad u_1[v_2/x] \longrightarrow^{\star}_{\text{cbv}} v$$

By reducing under an evaluation context and by chaining, we obtain:

$$t_1\ t_2 \longrightarrow^{\star}_{\text{cbv}} (\lambda x.u_1)\ t_2 \longrightarrow^{\star}_{\text{cbv}} (\lambda x.u_1)\ v_2 \longrightarrow_{\text{cbv}} u_1[v_2/x] \longrightarrow^{\star}_{\text{cbv}} v$$

See `LambdaCalculusBigStep/bigcbv_star_cbv`. □

Equivalence between small-step and big-step semantics

Lemma (From small-step to big-step, preliminary)

If $t_1 \longrightarrow_{cbv} t_2$ *and* $t_2 \downarrow_{cbv} v$*, then* $t_1 \downarrow_{cbv} v$*.*

Proof (Sketch).

By induction on the first hypothesis and case analysis on the second hypothesis. See `LambdaCalculusBigStep/cbv_bigcbv_bigcbv`. □

Lemma (From small-step to big-step)

If $t \longrightarrow^{\star}_{cbv} v$*, then* $t \downarrow_{cbv} v$*.*

Proof.

By induction on the first hypothesis, using $v \downarrow_{\text{cbv}} v$ in the base case and the above lemma in the inductive case.
See `LambdaCalculusBigStep/star_cbv_bigcbv`. □

An alternative to naïve substitution

A basic need is to record that x is bound to v while evaluating a term t.

So far, we have used an eager substitution, $t[v/x]$, but:

- This is inefficient.
- This does not respect the separation between immutable code and mutable data imposed by current hardware and operating systems.

Idea: instead of applying the substitution [v/x] to the code,
record the binding $x \mapsto v$ in a data structure, known as an environment.

An environment is a finite map of variables to values.

A first attempt

Let us try and define a new big-step evaluation judgement, $e \vdash t \downarrow_{\text{cbv}} v$.

(previous definition)

$$\textsc{BigCbvValue}\quad \frac{}{v \downarrow_{\text{cbv}} v}$$

$$\textsc{BigCbvApp}\quad \frac{t_1 \downarrow_{\text{cbv}} \lambda x.u_1 \qquad t_2 \downarrow_{\text{cbv}} v_2 \qquad u_1[v_2/x] \downarrow_{\text{cbv}} v}{t_1\ t_2 \downarrow_{\text{cbv}} v}$$

(attempt at a new definition)

$$\textsc{EBigCbvVar}\quad \frac{e(x) = v}{e \vdash x \downarrow_{\text{cbv}} v}$$

$$\textsc{EBigCbvLam}\quad \frac{}{e \vdash \lambda x.t \downarrow_{\text{cbv}} \lambda x.t}$$

$$\textsc{EBigCbvApp}\quad \frac{e \vdash t_1 \downarrow_{\text{cbv}} \lambda x.u_1 \qquad e \vdash t_2 \downarrow_{\text{cbv}} v_2 \qquad e[x \mapsto v_2] \vdash u_1 \downarrow_{\text{cbv}} v}{e \vdash t_1\ t_2 \downarrow_{\text{cbv}} v}$$

What is wrong with this definition?

Lexical scoping versus dynamic scoping

What value should the following OCaml code produce?

```
let x = 42 in
let f = fun () -> x in
let x = "oops" in
f()
```

Lexical scoping versus dynamic scoping

What value should the following OCaml code produce?

```
let x = 42 in
let f = fun () -> x in
let x = "oops" in
f()
```

Well,

- The answer is `42`. This is lexical scoping. This is λ-calculus.
- The answer is not `"oops"`. That would be dynamic scoping.

Lexical scoping versus dynamic scoping

What value should the following OCaml code produce?

```
let x = 42 in
let f = fun () -> x in
let x = "oops" in
f()
```

Well,

- The answer is 42. This is lexical scoping. This is λ-calculus.
- The answer is not `"oops"`. That would be dynamic scoping.

Thus, the free variables of a λ-abstraction must be evaluated:

- in the environment that exists at the function's creation site,
- not in the environment that exists at the function's call site.

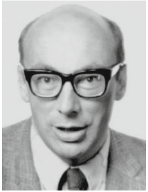
A failed attempt

Thus, our first attempt is wrong:

- It implements dynamic scoping instead of lexical scoping.
- If $e \vdash t \downarrow_{\text{cbv}} v$ and $fv(t) \subseteq dom(e)$ then we would expect that v is closed and $t[e] \downarrow_{\text{cbv}} v$ holds — but that is not the case.
- The candidate rule EBigCbvLam obviously violates this property. It fails to record the environment that exists at function creation time.

How can we fix the problem?

Closures



The result of evaluating a λ-abstraction $\lambda x.t$, where $fv(\lambda x.t)$ may be nonempty, should not be $\lambda x.t$.

It should be a closure $\langle \lambda x.t \mid e \rangle$,

- that is, a pair of a λ-abstraction and an environment,
- in other words, a pair of a code pointer and a pointer to a heap-allocated data structure.

Landin, The Mechanical Evaluation of Expressions, 1964.

Closures and environments

The abstract syntax of closures is:

$$c ::= \langle \lambda x.t \mid e \rangle$$

We expect the evaluation of a term to produce a closure:

$$e \vdash t \downarrow_{\text{cbv}} c$$

Because evaluating x produces $e(x)$,
an environment must be a finite map of variables to closures:

$$e ::= [] \mid e[x \mapsto c]$$

Thus, the syntaxes of closures and environments are mutually inductive.

A big-step semantics with environments

Evaluating a λ-abstraction produces a newly allocated closure.

$$\frac{\textsc{EBigCbvVar} \quad e(x) = c}{e \vdash x \downarrow_{\text{cbv}} c}
\qquad
\frac{\textsc{EBigCbvLam} \quad fv(\lambda x.t) \subseteq dom(e)}{e \vdash \lambda x.t \downarrow_{\text{cbv}} \langle \lambda x.t \mid e \rangle}$$

$$\frac{\begin{array}{c}\textsc{EBigCbvApp} \\ e \vdash t_1 \downarrow_{\text{cbv}} \langle \lambda x.u_1 \mid e' \rangle \\ e \vdash t_2 \downarrow_{\text{cbv}} c_2 \\ e'[x \mapsto c_2] \vdash u_1 \downarrow_{\text{cbv}} c\end{array}}{e \vdash t_1\ t_2 \downarrow_{\text{cbv}} c}$$

Invoking a closure causes the closure's code to be evaluated in the closure's environment, extended with a binding of formal to actual.

Equivalence between big-step semantics without and with environments

How can we relate the judgements $t \downarrow_{\mathrm{cbv}} v$ and $e \vdash t \downarrow_{\mathrm{cbv}} c$?

What lemma should we state?

Equivalence between big-step semantics without and with environments

How can we relate the judgements $t \downarrow_{\text{cbv}} v$ and $e \vdash t \downarrow_{\text{cbv}} c$?

What lemma should we state?

Assuming t is closed, we would like to prove that

$$t \downarrow_{\text{cbv}} v$$

holds if and only if

Equivalence between big-step semantics without and with environments

How can we relate the judgements $t \downarrow_{\text{cbv}} v$ and $e \vdash t \downarrow_{\text{cbv}} c$?

What lemma should we state?

Assuming t is closed, we would like to prove that

$$t \downarrow_{\text{cbv}} v$$

holds if and only if

$$[] \vdash t \downarrow_{\text{cbv}} c$$

holds for some closure c such that c represents v in a certain sense.

Decoding closures

c represents *v* can be defined as $\lceil c \rceil = v$, where $\lceil c \rceil$ is defined by:

$$\lceil \langle \lambda x.t \mid e \rangle \rceil \quad = \quad (\lambda x.t)[\lceil e \rceil]$$

and where the substitution $\lceil e \rceil$ maps every variable x in $dom(e)$ to $\lceil e(x) \rceil$.

($\lceil c \rceil$ and $\lceil e \rceil$ are mutually inductively defined.)

Equivalence between big-step semantics without and with environments

One implication is easily established:

Lemma (Soundness of the environment semantics)

$e \vdash t \downarrow_{cbv} c$ implies $t[\lceil e \rceil] \downarrow_{cbv} \lceil c \rceil$.

Proof (Sketch).

By induction on the hypothesis.
See `LambdaCalculusBigStep/ebigcbv_bigcbv`. □

In particular, $[] \vdash t \downarrow_{\text{cbv}} c$ implies $t \downarrow_{\text{cbv}} \lceil c \rceil$.

Equivalence between big-step semantics without and with environments

The reverse implication requires a more complex statement:

Lemma (Completeness of the environment semantics)

If $t[\lceil e \rceil] \downarrow_{cbv} v$, where $fv(t) \subseteq dom(e)$ and e is well-formed, then there exists c such that $e \vdash t \downarrow_{cbv} c$ and $\lceil c \rceil = v$.

Proof (Sketch).

By induction on the first hypothesis and by case analysis on t.
See `LambdaCalculusBigStep/bigcbv_ebigcbv`. □

In particular, if t is closed, then $t \downarrow_{\text{cbv}} v$ implies $[] \vdash t \downarrow_{\text{cbv}} c$, for some closure c such that $\lceil c \rceil = v$.

Equivalence between big-step semantics without and with environments

The notion of well-formedness on the previous slide is inductively defined:

$$\frac{fv(\lambda x.t) \subseteq dom(e) \qquad e \text{ is well-formed}}{\langle \lambda x.t \mid e \rangle \text{ is well-formed}} \qquad \frac{\forall x, x \in dom(e) \Rightarrow e(x) \text{ is well-formed}}{e \text{ is well-formed}}$$

Lemma (Well-formedness is an invariant)

If $e \vdash t \downarrow_{cbv} c$ holds and e is well-formed, then c is well-formed.

Proof.

See `LambdaCalculusBigStep/ebigcbv_wf_cvalue`. □

This property is exploited in the proof of the previous lemma.

From big-step semantics to interpreter, again

The big-step semantics $e \vdash t \downarrow_{\text{cbv}} c$ is a 3-place relation.

We now wish to define a (partial) function of two arguments e and t.

We could do this in OCaml, as we did earlier today.

Let us do it in Coq and prove this interpreter correct and complete!

See `LambdaCalculusInterpreter`.

Syntax

The syntax of terms (in de Bruijn's representation) is as before.

The syntax of closures and environments is as shown earlier:

```
Inductive cvalue :=
| Clo: {bind term} -> list cvalue -> cvalue.

Definition cenv :=
  list cvalue.
```

A first attempt

```
Fail Fixpoint interpret (e : cenv) (t : term) : cvalue :=
  match t with
  | Var x =>
      nth x e dummy_cvalue
        (* dummy is used when x is out of range *)
  | Lam t =>
      Clo t e
  | App t1 t2 =>
      let cv1 := interpret e t1 in
      let cv2 := interpret e t2 in
      match cv1 with Clo u1 e' =>
        interpret (cv2 :: e') u1
      end
  end.
```

Why is this definition rejected by Coq?

A standard trick: fuel

We parameterize the interpreter with a maximum recursive call depth *n*.

```
Fixpoint interpret (n : nat) e t : option cvalue :=
  match n with
  | 0   => None (* not enough fuel *)
  | S n =>
      match t with
      | Var x      => Some (nth x e dummy_cvalue)
      | Lam t      => Some (Clo t e)
      | App t1 t2 =>
          interpret n e t1 >>= fun cv1 =>
          interpret n e t2 >>= fun cv2 =>
          match cv1 with Clo u1 e' =>
            interpret n (cv2 :: e') u1
          end
  end end.
```

The interpreter can now fail, therefore has return type `option cvalue`.

Equivalence between the big-step semantics and the interpreter

If the interpreter produces a result, then it is a correct result.

Lemma (Soundness of the interpreter)

If interpret n e t = Some c and fv(t) ⊆ dom(e) and e is well-formed then $e \vdash t \downarrow_{cbv} c$ *holds.*

Proof (Sketch).

By induction on *n*, by case analysis on *t*, and by inspection of the first hypothesis. See `LambdaCalculusInterpreter/interpret_ebigcbv`. □

An interpreter that always returns *None* would satisfy this lemma, hence the need for a completeness statement...

Equivalence between the big-step semantics and the interpreter

If the evaluation of t is supposed to produce c, then, given sufficient fuel, the interpreter returns c.

Lemma (Completeness of the interpreter)

If $e \vdash t \downarrow_{cbv} c$, then there exists n such that $interpret\ n\ e\ t = Some\ c$.

Proof (Sketch).

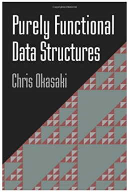
By induction on the hypothesis, exploiting the fact that *interpret* is monotonic in n, that is, $n_1 \leq n_2$ implies $interpret\ n_1\ e\ t \preceq interpret\ n_2\ e\ t$, where the "definedness" partial order $\preceq$ is generated by $None \preceq Some\ c$. See `LambdaCalculusInterpreter/ebigcbv_interpret`. □

Summary

If t is closed and v is a value, then the following are equivalent:

$t \longrightarrow^{\star}_{\text{cbv}} v$	small-step substitution semantics
$t \downarrow_{\text{cbv}} v$	big-step substitution semantics
$\exists c \begin{cases} [] \vdash t \downarrow_{\text{cbv}} c \\ \lceil c \rceil = v \end{cases}$	big-step environment semantics
$\exists c \exists n \begin{cases} \mathit{interpret}\ n\ []\ t = \mathit{Some}\ c \\ \lceil c \rceil = v \end{cases}$	interpreter

Complexity and cost model



For simplicity, we have represented environments as lists.

Thus, extension has complexity $O(1)$, but lookup has complexity $O(n)$, where n is the number of variables in scope.

For greater efficiency, one should use a data structure that allows both operations in time $O(\log n)$, such as Okasaki's random access lists.

When “extracting” the interpreter from Coq to OCaml, one should also instruct Coq to represent natural numbers as OCaml machine integers.

Okasaki, Purely functional data structures, 1996 (§6.4.1).

Complexity and cost model

With these changes, the interpreter is reasonably efficient.

For time, it offers a relatively clear cost model:

- Evaluating a variable costs $O(\log n)$.
- Evaluating a λ-abstraction costs $O(1)$.
- Evaluating an application costs $O(\log n)$.

n is the maximum number of variables in scope and could be considered $O(1)$, as it depends only on the program's text, not on the input data.

Caveat: the cost of garbage collection is not accounted for in this model.

Digression: the cost of garbage collection

Let H be the total heap size.

Let R be the total size of the live objects. Thus, $R \leq H$.

Assuming a copying collector, one collection costs $O(R)$.

Collection takes place when the heap is full, so frees up $H - R$ words.

Thus, the amortized cost of collection, per freed-up word, is

$$\frac{O(R)}{H - R}$$

Under the hypothesis $\frac{R}{H} \leq \frac{1}{2}$, this cost is $O(1)$. That is,

> *Provided the heap is large enough,*
> *freeing up an object takes constant (amortized) time.*

Full closures versus minimal closures

In reality, this interpreter has one subtle but serious inefficiency.

When a closure $\langle \lambda x.t \mid e \rangle$ is allocated,
the entire environment e is stored in it,
even though $fv(\lambda x.t)$ may be a strict subset of the domain of e.

We store data that the closure will never need. This is a space leak!

To fix this, one should store a trimmed-down environment in the closure.

Exercise: state and prove that, if x does not occur free in t, then the evaluation of t in an environment e does not depend on the value $e(x)$.

Exercise: define an optimized interpreter where, at a closure allocation, every unneeded value in e is replaced with a dummy value. Prove it equivalent to the simpler interpreter.

Syntactic sugar

Some constructs may be be viewed as syntactic sugar, that is, compiled away by macro-expansion.

E.g., "let $x = t_1$ in t_2" can be viewed as sugar for "$(\lambda x.t_2)\ t_1$".

This yields the desired semantics. The following are lemmas:

$$\textsc{LetV}\quad \frac{}{\text{let } x = v \text{ in } t \longrightarrow_{\text{cbv}} t[v/x]} \qquad \textsc{LetL}\quad \frac{t \longrightarrow_{\text{cbv}} t'}{\text{let } x = t \text{ in } u \longrightarrow_{\text{cbv}} \text{let } x = t' \text{ in } u}$$

One may prefer to view "let $x = t_1$ in t_2" as a primitive construct if there is:

- a special typing rule for it, e.g., in ML;
- a special compilation rule for it, e.g., in the CPS transform.
- a restriction of applications to the form "$v\ v$", so "let" is the only sequencing construct.

Products

It is easy to add pairs and projections to the (call-by-value) λ-calculus.

$$
\begin{array}{rcll}
t & ::= & \ldots \mid (t, t) \mid \pi_i\ t & \quad \text{where } i \in \{0, 1\} \\
v & ::= & \ldots \mid (v, v) & \\
E & ::= & \ldots \mid (E, t) \mid (v, E) \mid \pi_i\ E &
\end{array}
$$

One new reduction rule is needed:

$$
\textsc{Proj}\quad \frac{}{\pi_i\ (v_0, v_1) \longrightarrow_{\text{cbv}} v_i}
$$

Exercise: Extend the call-by-name λ-calculus with pairs and projections.

Exercise: Propose a definition of pairs and projections as sugar in the call-by-value λ-calculus. Check that this yields the desired semantics.

Sums

One similarly adds injections and case analysis to CBV λ-calculus.

$$
\begin{array}{rcll}
t & ::= & \ldots \mid \mathsf{inj}_i\ t \mid \mathsf{case}\ t\ \mathsf{of}\ x.t \parallel x.t & \quad \text{where } i \in \{0, 1\} \\
v & ::= & \ldots \mid \mathsf{inj}_i\ v & \\
E & ::= & \ldots \mid \mathsf{inj}_i\ E \mid \mathsf{case}\ E\ \mathsf{of}\ x.t \parallel x.t &
\end{array}
$$

One new reduction rule is needed:

$$
\textsc{Case}\quad \frac{}{\mathsf{case}\ \mathsf{inj}_i\ v\ \mathsf{of}\ x_0.t_0 \parallel x_1.t_1 \longrightarrow_{\mathsf{cbv}} t_i[v/x_i]}
$$

Exercise: Extend the call-by-name λ-calculus with sums.

Recursive functions

The construct $\lambda x.t$ is replaced with $\mu f.\lambda x.t$.

$$\begin{array}{rcl} t & ::= & \ldots \mid \mu f.\lambda x.t \\ v & ::= & \ldots \mid \mu f.\lambda x.t \end{array}$$

$\lambda x.t$ is sugar for $\mu\_.\lambda x.t$.

"let rec $f\ x = t$ in u" is sugar for "let $f = \mu f.\lambda x.t$ in u".

The β-reduction rule is amended as follows:

$$\beta_v \quad \frac{}{(\mu f.\lambda x.t)\ v \longrightarrow_{\text{cbv}} t[v/x][\mu f.\lambda x.t/f]}$$

A few things to remember

An efficient interpreter uses environments and closures, not substitutions.

- It can (easily) be proved correct and complete!

There are several styles of (operational) semantics.

- They can (easily) be proved equivalent!

For denotational semantics, see:

Benton, Birkedal, Kennedy, Varming, Formalizing domains, ultrametric spaces and semantics of programming languages, 2010.

Dockins, Formalized, Effective Domain Theory in Coq, 2014.

A few things to remember

Machine-checked proofs are hard when your definitions are too complex, your statements are wrong, and you are missing key lemmas and tactics.

By which I mean, of course,

A few things to remember

Machine-checked proofs are hard when your definitions are too complex, your statements are wrong, and you are missing key lemmas and tactics.

By which I mean, of course, that machine-checking helps (forces) you to

- get definitions right,
- write precise statements,
- develop high-level lemmas and tactics.

“But as for you, be strong and do not give up, for your work will be rewarded.”