Functional programming languages Part I: operational semantics

Xavier Leroy

INRIA Rocquencourt

MPRI 2-4-2, 2006

X. Leroy (INRIA)

Functional programming languages

MPRI 2-4-2, 2006 1 / 45

What is a functional programming language?

Various answers:

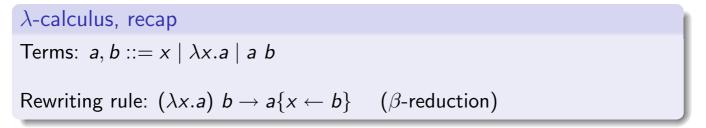
- By examples: Caml, SML, Haskell, Scheme, ...
- "A language that takes functions seriously"
- A language that manipulates functions with free variables as first-class values, following (more or less) the model of the λ-calculus.

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.



X. Leroy (INRIA)

Functional programming languages

MPRI 2-4-2, 2006 3 / 45

From $\lambda\text{-calculus}$ 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

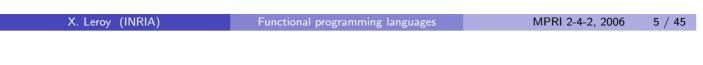
In this lecture:

1 Reduction strategies



Efficient execution models

- Natural semantics
- Environments and closures
- Explicit substitutions



Reduction strategies

Call-by-value in structural operational style (SOS) (G. Plotkin, 1981)

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

Terms:	a, b ::= N	integer constant
	X	variable
	<i>\lambda x. a</i>	function abstraction
	a b	function application
Values	$v = M \mid \lambda v \rangle$	

Values: $v ::= N \mid \lambda x. a$

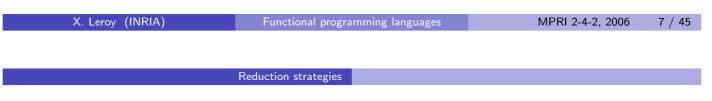
One-step reduction relation $a \rightarrow a'$, in SOS:

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

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

Example of reduction

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



Features of the reduction relation

• Weak reduction:

We cannot reduce under a λ -abstraction.

$$\frac{a \rightarrow a'}{\lambda x.a} \rightarrow \lambda x.a'$$

• 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.

$$(\lambda x.a) \mathbf{v} \rightarrow a[x \leftarrow v]$$

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 \rightarrow a'.
```

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

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4-2, 2006	9 / 45
	Reduction strategies		
	Reduction strategies		

Reduction sequences

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

- Termination: $a \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow v$ The value v is the result of evaluating a.
- Divergence: $a \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \ldots$ The sequence of reductions is infinite.
- Error: $a \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \not\rightarrow$ 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: $(\lambda x. x x) 2 \rightarrow 2 2 \neq$

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

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4-2, 2006	11 / 45
	Reduction strategies		

An alternative to SOS: reduction contexts

(A. Wright and M. Felleisen, 1992)

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

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

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

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

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

E ::= [] reduction at top of term| E b reduction in the left part of an application| v E reduction in the right part of an application

Example of reductions with contexts

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

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

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

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

$$\rightarrow 1$$

$$MR12-2.200 13/45$$

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:

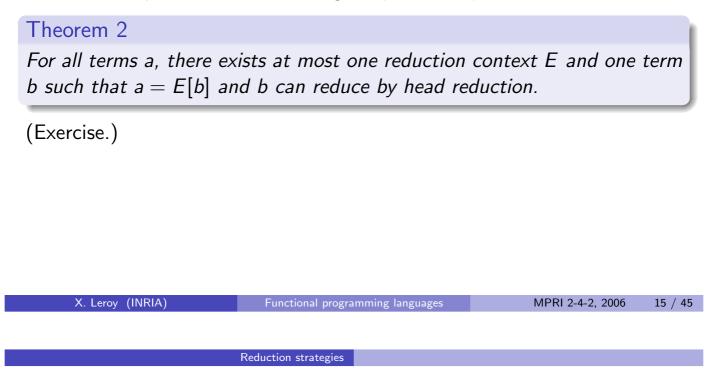
$$\frac{(\lambda x.x) \ 1 \to 1}{(\lambda x.\lambda y. \ y \ x) \ ((\lambda x.x) \ 1) \to (\lambda x.\lambda y. \ y \ x) \ 1} (\mathsf{app-r})$$
$$\frac{(\lambda x.\lambda y. \ y \ x) \ ((\lambda x.x) \ 1) \to (\lambda x.\lambda y. \ y \ x) \ 1}{(\lambda x.\lambda y. \ y \ x) \ ((\lambda x.x) \ 1) \ (\lambda x.x) \to (\lambda x.\lambda y. \ y \ x) \ 1 \ (\lambda x.x)} (\mathsf{app-l})$$

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

Reduction strategies

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:



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:

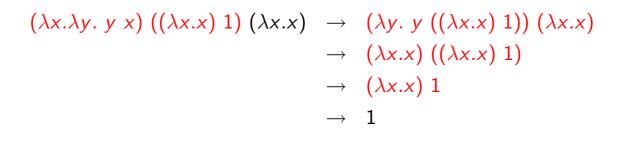
$$(\lambda x.a) \ b \to a[x \leftarrow b] \ (\beta_n) \qquad \qquad \frac{a \to a'}{a \ b \to a' \ b} \ (app-l)$$

In Wright-Felleisen style:

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

Reduction strategies

Example of reduction sequence



X. Leroy (INRIA)	Functional programming languages	MPRI 2-4-2, 2006	17 / 45

Enriching the language

Outline

Reduction strategies

2 Enriching the language

3 Efficient execution models

- Natural semantics
- Environments and closures
- Explicit substitutions

Enriching the language

a, *b* ::= *N*

Terms:

	<i>x</i>
	$\mid \lambda x. a$
	a b
	$\mid \mu f.\lambda x.$ a
	a op b
	$\mid C(a_1,\ldots,a_n)$
	\mid match a with $p_1 \mid \ldots \mid p_n$
Operators:	$op ::= + - \dots < = > \dots$
Patterns:	$p ::= C(x_1, \ldots, x_n) o a$
Values:	$v ::= N \mid C(v_1, \ldots, v_n)$
	$\mid \lambda x. \; a \mid \mu f. \lambda x. \; a$

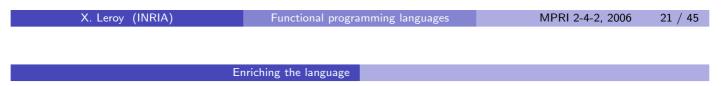
integer constant variable function abstraction function application recursive function arithmetic operation data structure construction pattern-matching

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4-2, 2006 19) / 45
Er	riching the language		
Example			
The Caml expression			
<pre>fun x lst -> let rec map f ls match lst with in map (fun y -></pre>	h [] -> [] hd :: tl	-> f hd :: map f	tl
can be expressed as			
	h Nil() $ ightarrow$ Nil() Cons(hd, tl) $ ightarrow$ Cons	s(f hd, map f tl))	

Enriching the language

Some derived forms

let and let rec bindings:



Some derived forms

Booleans and if-then-else:

Pairs and projections:

$$(a,b) \mapsto \operatorname{Pair}(a,b)$$

fst $(a) \mapsto \operatorname{match} a \operatorname{with} \operatorname{Pair}(x,y) \to x$
snd $(a) \mapsto \operatorname{match} a \operatorname{with} \operatorname{Pair}(x,y) \to y$

Enriching the language

Reduction rules

Head reductions:

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

Contexts:

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

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4-2, 2006	23 / 45
E	nriching the language		

Reduction rules for derived forms

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

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

Similarly for contexts:

$$E ::= \dots | \text{let } x = E \text{ in } a | \text{if } E \text{ then } a \text{ else } b \\ | (E, a) | (v, E) | \text{fst}(E) | \text{snd}(E)$$

Outline

1 Reduction strategies

2 Enriching the language

3 Efficient execution models

- Natural semantics
- Environments and closures
- Explicit substitutions

X. Leroy (INRIA)

Functional programming languages

MPRI 2-4-2, 2006 25 / 45

Efficient execution models

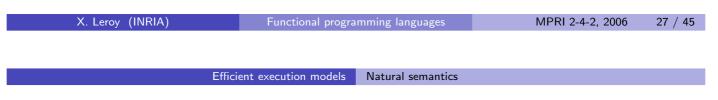
A naive interpreter that follows the reduction semantics

Algorithmic inefficiencies

Each reduction step needs to:

- Find the next redex, i.e. decompose the program *a* into $E[(\lambda x.b) v]$ $\Rightarrow \text{ time } O(height(a))$
- Perform the substitution $b[x \leftarrow v]$ $\Rightarrow \text{ time } O(size(b))$
- Solution 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.

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

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:

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

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4-2, 2006	29 / 45

Natural semantics

Example of evaluation derivation

$$\frac{\lambda x.x \Rightarrow \lambda x.x}{1 \Rightarrow 1} \\
\frac{\lambda x.\lambda y.y x}{\Rightarrow \lambda x.\lambda y.y x} \frac{1 \Rightarrow 1}{(\lambda x.x) 1 \Rightarrow 1} \\
\frac{\lambda y.y 1}{\Rightarrow \lambda y.y 1} \\
\frac{\lambda x.x \Rightarrow \lambda x.x}{1 \Rightarrow 1} \\
\frac{\lambda x.x \Rightarrow \lambda x$$

Evaluation rules for language extensions

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

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

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

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

X. Leroy (INRIA)	Functional program	mming languages	MPRI 2-4-2, 2006	31 / 45
E	fficient execution models	Natural semantics		

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.

Efficient execution models Natural semantics

Equivalence between reduction and natural semantics

Theorem 3

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

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:

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

Combining them together, we obtain the desired reduction sequence:

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

$$\square$$
X. Leroy (INRIA) Functional programming languages MPRI 2-4-2, 2006 33 / 45

Efficient execution models Natural semantics

Equivalence between reduction and natural semantics

Theorem 4

If $a \stackrel{*}{\rightarrow} v$, 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 $a \xrightarrow{*} v$.

$$v \Rightarrow v \text{ for all values } v (trivial)$$

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:

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

X. Leroy (INRIA)	Functional progra	mming languages	MPRI 2-4-2, 2006	35 / 45
Effi	cient execution models	Environments and clo	sures	

Lexical scoping

let x = 1 in let $f = \lambda y.x$ 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.

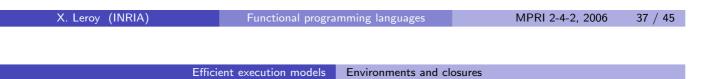
Function closures (P.J. Landin, 1964)

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	$x \mapsto 1$
let f = $\lambda y.x$ in	x \mapsto 1; f \mapsto (λ y.x)[x \mapsto 1]
<pre>let x = "foo" in</pre>	x \mapsto "foo"; f \mapsto (λ y.x)[x \mapsto 1]
f 0	evaluate x in environment x \mapsto 1; y \mapsto 0



Natural semantics with environments and closures

Values: $v ::= N \mid (\lambda x.a)[e]$ Environments: $e ::= x_1 \mapsto v_1; \ldots; x_n \mapsto v_n$

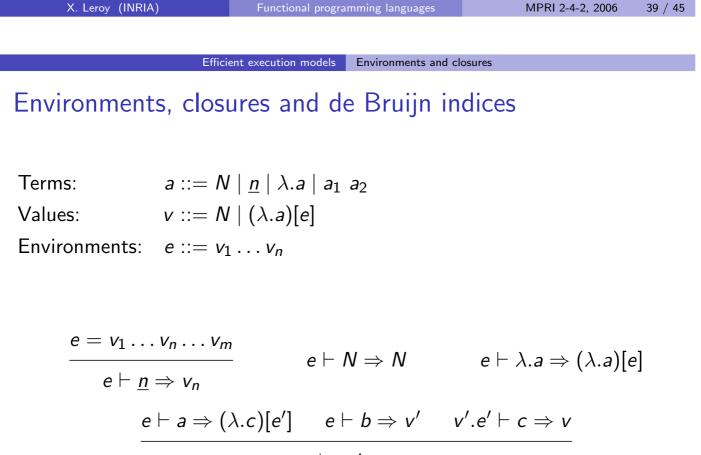
$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow (\lambda x.a)[e]$$
$$\frac{e \vdash a \Rightarrow (\lambda x.c)[e'] \qquad e \vdash b \Rightarrow v' \qquad e' + (x \mapsto v') \vdash c \Rightarrow v}{e \vdash a \ b \Rightarrow v}$$

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.

<u>*n*</u> is the variable bound by the *n*-th enclosing λ .

Environments become sequences of values $e ::= v_1 \dots v_n$ The *n*-th value is the value of variable <u>*n*</u>.



 $e \vdash a \ b \Rightarrow v$

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 in	t Var of int Lam of term	App of term * term
type value = Vint of in	t Vclos of term * value en	vironment
<pre>let rec eval e a = match a with Const n -> Vint n Var n -> env_lookup Lam a -> Vclos(Lam a App(a, b) -> match eval e a wir</pre>	a, e) th) -> e b in v e') c	
X. Leroy (INRIA)	Functional programming languages	MPRI 2-4-2, 2006 41 / 45

Efficient execution models Environments and closures

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	<i>O</i> (<i>n</i>)	O(1)
Array	O(1)	O(n)
Patricia tree	$O(\log n)$	$O(\log n)$

Environments as parallel substitutions

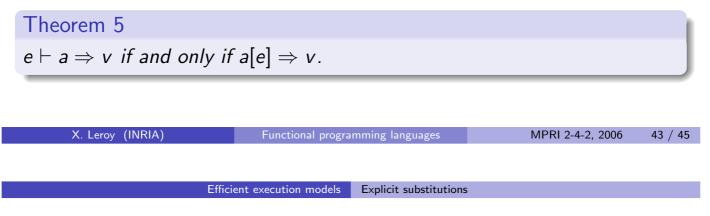
To reason about environment- and closure-based semantics, it is helpful to view environments as parallel substitutions

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

and closures as ordinary terms

 $a[e] \approx$ the substitution e applied to a

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



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:	$a ::= N \mid \underline{n} \mid \lambda.a \mid a_1 \mid a_2 \mid \underline{a[e]}$
Environments/substitutions:	e ::= <i>id</i> <i>a.e</i>
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.

Efficient execution models Explicit substitutions

Reduction rules with explicit substitutions

For call-by-value:

$$\begin{array}{cccc} \underline{n} \left[v_1 \dots v_n \dots \right] & \stackrel{\varepsilon}{\to} & v_n \\ (\lambda.a) [e] & v & \stackrel{\varepsilon}{\to} & a[v.e] \\ & (a \ b) [e] & \stackrel{\varepsilon}{\to} & a[e] \ b[e] \end{array}$$

For call-by-name:

$$\begin{array}{rcl}\underline{n} \left[a_{1} \dots a_{n} \dots \right] & \stackrel{\varepsilon}{\to} & a_{n} \\ & (\lambda.a) [e] & b & \stackrel{\varepsilon}{\to} & a [b.e] \\ & (a & b) [e] & \stackrel{\varepsilon}{\to} & a [e] & b [e] \end{array}$$

Same contexts as before.

X. Leroy (INRIA)

Functional programming languages

MPRI 2-4-2, 2006 45 / 45