Functional programming languages

Part I: operational semantics

Xavier Leroy

INRIA Rocquencourt

MPRI 2-4-2, 2006

What is a functional programming language?

Various answers:

- By examples: Caml, SML, Haskell, Scheme, . . .
- "A language that takes functions seriously"
- A language that manipulates functions with free variables as first-class values, following (more or less) the model of the λ-calculus.

Example 1

```
let rec map f lst =
  match lst with [] -> [] | hd :: tl -> f hd :: map f tl

let add_to_list x lst =
  map (fun y -> x + y) lst
```

The λ-calculus

The formal model that underlies all functional programming languages.

λ-calculus, recap

Terms: $a, b ::= x \mid \lambda x.a \mid a\ b$

Rewriting rule: $(\lambda x.a)\ b \rightarrow a\{x \leftarrow b\}$ $\quad$ (β-reduction)

From λ-calculus to a functional programming language

Take the λ-calculus and:

- Fix a reduction strategy.
  β-reductions in the λ-calculus can occur anywhere and in any order. This can affect termination and algorithmic efficiency of programs. A fixed reduction strategy enables the programmer to reason about termination and algorithmic complexity.
- Add primitive data types (integers, strings), primitive operations (arithmetic, logical), and primitive data structures (lists, records).
  All these can be encoded in the λ-calculus, but the encodings are unnatural and inefficient. These notions are so familiar to programmer as to deserve language support.
- Develop efficient execution models.
  Repeated rewriting by the β rule is a terribly inefficient way to execute programs on a computer.

Outline

In this lecture:

1. Reduction strategies
2. Enriching the language
3. Efficient execution models
   - Natural semantics
   - Environments and closures
   - Explicit substitutions

Call-by-value in structural operational style (SOS)

(G. Plotkin, 1981)

Terms (programs) and values (results of evaluation):

Terms:	$a, b ::= N$	integer constant
	$\mid x$	variable
	$\mid \lambda x.\ a$	function abstraction
	$\mid a\ b$	function application
Values:	$v ::= N \mid \lambda x.\ a$	

One-step reduction relation $a \rightarrow a'$, in SOS:

$$(\lambda x.a)\ v \rightarrow a[x \leftarrow v] \qquad (\beta_v)$$

$$\frac{a \rightarrow a'}{a\ b \rightarrow a'\ b}\ (\text{app-l}) \qquad\qquad \frac{b \rightarrow b'}{v\ b \rightarrow v\ b'}\ (\text{app-r})$$

Example of reduction

$$\dfrac{\dfrac{(\lambda x.x)\ 1 \to x[x \leftarrow 1] = 1}{(\lambda x.\lambda y.\ y\ x)\ ((\lambda x.x)\ 1) \to (\lambda x.\lambda y.\ y\ x)\ 1}\ \text{(app-r)}}{(\lambda x.\lambda y.\ y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x) \to (\lambda x.\lambda y.\ y\ x)\ 1\ (\lambda x.x)}\ \text{(app-l)}$$

Features of the reduction relation

- Weak reduction:
  We cannot reduce under a λ-abstraction.

$$\xcancel{\dfrac{a \to a'}{\lambda x.a \to \lambda x.a'}}$$

- Call-by-value:
  In an application $(\lambda x.a)\ b$, the argument b must be fully reduced to a value before β-reduction can take place.

$$(\lambda x.a)\ v \to a[x \leftarrow v]$$

Features of the reduction relation

- Left-to-right:
  In an application $a\ b$, we must reduce a to a value first before we can start reducing b.

$$\frac{b \to b'}{v\ b \to v\ b'}$$

  (Right-to-left is equally acceptable and equally easy to specify.)
- Deterministic:
  For every term a, there is at most one a' such that $a \to a'$.

  (Since values cannot reduce, there is at most one rule among (β_v), (app-l) and (app-r) that applies to a given term.)

Reduction sequences

Elementary reductions can be chained to describe how a term evaluates:

- Termination: $a \to a_1 \to a_2 \to \ldots \to v$
  The value v is the result of evaluating a.
- Divergence: $a \to a_1 \to a_2 \to \ldots \to a_n \to \ldots$
  The sequence of reductions is infinite.
- Error: $a \to a_1 \to a_2 \to \ldots \to a_n \not\to$
  when a_n is not a value but does not reduce.

Examples of reduction sequences

Terminating:

$$\begin{array}{rcl}(\lambda x.\lambda y.\ y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x) & \rightarrow & (\lambda x.\lambda y.\ y\ x)\ 1\ (\lambda x.x) \\ & \rightarrow & (\lambda y.\ y\ 1)\ (\lambda x.x) \\ & \rightarrow & (\lambda x.x)\ 1 \\ & \rightarrow & 1\end{array}$$

Error: $(\lambda x.\ x\ x)\ 2 \rightarrow 2\ 2 \not\rightarrow$

Divergence: $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow \ldots$

An alternative to SOS: reduction contexts

(A. Wright and M. Felleisen, 1992)

First, define head reductions (at the top of a term):

$$(\lambda x.a)\ v \xrightarrow{\varepsilon} a[x \leftarrow v] \quad (\beta_v)$$

Then, define reduction as head reduction within a reduction context:

$$\frac{a \xrightarrow{\varepsilon} a'}{E[a] \rightarrow E[a']} \text{ (context)}$$

where reduction context E (terms with a hole denoted $[\,]$) are defined by the following grammar:

$$\begin{array}{lll}E ::= & [\,] & \text{reduction at top of term} \\ & \mid E\ b & \text{reduction in the left part of an application} \\ & \mid v\ E & \text{reduction in the right part of an application}\end{array}$$

Example of reductions with contexts

$(\lambda x.\lambda y.\ y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x)$	take $E = (\lambda x.\lambda y.\ y\ x)\ [\,]\ (\lambda x.x)$
$\rightarrow (\lambda x.\lambda y.\ y\ x)\ 1\ (\lambda x.x)$	take $E = [\,]\ (\lambda x.x)$
$\rightarrow (\lambda y.\ y\ 1)\ (\lambda x.x)$	take $E = [\,]$
$\rightarrow (\lambda x.x)\ 1$	take $E = [\,]$
$\rightarrow 1$	

Equivalence between SOS and contexts

Reduction contexts in the Wright-Felleisen approach are in one-to-one correspondence with derivations in the SOS approach.
For example, the SOS derivation:

$$\dfrac{\dfrac{(\lambda x.x)\ 1 \rightarrow 1}{(\lambda x.\lambda y.\ y\ x)\ ((\lambda x.x)\ 1) \rightarrow (\lambda x.\lambda y.\ y\ x)\ 1}\ (\text{app-r})}{(\lambda x.\lambda y.\ y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x) \rightarrow (\lambda x.\lambda y.\ y\ x)\ 1\ (\lambda x.x)}\ (\text{app-l})$$

corresponds to the context $E = ((\lambda x.\lambda y.\ y\ x)\ [\,])\ (\lambda x.x)$ and the head reduction $(\lambda x.x)\ 1 \xrightarrow{\varepsilon} 1$.

Determinism of reduction under contexts

In the Wright-Felleisen approach, the determinism of the reduction relation $\rightarrow$ is a consequence of the following unique decomposition theorem:

Theorem 2

For all terms a, there exists at most one reduction context E and one term b such that $a = E[b]$ and b can reduce by head reduction.

(Exercise.)

Call-by-name

Unlike call-by-value, call-by-name does not evaluate arguments before performing β-reduction. Instead, it performs β-reduction as soon as possible; the argument will be reduced later when its value is needed in the function body.

In SOS style:

$$(\lambda x.a)\ b \rightarrow a[x \leftarrow b]\ \ (\beta_n) \qquad\qquad \frac{a \rightarrow a'}{a\ b \rightarrow a'\ b}\ \text{(app-l)}$$

In Wright-Felleisen style:

$(\lambda x.a)\ b \xrightarrow{\varepsilon} a[x \leftarrow b]$ with reduction contexts $E ::= [\,] \mid E\ b$ i.e. $E ::= [\,]$

Example of reduction sequence

$$
\begin{array}{rcl}
(\lambda x.\lambda y.\ y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x) & \rightarrow & (\lambda y.\ y\ ((\lambda x.x)\ 1))\ (\lambda x.x) \\
& \rightarrow & (\lambda x.x)\ ((\lambda x.x)\ 1) \\
& \rightarrow & (\lambda x.x)\ 1 \\
& \rightarrow & 1
\end{array}
$$

Outline

Enriching the language

Terms:	$a, b ::= N$	integer constant
	$\mid x$	variable
	$\mid \lambda x.\ a$	function abstraction
	$\mid a\ b$	function application
	$\mid \mu f.\lambda x.\ a$	recursive function
	$\mid a\ op\ b$	arithmetic operation
	$\mid C(a_1, \ldots, a_n)$	data structure construction
	$\mid \texttt{match}\ a\ \texttt{with}\ p_1 \mid \ldots \mid p_n$	pattern-matching
Operators:	$op ::= + \mid - \mid \ldots \mid < \mid = \mid > \mid \ldots$	
Patterns:	$p ::= C(x_1, \ldots, x_n) \rightarrow a$	
Values:	$v ::= N \mid C(v_1, \ldots, v_n)$	
	$\mid \lambda x.\ a \mid \mu f.\lambda x.\ a$	

Example

The Caml expression

```
fun x lst ->
  let rec map f lst =
    match lst with [] -> [] | hd :: tl -> f hd :: map f tl
  in
    map (fun y -> x + y) lst
```

can be expressed as

```
λx. λlst.
  (μmap. λf. λlst.
    match lst with Nil() → Nil()
                 | Cons(hd, tl) → Cons(f hd, map f tl))
  (λy. x + y) lst
```

Some derived forms

`let` and `let rec` bindings:

$$\begin{array}{lcl}
\texttt{let } x = a \texttt{ in } b & \mapsto & (\lambda x.b)\ a \\
\texttt{let } f\ x_1 \ldots x_n = a \texttt{ in } b & \mapsto & \texttt{let } f = \lambda x_1 \ldots \lambda x_n.a \texttt{ in } b \\
\texttt{let rec } f\ x = a \texttt{ in } b & \mapsto & \texttt{let } f = \mu f.\lambda x.a \texttt{ in } b \\
\texttt{let rec } f\ x = a & \mapsto & \texttt{let rec } f\ g\ x = (\texttt{let } f = f\ g \texttt{ in } a) \texttt{ in} \\
\texttt{and } g\ y = b & & \texttt{let rec } g\ y = (\texttt{let } f = f\ g \texttt{ in } b) \texttt{ in} \\
\texttt{in } c & & \texttt{let } f = f\ g \texttt{ in } c
\end{array}$$

Some derived forms

Booleans and if-then-else:

$$\begin{array}{lcl}
\texttt{true} & \mapsto & \texttt{True}() \\
\texttt{false} & \mapsto & \texttt{False}() \\
\texttt{if } a \texttt{ then } b \texttt{ else } c & \mapsto & \texttt{match } a \texttt{ with True}() \rightarrow b \mid \texttt{False}() \rightarrow c
\end{array}$$

Pairs and projections:

$$\begin{array}{lcl}
(a, b) & \mapsto & \texttt{Pair}(a, b) \\
\texttt{fst}(a) & \mapsto & \texttt{match } a \texttt{ with Pair}(x, y) \rightarrow x \\
\texttt{snd}(a) & \mapsto & \texttt{match } a \texttt{ with Pair}(x, y) \rightarrow y
\end{array}$$

Reduction rules

Head reductions:

$$\begin{array}{rcl}
(\mu f.\lambda x.a)\ v & \xrightarrow{\varepsilon} & a[f \leftarrow \mu f.\lambda x.a,\ x \leftarrow v] \\
N_1 + N_2 & \xrightarrow{\varepsilon} & N \quad \text{if } N = N_1 + N_2 \\
N_1 < N_2 & \xrightarrow{\varepsilon} & \texttt{true}() \quad \text{if } N_1 < N_2 \\
N_1 < N_2 & \xrightarrow{\varepsilon} & \texttt{false}() \quad \text{if } N_1 \geq N_2 \\
\texttt{match } C(\vec{v}) \texttt{ with } C(\vec{x}) \to a \mid \vec{p} & \xrightarrow{\varepsilon} & a[\vec{x} \leftarrow \vec{v}] \quad \text{if } |\vec{v}| = |\vec{x}| \\
\texttt{match } C(\vec{v}) \texttt{ with } C'(\vec{x}) \to a \mid \vec{p} & \xrightarrow{\varepsilon} & \texttt{match } C(\vec{v}) \texttt{ with } \vec{p} \quad \text{if } C \neq C'
\end{array}$$

Contexts:

$E ::= \ldots \mid E\ op\ a \mid v\ op\ E \mid C(\vec{v}, E, \vec{a}) \mid \texttt{match } E \texttt{ with } p$

Reduction rules for derived forms

Derived forms have sensible reduction rules that can be deduced from the rules on the previous page. For instance:

$$\begin{array}{rcl}
\texttt{let } x = v \texttt{ in a} & \xrightarrow{\varepsilon} & a[x \leftarrow v] \\
\texttt{if true then } a \texttt{ else } b & \xrightarrow{\varepsilon} & a \\
\texttt{if false then } a \texttt{ else } b & \xrightarrow{\varepsilon} & b \\
\texttt{fst}(v_1, v_2) & \xrightarrow{\varepsilon} & v_1 \\
\texttt{snd}(v_1, v_2) & \xrightarrow{\varepsilon} & v_2
\end{array}$$

Similarly for contexts:

$E ::= \ldots \mid \texttt{let } x = E \texttt{ in } a \mid \texttt{if } E \texttt{ then } a \texttt{ else } b$
$\quad \mid (E, a) \mid (v, E) \mid \texttt{fst}(E) \mid \texttt{snd}(E)$

Outline

1. Reduction strategies
2. Enriching the language
3. Efficient execution models
   - Natural semantics
   - Environments and closures
   - Explicit substitutions

A naive interpreter that follows the reduction semantics

```
type term =
  Const of int | Var of string | Lam of string * term | App of term * term

let rec subst x v = function          (* assumes v is closed *)
  | Const n -> Const n
  | Var y -> if x = y then v else Var y
  | Lam(y, b) -> if x = y then Lam(y, b) else Lam(y, subst x v b)
  | App(b, c) -> App(subst x v b, subst x v c)

let isvalue = function Const _ -> true | Lam _ -> true | _ -> false

exception Cannot_reduce

let rec reduce = function
  | App(Lam(x, a), v) when isvalue v -> subst x v a
  | App(a, b) -> if isvalue a then App(a, reduce b) else App(reduce a, b)
  | _ -> raise Cannot_reduce

let rec evaluate a = try evaluate (reduce a) with Cannot_reduce -> a
```

Algorithmic inefficiencies

Each reduction step needs to:

1. Find the next redex, i.e. decompose the program a into $E[(\lambda x.b)\ v]$
   $\Rightarrow$ time $O(height(a))$
2. Perform the substitution $b[x \leftarrow v]$
   $\Rightarrow$ time $O(size(b))$
3. Reconstruct the term $E[b[x \leftarrow v]]$
   $\Rightarrow$ time $O(height(a))$

Each reduction step takes non-constant time, in the worst case: linear in the size of the program.

Alternative to reduction sequences

We first address inefficiencies 1 and 3: finding the next redex and reconstructing the program after reduction.

Goal: amortize this cost over whole reduction sequences to a value.

$$a\ b \xrightarrow{a \xrightarrow{*} (\lambda x.c)} (\lambda x.c)\ b \xrightarrow{b \xrightarrow{*} v'} (\lambda x.c)\ v' \rightarrow c[x \leftarrow v'] \xrightarrow{*} v$$

Idea: define a relation $a\ b \Rightarrow v$ that follows this structure: first, reduce a to a function value; then, reduce b to a value; then, do the substitution; finally, reduce the substituted term to a value.

Natural semantics, a.k.a. big-step semantics

(G. Kahn, 1987)

Define a relation $a \Rightarrow v$, meaning "a evaluates to value v", by inference rules that follow the structure of a. The rules for call-by-value are:

$$N \Rightarrow N \qquad\qquad \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{a \Rightarrow \lambda x.c \qquad b \Rightarrow v' \qquad c[x \leftarrow v'] \Rightarrow v}{a\ b \Rightarrow v}$$

For call-by-name, replace the application rule by:

$$\frac{a \Rightarrow \lambda x.c \qquad c[x \leftarrow b] \Rightarrow v}{a\ b \Rightarrow v}$$

Example of evaluation derivation

$$\dfrac{\dfrac{\begin{array}{c}\lambda x.\lambda y.y\ x \\ \Rightarrow \lambda x.\lambda y.y\ x\end{array} \qquad \dfrac{\begin{array}{c}\lambda x.x \Rightarrow \lambda x.x \\ 1 \Rightarrow 1 \\ 1 \Rightarrow 1\end{array}}{(\lambda x.x)\ 1 \Rightarrow 1} \qquad \begin{array}{c}\lambda y.y\ 1 \\ \Rightarrow \lambda y.y\ 1\end{array}}{(\lambda x.\lambda y.y\ x)\ ((\lambda x.x)\ 1) \Rightarrow \lambda y.y\ 1} \qquad \lambda x.x \Rightarrow \lambda x.x \qquad \dfrac{\begin{array}{c}\lambda x.x \Rightarrow \lambda x.x \\ 1 \Rightarrow 1 \\ 1 \Rightarrow 1\end{array}}{(\lambda x.x)\ 1 \Rightarrow 1}}{(\lambda x.\lambda y.y\ x)\ ((\lambda x.x)\ 1)\ (\lambda x.x) \Rightarrow 1}$$

Evaluation rules for language extensions

$$\frac{a \Rightarrow \mu f.\lambda x.c \qquad b \Rightarrow v' \qquad c[f \leftarrow \mu f.\lambda x.c, x \leftarrow v'] \Rightarrow v}{a\ b \Rightarrow v}$$

$$\frac{a \Rightarrow N_1 \qquad b \Rightarrow N_2 \qquad N = N_1 + N_2}{a + b \Rightarrow N}$$

$$\frac{a \Rightarrow C(\vec{v}) \qquad |\vec{v}| = |\vec{x}| \qquad b[\vec{x} \leftarrow \vec{v}] \Rightarrow v}{\texttt{match}\ a\ \texttt{with}\ C(\vec{x}) \rightarrow b \mid p \Rightarrow v}$$

$$\frac{a \Rightarrow C'(\vec{v}) \qquad C' \neq C \qquad \texttt{match}\ a\ \texttt{with}\ p \Rightarrow v}{\texttt{match}\ a\ \texttt{with}\ C(\vec{x}) \rightarrow b \mid p \Rightarrow v}$$

An interpreter that follows the natural semantics

```
exception Error

let rec eval = function
  | Const n -> Const n
  | Var x -> raise Error
  | Lam(x, a) -> Lam(x, a)
  | App(a, b) ->
      match eval a with
      | Lam(x, c) -> let v = eval b in eval (subst x v c)
      | _ -> raise Error
```

Note the complete disappearance of inefficiencies 1 and 3.

Equivalence between reduction and natural semantics

Theorem 3

If $a \Rightarrow v$, then $a \xrightarrow{*} v$.

Proof.

By induction on the derivation of $a \Rightarrow v$ and case analysis on a.
If $a = n$ or $a = \lambda x.b$, then $v = a$ and the result is obvious.
If $a = b\ c$, applying the induction hypothesis to the premises of $b\ c \Rightarrow v$, we obtain three reduction sequences:

$$b \xrightarrow{*} \lambda x.d \qquad c \xrightarrow{*} v' \qquad d[x \leftarrow v'] \xrightarrow{*} v$$

Combining them together, we obtain the desired reduction sequence:

$$b\ c \xrightarrow{*} (\lambda x.d)\ c \xrightarrow{*} (\lambda x.d)\ v' \rightarrow d[x \leftarrow v'] \xrightarrow{*} v$$

□

Equivalence between reduction and natural semantics

Theorem 4

If $a \xrightarrow{*} v$, where v is a value, then $a \Rightarrow v$.

Proof.

Follows from the two properties below and an easy induction on the length of the reduction sequence $a \xrightarrow{*} v$.

1. $v \Rightarrow v$ for all values v (trivial)
2. If $a \rightarrow b$ and $b \Rightarrow v$, then $a \Rightarrow v$ (exercise).

□

Alternative to textual substitution

Need: bind a variable x to a value v in a term a.

Inefficient approach: the textual substitution $a[x \leftarrow v]$.

Alternative: remember the binding $x \mapsto v$ in an auxiliary data structure called an environment. When we need the value of x during evaluation, just look it up in the environment.

The evaluation relation becomes $e \vdash a \Rightarrow v$
e is a partial mapping from names to values (CBV) or to terms (CBN).

Additional evaluation rule for variables:

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

Lexical scoping

```
let x = 1 in
let f = λy.x in
let x = "foo" in
f 0
```

In what environment should the body of the function `f` evaluate when we compute the value of `f 0` ?

- Dynamic scoping: in the environment current at the time we evaluate `f 0`. In this environment, `x` is bound to `"foo"`.
  This is inconsistent with the λ-calculus model and is generally considered as a bad idea.
- Lexical scoping: in the environment current at the time the function `f` was defined. In this environment, `x` is bound to `1`.
  This is what the λ-calculus prescribes.

Function closures

(P.J. Landin, 1964)

To implement lexical scoping, function abstractions $\lambda x.a$ must not evaluate to themselves, but to a function closure: a pair

$$(\lambda x.a)[e]$$

of the function text and an environment e associating values to the free variables of the function.

`let x = 1 in`	$\mathtt{x} \mapsto \mathtt{1}$
`let f = λy.x in`	$\mathtt{x} \mapsto \mathtt{1};\ \mathtt{f} \mapsto (\lambda \mathtt{y}.\mathtt{x})[\mathtt{x} \mapsto \mathtt{1}]$
`let x = "foo" in`	$\mathtt{x} \mapsto \mathtt{"foo"};\ \mathtt{f} \mapsto (\lambda \mathtt{y}.\mathtt{x})[\mathtt{x} \mapsto \mathtt{1}]$
`f 0`	evaluate `x` in environment $\mathtt{x} \mapsto \mathtt{1};\ \mathtt{y} \mapsto \mathtt{0}$

Natural semantics with environments and closures

Values: $v ::= N \mid (\lambda x.a)[e]$

Environments: $e ::= x_1 \mapsto v_1; \dots ; x_n \mapsto v_n$

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow (\lambda x.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda x.c)[e'] \qquad e \vdash b \Rightarrow v' \qquad e' + (x \mapsto v') \vdash c \Rightarrow v}{e \vdash a\ b \Rightarrow v}$$

From variable names to de Bruijn indices

(N. de Bruijn, 1972)

Instead of identifying variables by their names, de Bruijn's notation identifies them by their position relative to the λ-abstraction that binds them.

$$
\begin{array}{llll}
\lambda\mathtt{x}. & (\lambda\mathtt{y}. & \mathtt{y}\ \ \mathtt{x}) & \mathtt{x} \\
 & & |\ \ \ | & | \\
\lambda. & (\lambda. & \underline{1}\ \ \underline{2}) & \underline{1}
\end{array}
$$

$\underline{n}$ is the variable bound by the n-th enclosing λ.

Environments become sequences of values $e ::= v_1 \ldots v_n$
The n-th value is the value of variable $\underline{n}$.

Environments, closures and de Bruijn indices

Terms:	$a ::= N \mid \underline{n} \mid \lambda.a \mid a_1\ a_2$
Values:	$v ::= N \mid (\lambda.a)[e]$
Environments:	$e ::= v_1 \ldots v_n$

$$\frac{e = v_1 \ldots v_n \ldots v_m}{e \vdash \underline{n} \Rightarrow v_n} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda.a \Rightarrow (\lambda.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda.c)[e'] \qquad e \vdash b \Rightarrow v' \qquad v'.e' \vdash c \Rightarrow v}{e \vdash a\ b \Rightarrow v}$$

The canonical, efficient interpreter

The combination of natural semantics, environments, closures and de Bruijn indices leads to a call-by-value interpreter with no obvious algorithmic inefficiencies.

```
type term = Const of int | Var of int | Lam of term | App of term * term

type value = Vint of int | Vclos of term * value environment

let rec eval e a =
  match a with
  | Const n -> Vint n
  | Var n -> env_lookup n e
  | Lam a -> Vclos(Lam a, e)
  | App(a, b) ->
      match eval e a with
      | Vclos(Lam c, e') ->
          let v = eval e b in
          eval (env_add v e') c
      | _ -> raise Error
```


The canonical, efficient interpreter

The type α `environment` and the operations `env_lookup`, `env_add` can be chosen among different data structures:

Data structure	Cost of `lookup`	Cost of `add`
List	$O(n)$	O(1)
Array	$O(1)$	O(n)
Patricia tree	$O(\log n)$	$O(\log n)$

Environments as parallel substitutions

To reason about environment- and closure-based semantics, it is helpful to view environments as parallel substitutions

$$e = v_1 \dots v_n \quad \approx \quad [\underline{1} \leftarrow v_1; \dots; \underline{n} \leftarrow v_n]$$

and closures as ordinary terms

$$a[e] \quad \approx \quad \text{the substitution } e \text{ applied to } a$$

Application: proving the equivalence between the natural semantics with and without environments.

Theorem 5

$e \vdash a \Rightarrow v$ if and only if $a[e] \Rightarrow v$.

Explicit substitutions in reduction semantics

Going one step further, the notion of environment can be internalized within the language, i.e. presented as explicit terms with appropriate reduction rules. This is called the λ-calculus with explicit substitutions.

Terms:	$a ::= N \mid \underline{n} \mid \lambda.a \mid a_1\ a_2 \mid a[e]$
Environments/substitutions:	$e ::= id \mid a.e$
Values:	$v ::= N \mid (\lambda.a)[e]$

Explicit substitutions, M. Abadi, L. Cardelli, P.L. Curien, J.J. Lévy, *Journal of Functional Programming* 6(2), 1996.
Confluence properties of weak and strong calculi of explicit substitutions, P.L. Curien, T. Hardin, J.J. Lévy, *Journal of the ACM* 43(2), 1996.

Reduction rules with explicit substitutions

For call-by-value:

$$\begin{array}{rcl} \underline{n}\,[v_1 \ldots v_n \ldots] & \stackrel{\varepsilon}{\rightarrow} & v_n \\ (\lambda.a)[e]\ v & \stackrel{\varepsilon}{\rightarrow} & a[v.e] \\ (a\ b)[e] & \stackrel{\varepsilon}{\rightarrow} & a[e]\ b[e] \end{array}$$

For call-by-name:

$$\begin{array}{rcl} \underline{n}\,[a_1 \ldots a_n \ldots] & \stackrel{\varepsilon}{\rightarrow} & a_n \\ (\lambda.a)[e]\ b & \stackrel{\varepsilon}{\rightarrow} & a[b.e] \\ (a\ b)[e] & \stackrel{\varepsilon}{\rightarrow} & a[e]\ b[e] \end{array}$$

Same contexts as before.