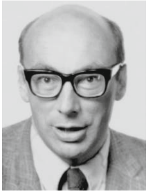
MPRI FUN

From operational semantics to interpreters

François Pottier

2025–2026

The λ-calculus



The formal model that underlies all functional programming languages.

Landin, Correspondence betw. ALGOL 60 and Church's λ-notation, 1965.

"It seems possible that the correspondence might form the basis of a formal description of the semantics of Algol 60."

The λ-calculus

Syntax:

$$\begin{array}{rcll} t, u & ::= & x \mid \lambda x.t \mid t\ t & \text{(terms)} \\ C & ::= & [\,] \mid \lambda x.C \mid C\ t \mid t\ C & \text{(contexts)} \end{array}$$

Think of a context as a term with a hole.

A reduction relation $t \longrightarrow t'$:

$$\begin{array}{rcll} (\lambda x.t)\ u & \longrightarrow & t[u/x] & & (\beta\text{-reduction}) \\ C[t] & \longrightarrow & C[t'] & \text{if } t \longrightarrow t' & \text{(reduction under a context)} \end{array}$$

Read $t[u/x]$ as "t, where u replaces x" or "t with u for x".

Read $C[t]$ as "the context C, where t replaces the hole".

Operational semantics

A reduction relation is also known as a small-step operational semantics.

It describes the actions of a machine at a very abstract level.

One step in the reduction relation corresponds to zero, one, or (usually) many steps of computation in a real machine.

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 [λ-calculus can] be viewed as a programming language.*

Denotational semantics

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

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 Rocq, 2009.

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

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

λ-calculus as a minimal functional programming language

What are the strengths of λ-calculus?

- Its syntax and semantics fit on one slide.
- It is Turing-complete.
- It is declarative.
  - No assignments or jumps.
- It is close to mathematical language.
  - Immutable variables.
  - Functions.
  - Functions as values.

From λ-calculus to a real functional programming language

Isn't there a gap between λ-calculus and real programming languages?

From λ-calculus to a real functional programming language

Isn't there a gap between λ-calculus and real programming languages?

- Its reduction relation is non-deterministic.
  - One would like to fix a reduction strategy.

From λ-calculus to a real functional programming language

Isn't there a gap between λ-calculus and real programming languages?

- Its reduction relation is non-deterministic.
  - One would like to fix a reduction strategy.
- It is not clear at first how to (efficiently) execute λ-terms.
  - One would like develop efficient execution mechanisms, where a separation between code and data is apparent.

From λ-calculus to a real functional programming language

Isn't there a gap between λ-calculus and real programming languages?

- Its reduction relation is non-deterministic.
  - One would like to fix a reduction strategy.
- It is not clear at first how to (efficiently) execute λ-terms.
  - One would like develop efficient execution mechanisms, where a separation between code and data is apparent.
- Pure λ-calculus is minimalistic. Every value is a function.
  - One would like to extend it with primitive data types and operations, algebraic data structures, recursive functions, mutable state, and more.

From λ-calculus to a real functional programming language

Isn't there a gap between λ-calculus and real programming languages?

- Its reduction relation is non-deterministic.
  - One would like to fix a reduction strategy.
- It is not clear at first how to (efficiently) execute λ-terms.
  - One would like develop efficient execution mechanisms, where a separation between code and data is apparent.
- Pure λ-calculus is minimalistic. Every value is a function.
  - One would like to extend it with primitive data types and operations, algebraic data structures, recursive functions, mutable state, and more.
- Pure λ-calculus is untyped. Every value is a function.
  - Once it is enriched with multiple kinds of values, one would like to define a static type system so as to detect many programming mistakes and remove the need for runtime checks.

From λ-calculus to a real functional programming language

Our agenda:

- Fix a reduction strategy, say, call-by-value (today).
- Propose an efficient execution mechanism (today).
- Enrich the language with primitive data, algebraic data (products, sums), recursion, and more (partly covered later in these slides).
- Define a type system (next week and several of the following weeks).

The call-by-value strategy

Values form a subset of terms:

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

A value represents the result of a computation.

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

$$\frac{\beta_v}{(\lambda x.t)\ v \longrightarrow_{cbv} t[v/x]} \qquad \textsc{AppL}\ \frac{t \longrightarrow_{cbv} t'}{t\ u \longrightarrow_{cbv} t'\ u} \qquad \textsc{AppVR}\ \frac{u \longrightarrow_{cbv} u'}{v\ u \longrightarrow_{cbv} v\ u'}$$

Example

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

$$\dfrac{\dfrac{\dfrac{x[1/x] = 1}{(\lambda x.x)\ 1 \longrightarrow_{cbv} 1}\ \beta_v}{(\lambda x.\lambda y.y\ x)\ ((\lambda x.x)\ 1) \longrightarrow_{cbv} (\lambda x.\lambda y.y\ x)\ 1}\ \textsc{AppR}}{(\lambda x.\lambda y.y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x) \longrightarrow_{cbv} (\lambda x.\lambda y.y\ x)\ 1\ (\lambda x.x)}\ \textsc{AppL}$$

Features of call-by-value reduction

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

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

Consequences:

- A function starts running only once it is called.
- A value cannot be reduced.
- The relation $t \longrightarrow_{cbv} t'$ can be considered a relation on closed terms. A term t is closed if it does not have any free variables: $fv(t) = \emptyset$.

Features of call-by-value reduction

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

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

Features of call-by-value reduction

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

$$\textsc{AppVR}\quad \frac{u \longrightarrow_{cbv} u'}{v\ u \longrightarrow_{cbv} v\ u'}$$

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

Reduction sequences

Sequences of reduction steps describe the behavior of a term.

The following three situations are mutually exclusive:

- Termination: $t \longrightarrow_{cbv} t_1 \longrightarrow_{cbv} t_2 \longrightarrow_{cbv} \ldots \longrightarrow_{cbv} v$
  The value v is the result of evaluating t.
  The term t converges to v.
- Divergence: $t \longrightarrow_{cbv} t_1 \longrightarrow_{cbv} t_2 \longrightarrow_{cbv} \ldots \longrightarrow_{cbv} t_n \longrightarrow_{cbv} \ldots$
  The sequence of reductions is infinite.
  The term t diverges.
- Error: $t \longrightarrow_{cbv} t_1 \longrightarrow_{cbv} t_2 \longrightarrow_{cbv} \ldots \longrightarrow_{cbv} t_n \not\longrightarrow_{cbv} \cdot$
  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:

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

Divergence:

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \longrightarrow_{cbv} (\lambda x.x\ x)\ (\lambda x.x\ x) \longrightarrow_{cbv} \ldots$$

Error:

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

The active redex is highlighted in red.

An alternative style: evaluation contexts

AppL and AppVR can be combined as follows:

$$\frac{\beta_v}{(\lambda x.t)\ v \longrightarrow^{head}_{cbv} t[v/x]} \qquad \frac{\textsc{Ctx} \quad t \longrightarrow^{head}_{cbv} t'}{E[t] \longrightarrow_{cbv} E[t']}$$

Head reduction $\longrightarrow^{head}_{cbv}$ allows reduction at the root.

Reduction $\longrightarrow_{cbv}$ allows reduction under an evaluation context E.

Evaluation contexts E are defined by $E ::= [\,] \mid E\ u \mid v\ E$.

This style replaces the many rules that allow evaluation under a context with a single rule.

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

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

One then says that u is the active redex in the term t.

The term t then reduces to $E[u']$ and only to this term.

Exercise: Prove this lemma.

The call-by-name strategy

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

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

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

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

Call-by-value versus call-by-name

If t terminates under CBV, then it also terminates under CBN ($\star$).

The converse is false:

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

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.

($\star$) 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 $\lambda\_.u$ whose purpose is to delay the evaluation of u.

$$
\begin{array}{rcl}
[\![x]\!] & = & x\;() \\
[\![\lambda x.t]\!] & = & \lambda x.[\![t]\!] \\
[\![t\;u]\!] & = & [\![t]\!]\;(\lambda\_.[\![u]\!])
\end{array}
$$

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

Encoding call-by-name in a CBV language

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

Encoding call-by-name in a CBV language

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

$$\Gamma \vdash t : T \quad \text{implies} \quad [\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!].$$

The translation of types is defined by

Encoding call-by-name in a CBV language

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

$$\Gamma \vdash t : T \quad \text{implies} \quad [\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!].$$

The translation of types is defined by

$$[\![T_1 \rightarrow T_2]\!] = \mathit{thunk}\ [\![T_1]\!] \rightarrow [\![T_2]\!]$$

where *thunk* T is $\mathit{unit} \rightarrow T$.

The translation of type environments is as follows:
$[\![x_1 : T_1; \ldots; x_n : T_n]\!]$ stands for $x_1 : \mathit{thunk}\ [\![T_1]\!]; \ldots; x_n : \mathit{thunk}\ [\![T_n]\!]$.

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 Rocq: 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:

$$
\begin{array}{rcl}
[\![x]\!] & = & \mathit{force}\ x \\
[\![\lambda x.t]\!] & = & \lambda x.[\![t]\!] \\
[\![t\ u]\!] & = & [\![t]\!]\ (\mathit{suspend}\ (\lambda\_.[\![u]\!]))
\end{array}
$$

Such a thunk evalutes 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* $(\lambda\_.u)$" 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.

Let us write a naïve interpreter by paraphrasing the small-step semantics.

This interpreter manipulates abstract syntax trees (ASTs).

Let us first spend a little time on this concept.

Abstract syntax trees

What is an abstract syntax tree?

Abstract syntax trees

What is an abstract syntax tree? $t ::= x \mid \lambda x.t \mid t\ t$

Abstract syntax trees

What is an abstract syntax tree? $t ::= x \mid \lambda x.t \mid t\ t$

It is an algebraic data structure—in OCaml, something like this:

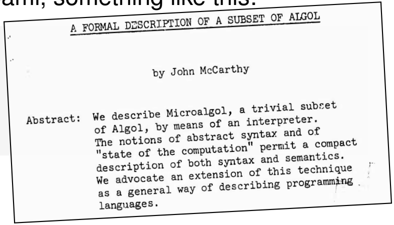
```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

Abstract syntax trees

What is an abstract syntax tree? $t ::= x \mid \lambda x.t \mid t\ t$

It is an algebraic data structure—in OCaml, something like this:

```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

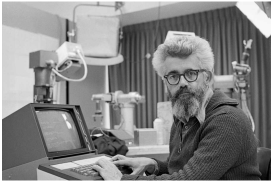


A FORMAL DESCRIPTION OF A SUBSET OF ALGOL

by John McCarthy

Abstract: We describe Microalgol, a trivial subset of Algol, by means of an interpreter. The notions of abstract syntax and of "state of the computation" permit a compact description of both syntax and semantics. We advocate an extension of this technique as a general way of describing programming languages.

McCarthy, A formal description of a subset of Algol, 1964.

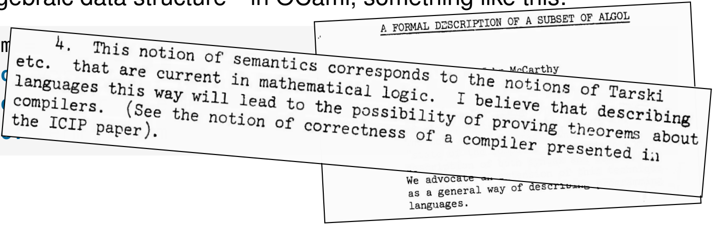


Abstract syntax trees

What is an abstract syntax tree? $t ::= x \mid \lambda x.t \mid t\ t$

It is an algebraic data structure—in OCaml, something like this:

```
type term
  | Var
  | Lam
  | App
```



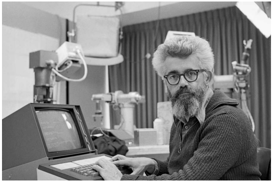
A FORMAL DESCRIPTION OF A SUBSET OF ALGOL

McCarthy

4. This notion of semantics corresponds to the notions of Tarski etc. that are current in mathematical logic. I believe that describing languages this way will lead to the possibility of proving theorems about compilers. (See the notion of correctness of a compiler presented in the ICIP paper).

We advocate ... as a general way of describ... languages.

McCarthy, A formal description of a subset of Algol, 1964.



Abstract syntax trees

What is an abstract syntax tree? $t ::= x \mid \lambda x.t \mid t\ t$

It is an algebraic data structure—in OCaml, something like this:

```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

What do the types `var` and `binder` represent?

Variables and binders

In the term $\lambda x. (x\ y)$,

- the first occurrence of x is a binding occurrence, or binder;
- the second occurrence of x and the single occurrence of y are ordinary occurrences, or variables.

A variable is meant to refer to an earlier binder. In this example,

- the variable x refers to the binder λx; it is bound;
- the variable y refers to no visible binder; it is free; it would become bound if the term $\lambda x. (x\ y)$ was placed in the scope of a binder λy.

Representing variables and binders

In the definition of abstract syntax trees,

```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

how should variables and binders be represented?

That is, how should the types `var` and `binder` be defined?

Representing variables and binders

In the definition of abstract syntax trees,

```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

how should variables and binders be represented?

That is, how should the types `var` and `binder` be defined?

Many answers are possible; see separate slides on this topic.

Here, I present two approaches, nominal style and de Bruijn style.

Nominal-style abstract syntax

In nominal style, both variables and binders are names, as on paper.

Nominal-style abstract syntax

In nominal style, both variables and binders are names, as on paper.

What is a name?

Nominal-style abstract syntax

In nominal style, both variables and binders are names, as on paper.

What is a name? What operations on names are needed?

Nominal-style abstract syntax

In nominal style, both variables and binders are names, as on paper.

What is a name? What operations on names are needed?

- testing the equality of two names;
- generating a fresh name;
- (optional) a total order on names.

Any type that offers these operations can serve as a type of names.

```
type name = int
type var = name
type binder = name
```

See `LambdaNominal.ml`.

Nominal-style abstract syntax

This is the abstract syntax of the λ-calculus in nominal style:

```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

Nominal-style abstract syntax

This is the abstract syntax of the λ-calculus in nominal style:

```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

For example, the "identity" term $\lambda x.x$ can be represented in several ways:

```
let mkid (x : name) : term = Lam (x, Var x)
let id0 : term = mkid 0
let id1 : term = mkid 1
let () = assert (aeq id0 id1)
```

This is a downside of the nominal representation: it is not canonical.

Exercise: Define the function `aeq`, which (efficiently) determines whether two terms are α-equivalent. What is its time complexity?

Substitution

`subst x v t` replaces the variable `x` with the term `v` in the term `t`.

```
let rec subst (x : var) (v : term) (t : term) : term =
  match t with
  | Var y ->
      if y <> x then Var y else v
  | Lam (y, t) ->
      Lam (y, if y = x then t else subst x v t)
  | App (t1, t2) ->
      App (subst x v t1, subst x v t2)
```

Is this code correct?

Substitution

`subst x v t` replaces the variable `x` with the term `v` in the term `t`.

```
let rec subst (x : var) (v : term) (t : term) : term =
  match t with
  | Var y ->
      if y <> x then Var y else v
  | Lam (y, t) ->
      Lam (y, if y = x then t else subst x v t)
  | App (t1, t2) ->
      App (subst x v t1, subst x v t2)
```

Is this code correct? Yes, it is correct provided `v` is closed.

Otherwise, a more complex capture-avoiding substitution is needed.

Exercise: Define a capture-avoiding variant of the function `subst`.

All representations of variables and binders involve renaming variables to avoid collisions. In de Bruijn style, “lift” serves this purpose.

Recognizing values

Let us now come back to our naïve small-step interpreter.

We restrict our attention to closed terms.

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

```
let is_value = function
  | Var _ -> assert false (* we work with closed terms only *)
  | Lam _ -> true
  | App _ -> false
```

Performing one step of reduction

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

```
exception Irreducible
let rec step (t : term) : term =
  match t with
  | Lam _ | Var _ ->
      raise Irreducible
  | App (Lam (x, t), v) when is_value v -> (* Plotkin's BetaV *)
      subst x v t
  | App (t, u) when not (is_value t) ->    (* Plotkin's AppL  *)
      let t' = step t in App (t', u)
  | App (v, u) when is_value v ->          (* Plotkin's AppVR *)
      let u' = step u in App (v, u')
  | App (_, _) ->                     (* All cases covered already *)
      assert false                    (*  but OCaml cannot see it. *)
```

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

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
  | exception Irreducible ->
      t
  | t' ->
      eval t'
```

This is it—the naïve small-step interpreter is complete.

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

de Bruijn-style abstract syntax

In de Bruijn style, a variable is a natural number,
which can be understood as a pointer to an earlier binder.
0 refers to the most recent binder; 1 refers to the next binder; and so on.

```
type var = int (* a de Bruijn index *)
```

A binder carries no information.

```
type binder = unit
```

See `LambdaDeBruijn.ml`.

See also my separate slides on this topic.

de Bruijn-style abstract syntax

This is the abstract syntax of the λ-calculus in de Bruijn style:

```
type term =
  | Var of var
  | Lam of binder * term
  | App of term * term
```

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

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

This representation is canonical. α-equivalence is equality.

de Bruijn-style abstract syntax

This is the abstract syntax of the λ-calculus in de Bruijn style:

```
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)
```

This representation is canonical. α-equivalence is equality.

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 $u \cdot id$.

Capture-avoiding 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
```

The terms in the image of σ need not be closed. (In our use, they are.)

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 the term t moves down into k binders (separate slides).

Performing one step of reduction

The naïve small-step interpreter in de Bruijn style is almost unchanged:

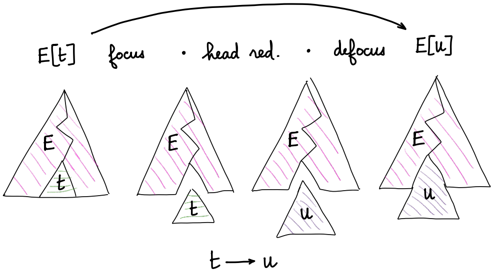
```
exception Irreducible
let rec step (t : term) : term =
  match t with
  | Lam _ | Var _ ->
      raise Irreducible
  | App (Lam t, v) when is_value v ->   (* Plotkin's BetaV *)
      subst (singleton v) t
  | App (t, u) when not (is_value t) -> (* Plotkin's AppL  *)
      let t' = step t in App (t', u)
  | App (v, u) when is_value v ->       (* Plotkin's AppVR *)
      let u' = step u in App (v, u')
  | App (_, _) ->                  (* All cases covered already *)
      assert false                 (*  but OCaml cannot see it. *)
```

Sources of inefficiency

Unfortunately, this small-step interpreter is terribly inefficient.

At each reduction step, one must:

- Focus: decompose the term as $E[t]$ where t is a redex $(\lambda x.t')\ v$.
- Substitute: compute the reduct u, that is, the term $t'[v/x]$.
- Defocus: plug u back into the context E to obtain the term $E[u]$.



Sources of inefficiency

There are 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.

The cost of one function call depends on:

- the depth at which this function call takes place;
- the size of the function that is called.

This is not good—a programmer expects a function call to take time $O(1)$.

Sources of inefficiency

The small-step substitution-based semantics shines by its simplicity.

It can be an asset when reasoning about programs,
but does not suggest an efficient execution scheme.

In the following, we remedy the problem in two stages:

- by moving from small-step to big-step semantics,
  we remove the need to defocus and refocus.
- by moving from substitution-based to environment-based semantics,
  we remove the need to perform 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:

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

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

Evaluate operator; evaluate operand; call; continue execution.

Idea: define a "big-step" relation $t \downarrow_{cbv} v$,
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 $t \downarrow_{cbv} v$ means that evaluating t terminates and produces v.

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

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

Exercise: define $\downarrow_{cbn}$.

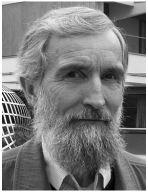
Example

Let us write $\downarrow$ for $\downarrow_{cbv}$, 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 $t \longrightarrow_{cbv} t'$ has linear structure,
a proof of $t \downarrow_{cbv} v$ 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.

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 $t \downarrow_{cbv} v$ holds.

```
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)
      | _      -> assert false (* every value is a function *)
```

If `eval` terminates normally, then it obviously returns a value.
It can also diverge.

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 $t \downarrow_{cbv} v$.
Case BigCbvValue. We have $t = v$. The result is immediate.
Case BigCbvApp. t is $t_1\ t_2$, and we have three subderivations:

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

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

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

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

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

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 $v \downarrow_{cbv} v$ in the base case and the above lemma in the inductive case.
See `LambdaCalculusBigStep/star_cbv_bigcbv`. □

Limitations of big-step semantics

The judgement $t \downarrow_{cbv} v$ describes a terminating computation.

This judgement does not allow saying that "t diverges" or "t crashes".

One can define these two extra judgements in big-step style,
but this requires many rules and seems intuitively redundant.

Charguéraud, Pretty-Big-Step Semantics, 2012.

Dagnino, A meta-theory for big-step semantics, 2022.

Charguéraud, Chlipala, Erbsen, Gruetter,
Omnisemantics: Smooth Handling of Nondeterminism, 2023.

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, $e \vdash t \downarrow_{cbv} v$.

(previous definition)

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

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

(attempt at a new definition)

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

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

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

What is wrong with this definition?

A first attempt

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

(previous definition)

BigCbvValue

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

BigCbvApp

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

(attempt at a new definition)

EBigCbvVar

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

EBigCbvLam

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

EBigCbvApp

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

What is wrong with this definition?

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

In $e \vdash t \downarrow_{cbv} v$, we expect $fv(t) \subseteq dom(e)$. 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()
```

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.

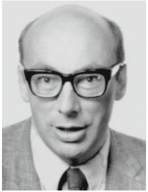
A failed attempt

Thus, our first attempt is wrong:

- It implements dynamic scoping instead of lexical scoping.
- If $e \vdash t \downarrow_{cbv} v$ and $fv(t) \subseteq dom(e)$ then we would expect that v is closed and $t[e] \downarrow_{cbv} v$ 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 $\lambda x.t$ in environment e, where $fv(\lambda x.t)$ may be nonempty, should not be just $\lambda x.t$.

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:

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

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.

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

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

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

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 $t \downarrow_{cbv} v$ and $e \vdash t \downarrow_{cbv} c$?

What lemma should we state?

Equivalence between big-step semantics without and with environments

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

What lemma should we state?

Assuming t is closed, we would like to prove that

$$t \downarrow_{cbv} v$$

holds if and only if

Equivalence between big-step semantics without and with environments

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

What lemma should we state?

Assuming t is closed, we would like to prove that

$$t \downarrow_{cbv} v$$

holds if and only if

$$[] \vdash t \downarrow_{cbv} v \qquad \text{— really?}$$

Equivalence between big-step semantics without and with environments

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

What lemma should we state?

Assuming t is closed, we would like to prove that

$$t \downarrow_{cbv} v$$

holds if and only if

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

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

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 $t \downarrow_{cbv} v$ implies $[] \vdash t \downarrow_{cbv} c$, 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 $e \vdash t \downarrow_{cbv} c$ 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 Rocq and prove this interpreter correct and complete!

As I am back in Rocq (as opposed to paper), I use de Bruijn style again.

See `LambdaCalculusInterpreter`.

Syntax

The syntax of terms is as before.

The syntax of closures and environments in de Bruijn's style is as follows:

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

Definition cenv :=
  list cvalue.
```

A closure `Clo t e` is a pair of a term and an environment.

An environment `e` is a list of closures.
It is understood as a finite map of variables to closures.

A first attempt

```
Fail Fixpoint interpret (e : cenv) (t : term) : cvalue :=
  match t with
  | Var x =>
      nth x e dummy_cvalue
        (* a dummy value 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 Rocq?

A first attempt

```
Fail Fixpoint interpret (e : cenv) (t : term) : cvalue :=
  match t with
  | Var x =>
      nth x e dummy_cvalue
        (* a dummy value 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 Rocq?

It is not structurally recursive.

In the last recursive call, no parameter decreases.

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: its result type is `option cvalue`.

`>>=` is the bind combinator of the option monad.

As soon as a subcomputation returns `None`, everything stops.

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$(t) \subseteq$ 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:

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

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.

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.*

Appel, Compiling with Continuations (page 205), 1991.

Just for fun

Here is what happened in 1960 during one of the first demonstrations of a LISP system to an industrial audience:

> Everything was going well, if slowly, when suddenly the Flexowriter began to type (at ten characters per second):
>
> "THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS:"

McCarthy, History of LISP, 1981.

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.

Scaling up

To become a real-world, comfortable programming language, the λ-calculus must be enriched with many features.

- Sometimes a feature can be considered primitive, that is, given as part of the definition of the language;
- sometimes it can be encoded, that is, explained as syntactic sugar for existing features.

The more powerful the existing features, the easier it is to encode new features.

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."

Scaling up

In the following slides, we examine how to extend λ-calculus with

- local definitions,
- integers,
- (binary) products,
- (binary) sums,
- (single) recursive functions.

Mutable state (references) is examined in a later lecture.

Scaling up

Not treated in this course:

- Exceptions.

  Wright and Felleisen, A Syntactic Approach to Type Soundness, 1994.

- Control effects; effect handlers.

  Pretnar, An Introduction to Algebraic Effects and Handlers, 2015.

- Concurrency.

  Jung *et al.*, Iris from the ground up, 2018.

- Relaxed memory.

  Kaiser *et al.*, Strong logic for weak memory, 2017.
  Mével *et al.*, Cosmo: A Concurrent Separation Logic for Multicore OCaml, 2020.

Local definitions

One could view "$let\ x = t_1\ in\ t_2$" as sugar for "$(\lambda x.t_2)\ t_1$".

This yields the desired semantics. The following are lemmas:

$$\frac{}{let\ x = v\ in\ t \longrightarrow_{cbv} t[v/x]} \qquad \frac{t \longrightarrow_{cbv} t'}{let\ x = t\ in\ u \longrightarrow_{cbv} let\ x = t'\ in\ u}$$

Or, more commonly, one views "$let\ x = t_1\ in\ t_2$" as a primitive construct.
The above rules are then part of the definition of the reduction relation.

- This lets an interpreter or compiler treat it in a more efficient way.
- This lets a type-checker treat it in a special way.
  - In ML, *let*-bound variables receive polymorphic type schemes.
- If *let* is primitive then every other construct can be restricted so that *let* is the only sequencing construct.
  - e.g., applications are restricted to the form "$v\ v$"
  - this is known as administrative normal form or monadic (normal) form.

Bowman, A Low-Level Look at A-Normal Form, 2024.

Integers

Here is how to extend the call-by-value λ-calculus with primitive integers and primitive operations on integers—here, just addition.

$$\begin{array}{rcll} t & ::= & \ldots \mid \underline{k} \mid t + t & \qquad \text{where } k \in \mathbb{Z} \\ v & ::= & \ldots \mid \underline{k} & \\ E & ::= & \ldots \mid E + t \mid v + E & \end{array}$$

One new reduction rule is needed:

$$\underline{k_1} + \underline{k_2} \longrightarrow_{cbv} \underline{k_1 + k_2}$$

Once λ-calculus is extended with new forms of values, some terms appear that cannot be reduced yet are not values: they are stuck.

$$\begin{array}{rl} \underline{42}\ \underline{24} & \text{is stuck (expected function, got integer)} \\ \underline{42} + \lambda x.x & \text{is stuck (expected integer, got function)} \end{array}$$

A stuck term can be understood as a runtime error.

Products

Here is how to extend the call-by-value λ-calculus with binary products, that is, pairs and projections.

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

One new reduction rule is needed:

$$\pi_i\ (v_0, v_1) \longrightarrow_{cbv} v_i$$

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

Here is how to extend the call-by-value λ-calculus with binary sums, that is, injections and case analysis.

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

One new reduction rule is needed:

$$case\ inj_i\ v\ of\ x_0.t_0 \parallel x_1.t_1 \longrightarrow_{cbv} t_i[v/x_i]$$

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

Recursive functions

Here is how to extend the call-by-value λ-calculus with a primitive form of recursive functions.

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

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

$\lambda x.t$ can be viewed as sugar for $\mu f.\lambda x.t$ where $f \notin fv(\lambda x.t)$.

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

The reduction rule β_v is amended as follows:

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

An equivalent and perhaps more readable formulation is:

$$\frac{u = \mu f.\lambda x.t}{u\ v \longrightarrow_{cbv} t[v/x][u/f]}$$