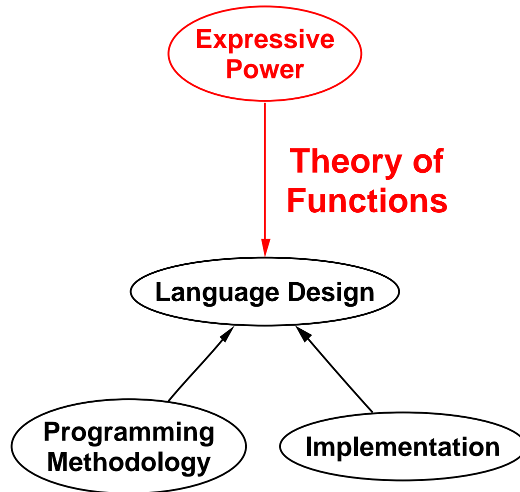
Theory of Programming Languages



©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Reserved.

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
- algebraic equivalence
- logical assertions

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Reserved.

The lambda calculus

World's simplest programming language:

variables, abstraction, application

exp ::= *var*
| λ *var* . *exp*
| *exp exp*

Typically $M, N, \ldots$ stand for *exps* (λ-terms)
$x, y, z, \ldots$ stand for variables

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

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Reserved.

Can encode any data structures

Recall from 91.301 that data can be encoded as functions!

```
(define cons (x y)
  (lambda (msg)
    (if (= msg 1)
        x
        (if (= msg 2)
            y))))
 (define car (p) (p 1))
 (define cdr (p) (p 2))
```

uScheme cons cell as function.

Need to define `if`, `=`, `1`, `2` in λ-calc: can do this.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Reserved.

Lambda calculus, continued

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

$$\begin{aligned} add2 &= \lambda x. + x 2 \\ add2' &= \lambda x. + 2 x \\ add2'' &= +2 \\ revapply &= \lambda x.\lambda y.yx \end{aligned}$$

Amazingly, as powerful as any known programming language

even without constants!

Ideal vehicle for

- proving theorems
- experimenting with features

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Another functional language

λ-calculus Scheme ML

– no need to evaluate before applying.
– no need for recursive definitions.
– no need for "if".
– can do everything with just functions.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

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

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Operational semantics of lambda calculus

New kind of operational semantics: small-step

Judgment: $M \to N$ ("M reduces to N in one step")

- No environment!
- Just pushing terms around: calculus

The substitution model in 91.301

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Reduction rules

Central rule based on substitution

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

Structural rules: Beta-reduce anywhere, any time

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

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Evaluating lambda-terms

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

$$\underline{(\lambda x.\lambda y.yx)(+\,3\,4)}(\lambda x.+x\,2) \xrightarrow{\beta}$$
$$\underline{(\lambda y.y(+\,3\,4))(\lambda x.+x\,2)} \xrightarrow{\beta}$$
$$\underline{(\lambda x.+x\,2)(+\,3\,4)} \xrightarrow{\beta}$$
$$+\underline{(+\,3\,4)}\,2 \xrightarrow{\delta}$$
$$\underline{+\,7\,2} \xrightarrow{\delta}$$
$$9$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Bound and Free variables

Seen bound variables before in programming languages, predicate logic, integral calculus...

$BV(M)$, the set of all bound variables in M:

$$BV(x) = \emptyset$$
$$BV(\lambda x.M) = BV(M) \cup \{x\}$$
$$BV(MN) = BV(M) \cup BV(N)$$

$FV(M)$, the set of all free variables in M:

$$FV(x) = \{x\}$$
$$FV(\lambda x.M) = FV(M) - \{x\}$$
$$FV(MN) = FV(M) \cup FV(N)$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Substitution

Heart of the lambda calculus
and difficult to implement correctly!

1. $x[M/x] = M$
2. $y[M/x] = y$
3. $(YZ)[M/x] = (Y[M/x])(Z[M/x])$
4. $(\lambda x.Y)[M/x] = \lambda x.Y$
5. $(\lambda y.Z)[M/x] = \lambda y.Z[M/x]$ if y not free in M
6. $(\lambda y.Z)[M/x] = \lambda w.(Z[w/y])[M/x]$ where w not free in Z or M

Last transformation is renaming of bound variables

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Renaming of bound variables

So important it has its own Greek letter:

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

Also has structural rules

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

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

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Conversion

Alpha-conversion (rename bound variable)

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

Beta-conversion (the serious evaluation rule)

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

Eta-conversion (all things are function things)

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

All structural: Convert whole term or subterm

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Equality

From any set of reduction rules, we can define a set of equality rules.

$$\frac{M \to N}{M = N} \qquad \frac{}{M = M} \qquad \frac{M = N}{N = M} \qquad \frac{M = N \quad N = P}{M = P}$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

The really important theorem

Two terms that convert are in some sense equivalent because of

Church-Rosser Theorem

if $B = C$
there exists D s.t. $B \to^* D$ and $C \to^* D$

(Proof in 91.538)

A syntactic proof of consistency in 1930s.
No semantic proof of consistency until 1960s.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Normal Forms

If there is no B such that $A \rightarrow B$, A is in normal form
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. Typically we ignore alpha-conversion. Theoreticians talk about equivalence classes under alpha-conversion.)

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

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 "apply" rather than "eval and apply"

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Computing with the lambda calculus

We convince ourselves we can write programs

Booleans:

$$\begin{aligned}
\text{true} &= \lambda x.\lambda y.x \\
\text{false} &= \lambda x.\lambda y.y \\
\text{if} &= \lambda p.\lambda x.\lambda y.pxy \\
\text{if } P \text{ then } M \text{ else } N &= PMN
\end{aligned}$$

Products (tuples, records, structs):

$$\begin{aligned}
\text{pair} &= \lambda x.\lambda y.\lambda f.fxy \\
\text{fst} &= \lambda p.p(\lambda x.\lambda y.x) \\
\text{snd} &= \lambda p.p(\lambda x.\lambda y.y)
\end{aligned}$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

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!

$$\begin{aligned}
L &= \lambda a.\lambda f.\lambda g.fa \\
R &= \lambda b.\lambda f.\lambda g.gb
\end{aligned}$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

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

$\texttt{either}\ X\,(\lambda a.M)\,(\lambda b.N)$

where $\texttt{either} = \lambda x.\lambda left.\lambda right.x\ left\ right$
$L = \lambda a.\lambda f.\lambda g.fa$
$R = \lambda b.\lambda f.\lambda g.gb$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Simulating S-expressions

Sum of nil or product.

$$
\begin{aligned}
\texttt{nil} &= \lambda f.\lambda g.f\bot \\
\texttt{cons} &= \lambda a.\lambda d.\lambda f.\lambda g.gad \\
\texttt{car} &= \lambda p.p\bot(\lambda a.\lambda d.a) \\
\texttt{cdr} &= \lambda p.p\bot(\lambda a.\lambda d.d) \\
\texttt{null?} &= \lambda p.p(\lambda x.true)(\lambda a.\lambda d.false) \\
\bot &= (\lambda x.xx)(\lambda x.xx)
\end{aligned}
$$

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

A different encoding on Paulson page 15.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

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` $\Downarrow$ 3

Church: represent n **as** $\lambda f.\lambda x.$`fold` $f\ x\ n$.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Church Numerals

Encoding natural numbers as lambda-terms

$$
\begin{aligned}
\text{zero} &= \lambda f.\lambda x.x \\
\text{one} &= \lambda f.\lambda x.f\ x \\
\text{two} &= \lambda f.\lambda x.f(f\ x) \\
\text{n} &= \lambda f.\lambda x.f^{(n)}x \\
\text{succ} &= \lambda n.\lambda f.\lambda x.f(n\ f\ x) \\
\text{plus} &= \lambda n.\lambda m.n\ \text{succ}\ m \\
\text{times} &= \lambda n.\lambda m.n\ (\text{plus}\ m)\ \text{zero}
\end{aligned}
$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

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
```

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

From Calculus to Language

Lambda calculus β-reduces anywhere, stops at normal form:

- optimizes functions as well as applying them (reduction under λ).
- keeps on reducing even if already error (reduction of operand when operator is free variable).

Taming λ-calculus:

- What is an acceptable form for redex (β vs. β_v).
- How to pick a redex (leftmost / rightmost / random, innermost / outermost).
- When to stop (normal form / head nf / weak hnf).

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Beta-value Reduction

β reduction

$$\frac{}{(\lambda x.M)P \xrightarrow{\beta} M[P/x]} \quad \textbf{(BETA)}$$

β_v reduction of λ-abstraction

$$\frac{}{(\lambda x.M)(\lambda y.N) \xrightarrow{\beta_v} M[(\lambda y.N)/x]} \quad \textbf{(BETA-V)}$$

In general

$$\frac{}{(\lambda x.M)V \xrightarrow{\beta_v} M[V/x]} \quad \textbf{(BETA-V)}$$

where V is one of λ-abstraction, variable, constant.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Head-normal forms

Head normal form

$\lambda x_1.\cdots\lambda x_n.xM_1\cdots M_m \qquad n \geq 0$

Weak Head-normal form

$\lambda x_1.\cdots\lambda x_n.M_1\cdots M_m \qquad n > 0$ or $(n = 0$ and $M_1 \equiv x)$

Logicians like Head normal form: all terms that do not have head normal forms are indistinguishable.

Prog Lang people like Weak Head-normal form:

- Don't reduce inside a function body
- Don't reduce operator if you can not reduce operand.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Reduction orders

- leftmost / rightmost / random: what is relative order of reducing in operator vs reducing in operand if both are possible.
- outermost – find reduction as close to the top of the abstract syntax tree of the term as possible. Don't reduce operands. Modified by leftmost (resp. rightmost): in $M\ N$ if not a redex, check first for redex in M (resp. N).
- innermost – find reduction as close to the bottom of the abstract syntax tree as possible. Reduce arguments vefore applying function.

Many others possible...

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Reduction orders

normal: leftmost-outermost β-reduction to normal form.

Call-by-name: leftmost-outermost not inside a λ or if operator is a variable to weak head-normal form. (Haskell, Miranda, SASL, ...)

applicative: leftmost-innermost β_v-reduction to Normal form.

Call-by-value: innermost β_v-reduction not inside a λ or if operator is a variable to weak head-normal form. (ML (leftmost), Scheme (random), C, ...)

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Reductions for languages: Call-by-Name

Lambda Calculus:

$$\frac{}{(\lambda x.M)N \xrightarrow{\beta} M[N/x]} \quad \frac{M \xrightarrow{\beta} M'}{MN \xrightarrow{\beta} M'N} \quad \frac{N \xrightarrow{\beta} N'}{MN \xrightarrow{\beta} MN'} \quad \frac{M \xrightarrow{\beta} M'}{\lambda x.M \xrightarrow{\beta} \lambda x.M'}$$

Call-by-name

$$\frac{}{(\lambda x.M)N \xrightarrow{\beta} M[N/x]} \quad \frac{M \xrightarrow{\beta} M'}{MN \xrightarrow{\beta} M'N}$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Reductions for languages: Call-by-Value

Lambda Calculus:

$$\frac{}{(\lambda x.M)N \xrightarrow{\beta} M[N/x]} \quad \frac{M \xrightarrow{\beta} M'}{MN \xrightarrow{\beta} M'N} \quad \frac{N \xrightarrow{\beta} N'}{MN \xrightarrow{\beta} MN'} \quad \frac{M \xrightarrow{\beta} M'}{\lambda x.M \xrightarrow{\beta} \lambda x.M'}$$

Call-by-value

$$\frac{}{(\lambda x.M)V \xrightarrow{\beta}_v M[V/x]} \quad \frac{M \xrightarrow{\beta}_v M'}{MN \xrightarrow{\beta}_v M'N} \quad \frac{N \xrightarrow{\beta}_v N'}{VN \xrightarrow{\beta}_v VN'}$$

Where V is a value: either x a free variable, or $\lambda x.M$ a λ-abstraction (function).

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Evaluation context format

Define a grammar for where evaluations can occur in a term. [] "hole" must be filled by a redex. λ-calc:

```
E ::= []
    | E M
    | M E
    | λ var . E
```

Call-by-Name

```
E ::= []
    | E M
```

Call-by-Value

```
E ::= []
    | E M
    | V E
```

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Tiny functional language with numbers

Evaluation contexts:

```
E ::= []
    | (if E M M)
    | (E M)
    | (V E)
    | (op E)
```

Reduction rules:

$$\text{E}[((\text{lambda } (x)\ M)\ N)] \rightarrow \text{E}[M[N/x]]$$
$$\text{E}[(\text{if } 0\ M\ N)] \rightarrow \text{E}[N]$$
$$\text{E}[(\text{if } V\ M\ N)] \rightarrow \text{E}[M] \qquad V \neq 0$$
$$\text{E}[\text{add1 } n] \rightarrow \text{E}[m] \qquad m = n + 1$$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Combinators and Combinator reduction

Reduce using rules that we made when defining named terms. IF TRUE M N → M

IF TRUE does not reduce even though λ term does

IF, TRUE: boolean combinators, M, N: any terms.

A combinator in λ-calculus is any closed term.

Combinator reduction in λ-calculus requires that a term not be reduced unless the number of following expressions is ≥ the number of leading λs.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Combinators and Combinator reduction

Combinatory logic: start with terms S, K and rules for them.

Some standard combinators

name	reduction rule	λ term
I	I M → M	$\lambda x.x$
K	K M N → M	$\lambda x.\lambda y.x$
S	S P Q R → (P R) (Q R)	$\lambda x.\lambda y.\lambda z.xz(yz)$

S, K can be used to do anything that λ-calculus can do but without worrying about variables. Almost: if substitute lambda terms in CL-normal form may be able to do some more reduction. For example I = S K K in λ-calc, but S K K not reducible in CL. Miranda compiler uses combinators: S, K, additional combinators for performance.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Consistency / Inconsistency

If claim that K = S, then can prove any terms A, B that A = B.

K M N P = S M N P	so by defn of K, S:
M P = (M P) (N P)	consider M = I, P = I:
I I = I I (N I)	reduce a few times:
I = N I	consider N = K R:
I = K R I	reduce K:
I = R	for any R, so all terms = I = each other!

A system that proves everything equal is **inconsistent**.

Moral: be careful about adding rules.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Solving recursion equations

What is

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

Not a definition (mathematical nonsense!)

An equation to be solved for `fact`.

Key idea: Recursion $=$ fixed point

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

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

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Is fixed point for real?

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

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)

$g\,F\,n = \text{if } n = 0 \text{ then 1 else } n \times F(n-1)$

$= n \times F(n-1)$

$= n \times \text{factorial}(n-1)$ (induction hypothesis)

$= \text{factorial}\,n$

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Fixpoint Combinators

X is a fixed point of M if MX = X

Suppose we have a term Y such that YM = M(YM) then YM is a fixpoint of M.

If X = YM, turn equation around and get MX = X

A combinator (closed term) Y that satisfies the rule: "For all M, YM = M(YM)" is called a fixpoint combinator.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Finding a fixed point

We can find a fixed point for *any* function.

Let

$$Y = \lambda f.(\lambda d.f(dd))(\lambda d.f(dd))$$
$$Y\,g = (\lambda d.g(dd))(\lambda d.g(dd))$$

and by beta-reduction

$$Y\,g = g\,((\lambda d.g(dd))(\lambda d.g(dd))$$
$$Y\,g = g\,(Y\,g)$$

so

$$F = Y\,g$$

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

Y is a "fixed-point combinator" there are many.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

A use for η

Problem: Y does not work for call-by-value.

$Y(\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n-1))5 =$
nontermination

$Y = \lambda f.(\lambda d.f(dd))(\lambda d.f(dd))$

$Y_v = \lambda f.(\lambda d.f(\lambda a.(dd)a))(\lambda d.f(\lambda a.(dd)a))$

η-expansion ($M \rightarrow \lambda x.Mx$) saves the day.

Don't create $f(Yf)$ from Yf until argument a is available.

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Syntactic sugar for the lambda calculus

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

also products and sums (union, datatype) as above

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

A metacircular evaluator for λ-calculus

Mogensen's interpreter:

A metacircular interpreter in language L is a term E in L such that for all terms M in L
E (xlate (M)) = $M \Downarrow$

absyn(x)	Var(x)	
absyn($M\ N$)	App(absyn(M), absyn(N))	
absyn($\lambda x.M$)	Abs($\lambda x.$ absyn(M))	(H.O.A.S)

xlate(x)	$\lambda a.\lambda b.\lambda c.a\ x$
xlate($M\ N$)	$\lambda a.\lambda b.\lambda c.b$ xlate(M) xlate(N)
xlate($\lambda x.M$)	$\lambda a.\lambda b.\lambda c.c\ \lambda x.$ xlate(M)

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

A metacircular evaluator for λ-calculus II

$E[\text{Var}(x)] = x$
$E[\text{App}(M,N)] = E[M]\ E[N]$
$E[\text{Abs}(M)] = \lambda v.E[(M\ v)]$

$$\mathbf{E} \equiv Y\ \lambda e.\lambda m.m\ (\lambda x.x)\ (\lambda m.\lambda n.(e\ m)(e\ n))\ (\lambda m.\lambda v.e(m\ v))$$

xlate(x) $\lambda a.\lambda b.\lambda c.a\ x$
xlate($M\ N$) $\lambda a.\lambda b.\lambda c.b$ xlate(M) xlate(N)
xlate($\lambda x.M$) $\lambda a.\lambda b.\lambda c.c\ \lambda x.$ xlate(M)

Is E(xlate(λx.x x)) = λx.x x ?

Time for automated tools...

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.

Summary

- λ-calculus is simplest functional language
- Small-step operational semantics via substitution
- Is Turing Complete (can encode any computation)
- Limit to get behavior of real functional languages.
- Context rules or Evaluation Context grammars.
- Fixpoint combinators for solving recursion equations.

(equivalence of environments to substitutions obvious but messy to prove).

©Copyright 2002 Norman Ramsey, revisions copyright 2004 Allyn Dimock. All Rights Rese rved.