MPRI 2.4

Operational semantics and reduction strategies

François Pottier

2018

The λ-calculus

The formal model that underlies all functional programming languages.

Abstract syntax:

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

Reduction:

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

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

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

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 (next week).
- Enrich the language with primitive data types and operations, recursion, algebraic data structures, and so on (next week).
- Define a static type system (Rémy's lectures).

1 Reduction strategies

Operational semantics

Plotkin: — *It is only through having an operational semantics that the [λ-calculus can] be viewed as a programming language.*

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

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.

The call-by-value strategy

Values form a subset of terms:

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

A value represents the result of a computation.

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

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

This is known as a small-step operational semantics.

Example

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

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

Features of call-by-value reduction

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

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

  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.

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

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.

$$\text{AppVR}\;\frac{u \longrightarrow_{\text{cbv}} u'}{v\ u \longrightarrow_{\text{cbv}} v\ u'}$$

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

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

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:

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

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

$$\frac{\begin{array}{l}\textsc{Ctx}\\ \quad t \longrightarrow^{\text{head}}_{\text{cbv}} t'\end{array}}{E[t] \longrightarrow_{\text{cbv}} E[t']}$$

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 $t = E[u]$ and $u \longrightarrow^{head}_{cbv} \cdot$.

The call-by-name strategy

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

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

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

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

Call-by-value versus call-by-name

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

The converse is false:

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

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.

(*) 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) (Coq 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

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

where thunk T is unit $\rightarrow T$.

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

Encoding call-by-value in a CBN language

This is somewhat more involved.

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

Call-by-need

Call-by-need, also known as lazy evaluation, eliminates the main inefficiency of call-by-name (namely, possibly repeated computation) by introducing memoization.

It, too, can be defined via an operational semantics (Ariola and Felleisen, 1997; Maraist, Odersky, Wadler, 1998).

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 a square root 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!

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:

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

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.)