

MPRI 2.4

From operational semantics to interpreters

François Pottier



2023

The λ -calculus

The formal model that underlies all functional programming languages.

Abstract syntax:

$$t, u ::= x \mid \lambda x. t \mid t \ t \quad (\text{terms})$$

Reduction:

$$(\lambda x. t) \ u \longrightarrow t[u/x] \quad (\beta)$$

Mnemonic: read $t[u/x]$ as “ t , where u is substituted for x ”.

Landin, [Correspondence betw. ALGOL 60 and Church's \$\lambda\$ -notation](#), 1965.

“It seems possible that the correspondence might form the basis of a formal description of the semantics of Algo 60.”

From the λ -calculus to a functional programming language

Start from the λ -calculus, and follow several steps:

- Fix a **reduction strategy** (today).
- Develop **efficient execution mechanisms** (today).
- **Enrich the language** with primitive data types and operations, recursion, algebraic data structures, and so on.
- Define a static **type system** (next week).

Landin, **The next 700 programming languages**, 1966.

“Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things.”

Operational semantics

An operational semantics describes **the actions of a machine**, in the simplest possible manner / at the most abstract level.

Plotkin, **A Structural Approach to Operational Semantics**, 1981, (2004).

Plotkin, **The Origins of Structural Operational Semantics**, 2004.

Plotkin: — *It is only through having an operational semantics that the $[\lambda\text{-calculus can}]$ be viewed as a programming language.*

Scott: — *Why call it operational semantics? What is operational about it?*

Denotational semantics

Scott preferred **denotational** semantics, where the meaning of a program is a mathematical function of an input to an output.

Benton, Kennedy, Varming,

Some Domain Theory and Denotational Semantics in Coq, 2009.

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

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

The call-by-value strategy

Values form a subset of terms:

$$\begin{array}{ll} t, u & ::= x \mid \lambda x. t \mid t \ t \quad (\text{terms}) \\ v & ::= x \mid \lambda x. t \quad (\text{values}) \end{array}$$

A value represents the **result** of a computation.

The **call-by-value** reduction relation $t \longrightarrow_{\text{cbv}} t'$ is inductively defined:

$$\begin{array}{c} \beta_v \\ \hline (\lambda x. t) \ v \longrightarrow_{\text{cbv}} t[v/x] \end{array} \qquad \begin{array}{c} \text{APPL} \\ t \longrightarrow_{\text{cbv}} t' \\ \hline t \ u \longrightarrow_{\text{cbv}} t' \ u \end{array} \qquad \begin{array}{c} \text{APPVR} \\ u \longrightarrow_{\text{cbv}} u' \\ \hline v \ u \longrightarrow_{\text{cbv}} v \ u' \end{array}$$

This is known as a **small-step** operational semantics.

Example

This is a proof (a.k.a. derivation) that **one** reduction step is permitted:

$$\begin{array}{c}
 \dfrac{x[1/x] = 1}{(\lambda x.x) \ 1 \longrightarrow_{\text{cbv}} 1} \beta_v \\
 \dfrac{(\lambda x.\lambda y.y \ x) \ ((\lambda x.x) \ 1) \longrightarrow_{\text{cbv}} (\lambda x.\lambda y.y \ x) \ 1}{(\lambda x.\lambda y.y \ x) \ ((\lambda x.x) \ 1) \longrightarrow_{\text{cbv}} (\lambda x.\lambda y.y \ x) \ 1} \text{APP R} \\
 \dfrac{(\lambda x.\lambda y.y \ x) \ ((\lambda x.x) \ 1) \longrightarrow_{\text{cbv}} (\lambda x.\lambda y.y \ x) \ 1 \ (\lambda x.x)}{(\lambda x.\lambda y.y \ x) \ ((\lambda x.x) \ 1) \longrightarrow_{\text{cbv}} (\lambda x.\lambda y.y \ x) \ 1 \ (\lambda x.x)} \text{APP L}
 \end{array}$$

Features of call-by-value reduction

- **Weak reduction.** One cannot reduce under a λ -abstraction.

$$\frac{t \longrightarrow_{\text{cbv}} t'}{\lambda x. t \longrightarrow_{\text{cbv}} \lambda x. t'}$$

Thus, **values do not reduce.**

Also, we are interested in reducing **closed terms** only.

- **Call-by-value.** An actual argument is reduced to a value **before** it is passed to a function.

$$(\lambda x. t) \text{ } v \longrightarrow_{\text{cbv}} t[v/x]$$

$$(\lambda x. t) (u_1 \ u_2) \longrightarrow_{\text{cbv}} t[u_1 \ u_2/x]$$

Features of call-by-value reduction

- **Left-to-right.** In an application $t\ u$, the term t must be reduced to a value before u can be reduced at all.

$$\text{AppVR} \quad \frac{u \longrightarrow_{\text{cbv}} u'}{V\ u \longrightarrow_{\text{cbv}} V\ u'}$$

- **Determinism.** For every term t , there is at most one term t' such that $t \longrightarrow_{\text{cbv}} t'$ holds.

Reduction sequences

Sequences of reduction steps describe the behavior of a term.

The following three situations are mutually exclusive:

- **Termination:** $t \longrightarrow_{\text{cbv}} t_1 \longrightarrow_{\text{cbv}} t_2 \longrightarrow_{\text{cbv}} \dots \longrightarrow_{\text{cbv}} v$
The value v is the result of evaluating t .
The term t **converges** to v .
- **Divergence:** $t \longrightarrow_{\text{cbv}} t_1 \longrightarrow_{\text{cbv}} t_2 \longrightarrow_{\text{cbv}} \dots \longrightarrow_{\text{cbv}} t_n \longrightarrow_{\text{cbv}} \dots$
The sequence of reductions is infinite.
The term t **diverges**.
- **Error:** $t \longrightarrow_{\text{cbv}} t_1 \longrightarrow_{\text{cbv}} t_2 \longrightarrow_{\text{cbv}} \dots \longrightarrow_{\text{cbv}} t_n \not\longrightarrow_{\text{cbv}} \cdot$
where t_n is not a value, yet does not reduce: t_n is **stuck**.
The term t **goes wrong**. This is a **runtime error**.

A strong **type system** rules out errors (Milner, 1978).

Some type systems rule out both errors and divergence.

Examples of reduction sequences

Termination:

$$\begin{aligned}
 (\lambda x. \lambda y. y \ x) ((\lambda x. x) \ 1) (\lambda x. x) &\longrightarrow_{\text{cbv}} (\lambda x. \lambda y. y \ x) \ 1 (\lambda x. x) \\
 &\longrightarrow_{\text{cbv}} (\lambda y. y \ 1) (\lambda x. x) \\
 &\longrightarrow_{\text{cbv}} (\lambda x. x) \ 1 \\
 &\longrightarrow_{\text{cbv}} 1
 \end{aligned}$$

Divergence:

$$(\lambda x. x \ x) (\lambda x. x \ x) \longrightarrow_{\text{cbv}} (\lambda x. x \ x) (\lambda x. x \ x) \longrightarrow_{\text{cbv}} \dots$$

Error:

$$(\lambda x. x \ x) \ 2 \longrightarrow_{\text{cbv}} 2 \ 2 \not\longrightarrow_{\text{cbv}} \cdot$$

The active redex is highlighted in red.

An alternative style: evaluation contexts

First, define **head reduction**:

$$\frac{\beta_v}{(\lambda x.t) \ v \longrightarrow_{\text{cbv}}^{\text{head}} t[v/x]}$$

Then, define **reduction** as head reduction under an evaluation context:

$$\frac{\text{Ctx} \quad t \longrightarrow_{\text{cbv}}^{\text{head}} t'}{E[t] \longrightarrow_{\text{cbv}} E[t']}$$

where evaluation contexts E are defined by $E ::= [] \mid E \ u \mid v \ E$.

Wright and Felleisen, **A syntactic approach to type soundness**, 1992.

Unique decomposition

In this alternative style, the determinism of the reduction relation follows from a **unique decomposition** lemma:

Lemma (Unique Decomposition)

For every term t , there exists at most one pair (E, u) such that

- $t = E[u]$
- $\exists u' \quad u \longrightarrow_{cbv}^{head} u'.$

The call-by-name strategy

The **call-by-name** reduction relation $t \longrightarrow_{\text{cbn}} t'$ is defined as follows:

$$\frac{\beta}{(\lambda x.t) \ u \longrightarrow_{\text{cbn}} t[u/x]} \qquad \frac{\text{APPL} \quad t \longrightarrow_{\text{cbn}} t'}{t \ u \longrightarrow_{\text{cbn}} t' \ u}$$

The **unevaluated** actual argument is passed to the function.

It is later reduced if / when / every time the function **demands** its value.

An example reduction sequence

$$\begin{array}{ll}
 (\lambda x. \lambda y. y \ x) ((\lambda x. x) \ 1) (\lambda x. x) & \longrightarrow_{\text{cbn}} (\lambda y. y \ ((\lambda x. x) \ 1)) (\lambda x. x) \\
 & \longrightarrow_{\text{cbn}} (\lambda x. x) ((\lambda x. x) \ 1) \\
 & \longrightarrow_{\text{cbn}} (\lambda x. x) \ 1 \\
 & \longrightarrow_{\text{cbn}} 1
 \end{array}$$

Call-by-value versus call-by-name

If t terminates under CBV, then it also terminates under CBN (*).

The converse is **false**:

$$\begin{aligned} (\lambda x. 1) \omega &\longrightarrow_{\text{cbn}} 1 \\ (\lambda x. 1) \omega &\longrightarrow_{\text{cbv}}^{\infty} \end{aligned}$$

where $\omega = (\lambda x. x \ x) (\lambda x. x \ x)$ diverges under both strategies.

Call-by-value can perform fewer reduction steps:

$(\lambda x. x + x) \ t$ evaluates t once under CBV, **twice** under CBN.

Call-by-name can perform fewer reduction steps:

$(\lambda x. 1) \ t$ evaluates t once under CBV, **not at all** under CBN.

(*) In fact, the **standardization** theorem implies that if t can be reduced to a value via any strategy, then it can be reduced to a value via CBN.

See **Takahashi (1995)**.

Encoding call-by-name in a CBV language

Use **thunks**: functions λ_u whose purpose is to delay the evaluation of u .

$$\begin{aligned} \llbracket x \rrbracket &= x () \\ \llbracket \lambda x. t \rrbracket &= \lambda x. \llbracket t \rrbracket \\ \llbracket t \ u \rrbracket &= \llbracket t \rrbracket (\lambda_ \llbracket u \rrbracket) \end{aligned}$$

Exercise: Can you **state** that this encoding is correct? Can you **prove** it?
— 2017 exam! (**paper assignment and solution**) (**Coq solution**)

Encoding call-by-name in a CBV language

In a simply-typed setting, this transformation is **type-preserving**: that is,

$$\Gamma \vdash t : T \quad \text{implies} \quad \llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket : \llbracket T \rrbracket.$$

The translation of types is defined by

$$\llbracket T_1 \rightarrow T_2 \rrbracket = \text{thunk } \llbracket T_1 \rrbracket \rightarrow \llbracket T_2 \rrbracket$$

where $\text{thunk } T$ is $\text{unit} \rightarrow T$.

The translation of type environments is as follows:

$\llbracket x_1 : T_1; \dots; x_n : T_n \rrbracket$ stands for $x_1 : \text{thunk } \llbracket T_1 \rrbracket; \dots; x_n : \text{thunk } \llbracket T_n \rrbracket$.

Encoding call-by-value in a CBN language

The reverse encoding is somewhat more involved.

The call-by-value **continuation-passing style** (CPS) transformation, studied later on in this course, achieves such an encoding.

Call-by-push-value

Levy: — *The existence of two separate paradigms is troubling.*

Levy proposes **call-by-push-value**,
a lower-level calculus into which both CBV and CBN can be encoded,
thus avoiding a certain amount of duplication between their theories.

Levy, **Call-by-Push-Value: A Subsuming Paradigm**, 1999.

Forster et al., **Call-By-Push-Value in Coq:
Operational, Equational, and Denotational Theory**, 2018.

Call-by-need

Call-by-need, a.k.a. **lazy evaluation**, eliminates the main inefficiency of call-by-name (namely, repeated computation) by introducing **memoization**.

Its description via an operational semantics involves:

- either **mutable state** and **sharing** ([Ariola and Felleisen, 1997](#); [Maraist, Odersky, Wadler, 1998](#));
- or **nondeterminism**: “call-by-need is clairvoyant call-by-value” ([Hackett and Hutton, 2019](#)).

It is used in Haskell, where it encourages a **modular style** of programming.

Hughes, [Why functional programming matters](#), 1990.

Also see [Harper's](#) and [Augustsson's](#) blog posts on laziness.

Newton-Raphson iteration (after Hughes)

This is pseudo-Haskell code. The colon `:` is “cons”.

An approximation of the square root of `n` can be computed as follows:

```
next n x = (x + n / x) / 2
repeat f a = a : (repeat f (f a))
within eps (a : b : rest) =
  if abs (a - b) <= eps then b
  else within eps (b : rest)
sqrt a0 eps n =
  within eps (repeat (next n) a0)
```

`repeat (next n) a0` is a **producer** of an infinite stream of numbers.

Its type is just “list of numbers” – look Ma, **no iterators** à la Java!

The **consumer** `within eps` decides how many elements to demand.

The two are programmed **independently**.

Encoding call-by-need in a CBV language

Call-by-need can be encoded into CBV by using **memoizing thunks**:

$$\begin{aligned} \llbracket x \rrbracket &= \text{force } x \\ \llbracket \lambda x. t \rrbracket &= \lambda x. \llbracket t \rrbracket \\ \llbracket t \ u \rrbracket &= \llbracket t \rrbracket (\text{suspend } (\lambda_. \llbracket u \rrbracket)) \end{aligned}$$

Such a thunk evaluates u when **first** forced,
then memoizes the result,
so no computation is required if the thunk is forced **again**.

Thunks can be thought of as an abstract type with this API or signature:

```
type 'a thunk
val suspend: (unit -> 'a) -> 'a thunk
val force: 'a thunk -> 'a
```

Encoding call-by-need in a CBV language

Exercise: implement the thunk API in OCaml. ([Solution.](#))

In reality, this exercise is unnecessary, as OCaml has built-in thunks:

- “suspend (λ_u) ” is written `lazy u`.
- “force x ” is written `Lazy.force x`.

Exercise: port Newton-Raphson iteration to OCaml.
Make sure that **each element is computed at most once**
and **no more elements than necessary** are computed.
Write tests to verify these properties. ([Solution.](#))

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 $E[\lambda(x.u) \ v]$.
- Perform the substitution $u[v/x]$.
- Construct the term $E[u[v/x]]$.

The time required to do this is **not** $O(1)$. Why?

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:

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

where $t_1 \longrightarrow_{\text{cbv}}^* \lambda x. u_1$ and $t_2 \longrightarrow_{\text{cbv}}^* v_2$. 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:

$$\begin{array}{c}
 \text{BIGCBVVALUE} \\
 \hline
 v \downarrow_{\text{cbv}} v
 \end{array}
 \qquad
 \begin{array}{c}
 \text{BIGCBVAPP} \\
 \hline
 \frac{t_1 \downarrow_{\text{cbv}} \lambda x. u_1 \quad t_2 \downarrow_{\text{cbv}} v_2 \quad u_1[v_2/x] \downarrow_{\text{cbv}} v}{t_1 \ t_2 \downarrow_{\text{cbv}} v}
 \end{array}$$

Exercise: define \downarrow_{cbn} .

Example

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

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

Whereas a proof of $t \rightarrow_{\text{cbv}} t'$ 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 $(\lambda x) b$ as result, then continue by executing $b(a/x)$.
Thus $c(a)$ has value d if c has value $(\lambda x) b$ 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.

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

Some history

Kahn promotes big-step operational semantics:

$\rho \vdash \text{number } N \Rightarrow N$	(1)
$\rho \vdash \text{true} \Rightarrow \text{true}$	(2)
$\rho \vdash \text{false} \Rightarrow \text{false}$	(3)
$\rho \vdash \lambda P. E \Rightarrow [\lambda P. E, \rho]$	(4)
$\frac{\rho \vdash \text{ident } i \mapsto \alpha}{\rho \vdash \text{ident } i \Rightarrow \alpha}$	(5)
$\frac{\rho \vdash E_1 \Rightarrow \text{true} \quad \rho \vdash E_2 \Rightarrow \alpha}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow \alpha}$	(6)
$\frac{\rho \vdash E_1 \Rightarrow \text{false} \quad \rho \vdash E_3 \Rightarrow \alpha}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow \alpha}$	(7)
$\frac{\rho \vdash E_1 \Rightarrow \alpha \quad \rho \vdash E_2 \Rightarrow \beta}{\rho \vdash (E_1, E_2) \Rightarrow (\alpha, \beta)}$	(8)
$\frac{\rho \vdash E_1 \Rightarrow [\lambda P. E, \rho_1] \quad \rho_1 \vdash E_2 \Rightarrow \alpha \quad \rho_1 \cdot P \mapsto \alpha \vdash E \Rightarrow \beta}{\rho \vdash E_1, E_2 \Rightarrow \beta}$	(9)
$\frac{\rho \vdash E_1 \Rightarrow \alpha \quad \rho \cdot P \mapsto \alpha \vdash E_1 \Rightarrow \beta}{\rho \vdash \text{let } P = E_1 \text{ in } E_1 \Rightarrow \beta}$	(10)
$\frac{\rho \cdot P \mapsto \alpha \vdash E_2 \Rightarrow \alpha \quad \rho \cdot P \mapsto \alpha \vdash E_1 \Rightarrow \beta}{\rho \vdash \text{letrec } P = E_2 \text{ in } E_1 \Rightarrow \beta}$	(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 $t \Downarrow_{cbv} v$ 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 $t \longrightarrow_{cbv}^* v$.

Proof.

By induction on the derivation of $t \downarrow_{cbv} v$.

Case **BIGCBVVALUE**. We have $t = v$. The result is immediate.

Case **BIGCBVAPP**. t is $t_1 \ t_2$, and we have three subderivations:

$$t_1 \downarrow_{cbv} \lambda x. u_1$$

$$t_2 \downarrow_{cbv} v_2$$

$$u_1[v_2/x] \downarrow_{cbv} v$$

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

$$t_1 \longrightarrow_{cbv}^* \lambda x. u_1$$

$$t_2 \longrightarrow_{cbv}^* v_2$$

$$u_1[v_2/x] \longrightarrow_{cbv}^* v$$

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

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

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 $t \longrightarrow_{cbv}^* v$, then $t \downarrow_{cbv} v$.

Proof.

By induction on the first hypothesis, using $v \downarrow_{cbv} v$ 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 (closed) values.

A first attempt

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

(previous definition)

BIGCBVVALUE

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

BIGCBVAPP

$$\frac{\begin{array}{c} t_1 \downarrow_{\text{cbv}} \lambda x. u_1 \\ t_2 \downarrow_{\text{cbv}} v_2 \\ u_1[v_2/x] \downarrow_{\text{cbv}} v \end{array}}{t_1 t_2 \downarrow_{\text{cbv}} v}$$

(attempt at a new definition)

EBIGCBVVAR

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

EBIGCBVLAM

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

EBIGCBVAPP

$$\frac{\begin{array}{c} e \vdash t_1 \downarrow_{\text{cbv}} \lambda x. u_1 \\ e \vdash t_2 \downarrow_{\text{cbv}} v_2 \\ e[x \mapsto v_2] \vdash u_1 \downarrow_{\text{cbv}} v \end{array}}{e \vdash t_1 t_2 \downarrow_{\text{cbv}} v}$$

What is **wrong** with this definition?

In $t \downarrow_{\text{cbv}} v$, both t and v are closed.

In $e \vdash t \downarrow_{\text{cbv}} v$, we expect $\text{fv}(t) \subseteq \text{dom}(e)$. What about v ? Is it closed?
What about the values stored in e ? Are they closed? ...

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 $\text{fv}(t) \subseteq \text{dom}(e)$ 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**.

$$\frac{\text{EBigCbVVar} \quad e(x) = c}{e \vdash x \downarrow_{\text{cbv}} c}$$

$$\frac{\text{EBigCbVLam} \quad fv(\lambda x.t) \subseteq dom(e)}{e \vdash \lambda x.t \downarrow_{\text{cbv}} \langle \lambda x.t \mid e \rangle}$$

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

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

$$\Box \vdash t \Downarrow_{\text{cbv}} c$$

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:

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

and where the substitution $\lceil e \rceil$ maps every variable x in $\text{dom}(e)$ 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 $t[[e]] \Downarrow_{cbv} [c]$.

Proof (Sketch).

By induction on the hypothesis.

See [LambdaCalculusBigStep/ebigcbv_bigcbv](#). □

In particular, $[] \vdash t \Downarrow_{cbv} c$ implies $t \Downarrow_{cbv} [c]$.

Equivalence between big-step semantics without and with environments

The reverse implication requires a more complex statement:

Lemma (Completeness of the environment semantics)

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

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 $t \downarrow_{cbv} v$ implies $\llbracket \cdot \rrbracket \vdash t \downarrow_{cbv} c$,
for some closure c such that $\llbracket c \rrbracket = v$.

Equivalence between big-step semantics without and with environments

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

$$\frac{\begin{array}{l} \text{fv}(\lambda x.t) \subseteq \text{dom}(e) \\ e \text{ is well-formed} \end{array}}{\langle \lambda x.t \mid e \rangle \text{ is well-formed}}$$

$$\frac{\forall x, x \in \text{dom}(e) \Rightarrow e(x) \text{ is well-formed}}{e \text{ is well-formed}}$$

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 envv :=  
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 $\text{interpret } n \ e \ t = \text{Some } c$ and $\text{fv}(t) \subseteq \text{dom}(e)$ 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 $\text{interpret } n \ e \ t = \text{Some } c$.

Proof (Sketch).

By induction on the hypothesis, exploiting the fact that *interpret* is monotonic in n , that is, $n_1 \leq n_2$ implies $\text{interpret } n_1 \ e \ t \leq \text{interpret } n_2 \ e \ t$, where the “definedness” partial order \leq is generated by $\text{None} \leq \text{Some } c$. See [LambdaCalculusInterpreter/ebigcbv_interpret](#). □

Summary

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

$$t \longrightarrow_{\text{cbv}}^* v$$

small-step substitution semantics

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

big-step substitution semantics

$$\exists c \left\{ \begin{array}{l} [] \vdash t \downarrow_{\text{cbv}} c \\ [c] = v \end{array} \right.$$

big-step environment semantics

$$\exists c \exists n \left\{ \begin{array}{l} \text{interpret } n \ [] \ t = \text{Some } c \\ [c] = v \end{array} \right.$$

interpreter

Cost model

We have represented environments as **lists**. Extension costs $O(1)$, but lookup has complexity $O(n)$, where n is the number of variables in scope.

A **better approach** is to represent the environment as an n -tuple. Then,

- evaluating a variable costs $O(1)$;
- evaluating a λ -abstraction costs $O(n)$;
- evaluating a function call costs $O(1)$.

n **can be considered $O(1)$** as it depends only on the program's text, not on the input data.

This **simple cost model** is implemented by the OCaml compiler.

Digression: the cost of garbage collection

The previous slide does not discuss 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 not allowed to become more than half full, 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.

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!