MPRI 2.4

Operational semantics and reduction strategies

François Pottier

2017

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:

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

Mnemonic: read $[u/x]t$ as "substitute u for x in t".

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:

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

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.

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

  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.

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

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.

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

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

Reduction sequences

Sequences of reduction steps describe the behavior of a term.

The three following 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.

Examples of reduction sequences

Termination:

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

Divergence:

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

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:

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

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

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

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 theorem:

Theorem (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 $t \longrightarrow_{\text{cbn}} t'$ is defined as follows:

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

The unevaluated actual argument is passed to the function.

It is later reduced if / when / every time the function body needs 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 ($\star$).

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.

($\star$) 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?

In a simply-typed setting, this transformation is type-preserving: that is, $\Gamma \vdash t : T$ implies $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]$, where

$$[\![T_1 \rightarrow T_2]\!] = (\text{unit} \rightarrow [\![T_1]\!]) \rightarrow [\![T_2]\!]$$

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.

Encoding call-by-need in a CBV language

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

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

"suspend ($\lambda\_.u$)" is written `lazy u` in OCaml.

"force x" is written `Lazy.force x`.

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