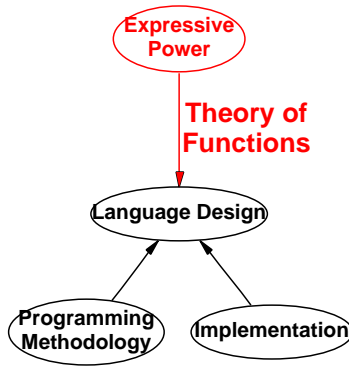


# Theory of Programming Languages



# Our view of theory

Precise answer to “what does this program mean?”

(a guide to implementors)

- lambda calculus
- type systems
- partial orders and lattices
- denotational semantics

## The lambda calculus

World’s simplest programming language:

variables, abstraction, application

```

exp ::= var
      | λ var . exp
      | exp exp
  
```

Typically  $M, N, \dots$  stand for *exps* ( $\lambda$ -terms)

$x, y, z, \dots$  stand for variables

Application associates to left, binds tighter than abstraction (parenthesize as needed)

## Lambda calculus, continued

Sometimes we will add constants like +, 1, 2:

```

add2    = λx. + x2
add2'   = λx. + 2x
add2''  = +2
revapply = λx. λy. yx
  
```

Amazingly, as powerful as any known programming language

even without constants!

Ideal vehicle for

- proving theorems
- experimenting with features

## Capsule view

**Abstract syntax:** application, abstraction, variable

**Values:** values are terms!!

- Typically terms in normal form
- Justifies the name “calculus”

**Environments:** Not used!

- (but names stand for terms)

**Evaluation rules:** coming up

**Initial basis:** sometimes empty, sometimes constants

## Operational semantics of lambda calculus

New kind of operational semantics: small-step

Judgment:  $M \rightarrow N$  (“ $M$  reduces to  $N$  in one step”)

- No environment!
- Just pushing terms around: calculus

## Reduction rules

### Central rule based on substitution

$$\frac{}{(\lambda x.M)N \xrightarrow{\beta} M[x \mapsto N]} \quad \text{(BETA)}$$

### Structural rules: Beta-reduce anywhere, any time

$$\frac{N \xrightarrow{\beta} N'}{MN \xrightarrow{\beta} MN'} \quad \frac{M \xrightarrow{\beta} M'}{MN \xrightarrow{\beta} M'N} \quad \frac{M \xrightarrow{\beta} M'}{\lambda x.M \xrightarrow{\beta} \lambda x.M'}$$

© Copyright 2002 Norman Ramsey. All Rights Reserved.

7

## Evaluating lambda-terms

### Use beta-reduction for applications, delta-reduction for constants

$$\begin{aligned} & (\lambda x.\lambda y.yx)(+34)(\lambda x.+x2) \xrightarrow{\beta} \\ & (\lambda y.y(+34))(\lambda x.+x2) \xrightarrow{\beta} \\ & (\lambda x.+x2)(+34) \xrightarrow{\beta} \\ & +(+34)2 \xrightarrow{\delta} \\ & +72 \xrightarrow{\delta} \\ & 9 \end{aligned}$$

© Copyright 2002 Norman Ramsey. All Rights Reserved.

8

## Substitution

### Heart of the lambda calculus and difficult to implement correctly!

$$\begin{aligned} x[x \mapsto M] &= M \\ y[x \mapsto M] &= y \\ (YZ)[x \mapsto M] &= (Y[x \mapsto M])(Z[x \mapsto M]) \\ (\lambda x.Y)[x \mapsto M] &= \lambda x.Y \\ (\lambda y.Z)[x \mapsto M] &= \lambda y.Z[x \mapsto M] \\ &\quad \text{if } x \text{ not free in } Z \text{ or } y \text{ not free in } M \\ (\lambda y.Z)[x \mapsto M] &= \lambda w.(Z[y \mapsto w])[x \mapsto M] \\ &\quad \text{where } w \text{ not free in } Z \text{ or } M \end{aligned}$$

### Last transformation is renaming of bound variables

© Copyright 2002 Norman Ramsey. All Rights Reserved.

9

## Renaming of bound variables

### So important it has its own Greek letter:

$$\frac{w \text{ not free in } Z}{\lambda y.Z \xrightarrow{\alpha} \lambda w.(Z[y \mapsto w])} \quad \text{(ALPHA)}$$

### Also has structural rules

### Don't rename bound variables? Watch out for capture!

$$\begin{aligned} (\lambda y.yx)[x \mapsto z] &= \lambda y.yz \\ (\lambda y.yx)[x \mapsto yz] &\stackrel{?}{=} \lambda y.yyz \quad \text{not likely—"captured" } y \\ (\lambda y.yx)[x \mapsto yz] &= \lambda w.wyz \end{aligned}$$

© Copyright 2002 Norman Ramsey. All Rights Reserved.

10

## Conversion

### Alpha-conversion (rename bound variable)

$$\frac{y \text{ not free in } Z}{\lambda x.Z \xrightarrow{\alpha} \lambda y.Z[x \mapsto y]}$$

### Beta-conversion (the serious evaluation rule)

$$\frac{}{(\lambda x.M)N \xrightarrow{\beta} M[x \mapsto N]}$$

### Eta-conversion:

$$\frac{x \text{ not free in } M}{\lambda x.Mx \xrightarrow{\eta} M}$$

### All structural: Convert whole term or subterm

© Copyright 2002 Norman Ramsey. All Rights Reserved.

11

## The really important theorem

### Two terms that convert are in some sense equivalent because of

## Church-Rosser Theorem

if  $A \rightarrow B$  and  $A \rightarrow C$   
there exists  $D$  s.t.  $B \rightarrow^* D$  and  $C \rightarrow^* D$

Real theorists prove this theorem...

© Copyright 2002 Norman Ramsey. All Rights Reserved.

12

## Normal Forms

If there is no  $B$  such that  $A \rightarrow B$ ,  $A$  is in normal form  
(typically we ignore alpha-conversion)

So called because of Church-Rosser.

Corollary:

if  $A \rightarrow^* B$ ,  $B$  in normal form, and  
 $A \rightarrow^* C$ ,  $C$  in normal form

then  $B$  and  $C$  are **identical**  
(up to renaming of bound variables)

## How to get a normal form

Normalization theorem:

If there is a normal form, can get to it by taking  
“leftmost, outermost redex”  
“normal order of evaluation”  
• to guarantee normal form  
• **not** the “normal way of doing things”

Danger of infinite loops:

$$(\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} (\lambda x.xx)(\lambda x.xx)$$

But

$$(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx)) \xrightarrow{\beta} \lambda y.y$$

Think “bodies before arguments”

## Computing with the lambda calculus

We convince ourselves we can write programs

Booleans:

true =  $\lambda x.\lambda y.x$   
false =  $\lambda x.\lambda y.y$   
if  $M$  then  $N$  else  $P$  =  $MNP$

Products (tuples, records, structs):

pair =  $\lambda x.\lambda y.\lambda f.fxy$   
fst =  $\lambda p.p(\lambda x.\lambda y.x)$   
snd =  $\lambda p.p(\lambda x.\lambda y.y)$

## Sums (discriminated unions)

ML has strong native support:

```
datatype ('a, 'b) sum = L of 'a | R of 'b
```

Scheme: as in C, a record with explicit tags

```
; let union be pair of tag and value
(val L (lambda (x) (cons 'left x)))
(val R (lambda (x) (cons 'right x)))
```

Lambda-calculus: functions!

$L = \lambda a.\lambda f.\lambda g.fa$   
 $R = \lambda b.\lambda f.\lambda g.gb$

## Getting a value out of a sum

Must be prepared for two cases; ML has it built in

```
case X of L a => M | R b => N
```

where  $a$  is free in  $M$ ,  $b$  is free in  $N$

Scheme requires explicit tag and extract:

```
(if (= (car X) 'left)
    (let ((a (cdr X))) M)
    (let ((b (cdr X))) N))
```

Lambda calculus—supply function for each case

either  $X(\lambda a.M)(\lambda b.N)$   
where either =  $\lambda x.\lambda left.\lambda right.x left right$   
 $L = \lambda a.\lambda f.\lambda g.fa$   
 $R = \lambda b.\lambda f.\lambda g.gb$

## Simulating S-expressions

Sum of nil or product.

nil =  $\lambda f.\lambda g.f \perp$   
cons =  $\lambda a.\lambda d.\lambda f.\lambda g.gad$   
car =  $\lambda p.p \perp (\lambda a.\lambda d.a)$   
cdr =  $\lambda p.p \perp (\lambda a.\lambda d.d)$   
null? =  $\lambda p.p(\lambda x.true)(\lambda a.\lambda d.false)$   
 $\perp = (\lambda x.xx)(\lambda x.xx)$

Relies on normal-order evaluation  
(won't work in Scheme)

## Encoding natural numbers

```
datatype nat = z | s of nat
val zero = z
val one = s(z)
val two = s(s(z))
val three = s(s(s(z)))
```

Define fold s.t. fold  $f$   $x$  replaces  $s \mapsto f$  and  $z \mapsto x$

```
fun fold f x z = x
  | fold f x (s n) = f (fold f x n)
```

Example: fold (fn k => k+1) 0 three ↓ 3

Church: represent  $n$  as  $\lambda f. \lambda x. \text{fold } f \ x \ n.$

## Church Numerals

### Encoding natural numbers as lambda-terms

```
zero = λf.λx.x
one = λf.λx.f x
two = λf.λx.f(f x)
n = λf.λx.f(n)x
succ = λn.λf.λx.f(n f x)
plus = λn.λm.n succ m
times = λn.λm.n (plus m) zero
```

## Church Numerals in Scheme

```
(val zero (lambda (f) (lambda (x) x)))
(val succ (lambda (n) (lambda (f)
  (lambda (x) (f ((n f) x))))))
(val three (succ (succ (succ zero))))
(val four (succ three))
(val plus (lambda (n) (lambda (m)
  ((n succ) m))))
(val times (lambda (n) (lambda (m)
  ((n (plus m)) zero))))
(val to-int (lambda (n)
  ((n (lambda (x) (+ x 1))) 0)))
-> (to-int three)
3
-> (to-int ((times three) four))
12
```

## Solving recursion equations

What is

$\text{fact} = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)?$

Not a definition (mathematical nonsense!)

An equation to be solved for fact.

Key idea: Recursion = fixed point

Write  $g = \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n-1)$

Solve  $\text{fact} = g \ \text{fact}$ : find fixed point of  $g$

## Is fixed point for real?

Recall  $g = \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n-1)$

Is there a function  $F$  such that  $g F = F$ ?

If so, then  $F$  is the factorial function

Proof by induction:

```
g F 0 = 1 (true for any F)
g F n = if n = 0 then 1 else n × F(n-1)
        = n × F(n-1)
        = n × factorial(n-1) (induction hypothesis)
        = factorial n
```

## Finding a fixed point

We can find a fixed point for any function.

Let

```
Y = λf.(λx.f(xx))(λx.f(xx))
Y g = (λx.g(xx))(λx.g(xx))
```

and by beta-conversion

```
Y g = g ((λx.g(xx))(λx.g(xx)))
Y g = g (Y g)
```

so

$F = Y g$

Actually, for any  $h$ ,  $Y h = h (Y h)$

$Y$  is a “fixed-point combinator”

## Using fixedpoint to implement fixedpoint

$\mu$ ML `val-rec` inherently recursive (fixedpoint built in)

```
-> (val-rec fix
      (lambda (f) (lambda (x) ((f (fix f)) x))))
fix : (('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)
-> (val g (lambda (fact)
            (lambda (n)
              (if (= n 0) 1 (* n (fact (- n 1)))))))
-> (val f (fix g))
-> (f 0)
1 : int
-> (f 5)
120 : int
-> (f 6)
720 : int
```

`fix` useful in interpreters

## Syntactic sugar for the lambda calculus

$(M, N)$	$\stackrel{def}{=}$	$\lambda f. f M N$
$\lambda(x, y). M$	$\stackrel{def}{=}$	$\lambda p. p(\lambda x. \lambda y. M)$
<code>let v = M in N</code>	$\stackrel{def}{=}$	$(\lambda v. N)M$
<code>letrec v = M in N</code>	$\stackrel{def}{=}$	$(\lambda v. N)(Y(\lambda v. M))$
$0, 1, 2, \dots$	$\stackrel{def}{=}$	$\lambda f. \lambda x. x, \lambda f. \lambda x. fx, \lambda f. \lambda x. f(fx) \dots$
$M + N$	$\stackrel{def}{=}$	$(\lambda x. \lambda y. x \text{ succ } y) M N$
<code>true</code>	$\stackrel{def}{=}$	$\lambda x. \lambda y. x$
<code>false</code>	$\stackrel{def}{=}$	$\lambda x. \lambda y. y$
<code>if P then M else N</code>	$\stackrel{def}{=}$	$PMN$

also products and sums (union, datatype) as above

## Typed lambda calculus

First-order typed lambda calculus—simplified version of Typed Impcore:

```
exp ::= var
      |  $\lambda$  var : type . exp
      | exp exp
      | const
type ::= base-type (e.g., int)
      | type  $\rightarrow$  type
```

Type rules as you would expect.

## Second-order typed lambda-calculus

Simplified Typed  $\mu$ Scheme (one arg, Curry everywhere):

```
exp ::=
  var                variable
  |  $\lambda$  var : type . exp  functional abstraction
  | exp exp          application
  | const            constant
  |  $\Lambda \alpha . exp$     type abstraction
  | exp [type]       type application
type ::=
  tycon              type constructor: int, bool, ...
  | tyvar            type variable:  $\alpha, \beta, \dots$ 
  |  $\forall \alpha . type$ 
  | type type
```

Type rules as for Typed  $\mu$ Scheme.

## Second-order examples: Church numerals

```
zero =  $\Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. x : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ 
three =  $\Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. f(f(fx))$ 
      :  $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ 
succ =  $\lambda n : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha).$ 
       $\Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. f(n[\alpha]fx)$ 
      :  $(\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ 
       $\rightarrow (\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ 
add =  $\lambda n : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha).$ 
       $n[\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)] \text{succ}$ 
      :  $(\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ 
       $\rightarrow (\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ 
       $\rightarrow (\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$ 
```

## Evaluating typed calculi

Theorem (“Erasure”):

Can evaluate typed  $\lambda$ -calculus (1st- and 2nd-order) by erasing all types and  $\Lambda$ 's, and evaluating as for untyped  $\lambda$ -calculus

Essentially done in Typed  $\mu$ Scheme, although interpreter erases types lazily.