François Pottier

Reduction strategies

MPRI 2.4

Operational semantics and reduction strategies

François Pottier



2017

François Pottier

Reduction strategies

The λ -calculus

The formal model that underlies all functional programming languages.

Abstract syntax:

 $t, u ::= x \mid \lambda x.t \mid t t$ (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.

François Pottier

Reduction strategies

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).

François Pottier

Reduction strategies



François Pottier

Reduction strategies

Operational semantics

Plotkin: — It is only through having an operational semantics that the $[\lambda$ -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.

François Pottier

Reduction strategies

The call-by-value strategy

Values form a subset of terms:

t,u	::=	x λx.t t t	(terms)
V	::=	$x \mid \lambda x.t$	(values)

A value represents the result of a computation.

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

$$\frac{\beta_{v}}{(\lambda x.t) \ v \longrightarrow_{cbv} [v/x]t} \qquad \frac{APPL}{t \longrightarrow_{cbv} t'} \qquad \frac{APPVR}{u \longrightarrow_{cbv} t' u} \qquad \frac{u \longrightarrow_{cbv} u'}{v \ u \longrightarrow_{cbv} v \ u'}$$

This is known as a small-step operational semantics.

Example

MPRI 2.4 Semantics

François Pottier

Reduction strategies

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

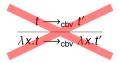
$$\frac{[x/1]x = 1}{(\lambda x.x) \ 1 \longrightarrow_{cbv} 1} \beta_{v}}{(\lambda x.\lambda y.y \ x) \ ((\lambda x.x) \ 1) \longrightarrow_{cbv} (\lambda x.\lambda y.y \ x) \ 1} APPR}{(\lambda x.\lambda y.y \ x) \ ((\lambda x.x) \ 1) \ (\lambda x.x) \longrightarrow_{cbv} (\lambda x.\lambda y.y \ x) \ 1 \ (\lambda x.x)} APPL$$

François Pottier

Reduction strategies

Features of call-by-value reduction

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

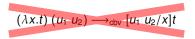


Thus, values do not reduce.

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.

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



François Pottier

Reduction strategies

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.

$$\frac{u \longrightarrow_{cbv} u'}{v \ u \longrightarrow_{cbv} v \ u'}$$

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

Reduction sequences

Sequences of reduction steps describe the behavior of a term.

The three following situations are mutually exclusive:

MPRI 2.4 Semantics

François Pottier Reduction strategies

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

François Pottier

Reduction strategies

Examples of reduction sequences

Termination:

$$(\lambda x.\lambda y.y x) ((\lambda x.x) 1) (\lambda x.x) \xrightarrow{\qquad \rightarrow_{cbv}} (\lambda x.\lambda y.y x) 1 (\lambda x.x) \xrightarrow{\qquad \rightarrow_{cbv}} (\lambda y.y 1) (\lambda x.x) \xrightarrow{\qquad \rightarrow_{cbv}} (\lambda x.x) 1 \xrightarrow{\qquad \rightarrow_{cbv}} 1$$

Divergence:

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

Error:

$$(\lambda x.x x) 2 \longrightarrow_{cbv} 2 2 \xrightarrow{}_{cbv} \cdot$$

The active redex is highlighted in red.

François Pottier

Reduction strategies

An alternative style: evaluation contexts

First, define head reduction:

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

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



where evaluation contexts E are defined by E ::= [] | E u | v E.

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

François Pottier

Reduction strategies

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 $u \longrightarrow_{cbv}^{head}$.

François Pottier

Reduction strategies

The call-by-name strategy

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

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

The unevaluated actual argument is passed to the function.

It is later reduced if / when / every time the function body needs its value.

François Pottier

Reduction strategies

An example reduction sequence

$$(\lambda x.\lambda y.y x) ((\lambda x.x) 1) (\lambda x.x) \longrightarrow_{cbn} (\lambda y.y ((\lambda x.x) 1)) (\lambda x.x) \longrightarrow_{cbn} (\lambda x.x) ((\lambda x.x) 1) \longrightarrow_{cbn} (\lambda x.x) 1 \longrightarrow_{cbn} 1$$

François Pottier

Reduction strategies

Call-by-value versus call-by-name

If *t* terminates under CBV, then it also terminates under CBN (*). The converse is false:

$$(\lambda x.1) \omega \longrightarrow_{cbn} 1$$

 $(\lambda x.1) \omega \longrightarrow_{cbv}^{\infty}$

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: (λx . 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).

François Pottier

Reduction strategies

Encoding call-by-name in a CBV language

Use thunks: functions $\lambda_{-}u$ whose purpose is to delay the evaluation of u.

$$\begin{bmatrix} x \end{bmatrix} = x ()$$

$$\lambda x.t \end{bmatrix} = \lambda x. \begin{bmatrix} t \end{bmatrix}$$

$$\begin{bmatrix} t \ u \end{bmatrix} = \begin{bmatrix} t \end{bmatrix} (\lambda_{-}. \begin{bmatrix} u \end{bmatrix})$$

Exercise: Can you state that this encoding is correct? Can you prove it?

 $\left[\right]$

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

$$\llbracket T_1 \to T_2 \rrbracket = (\operatorname{unit} \to \llbracket T_1 \rrbracket) \to \llbracket T_2 \rrbracket$$

François Pottier

Reduction strategies

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

MPRI 2.4 Semantics

François Pottier

Reduction strategies

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.

François Pottier

Reduction strategies

Encoding call-by-need in a CBV language

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

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

"suspend (λ_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.