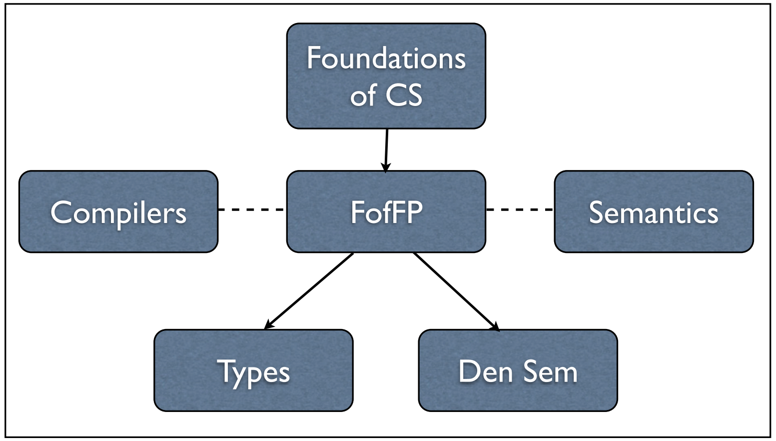
Foundations of functional programming

Matthew Parkinson
12 Lectures (Lent 2008)

Overview



Materials

Previous years notes are still relevant. Will get copies printed if sufficient demand.

Caveat: What's in the slides is what's examinable.

Motivation

Understanding:

- simple notion of computation

Encoding:

- Representing complex features in terms of simpler features

Functional programming in the wild:

- Visual Basic and C# have functional programming features.

(Pure) λ-calculus

$$M ::= x \mid (M\ M) \mid (\lambda x.M)$$

Syntax:
- x   variable
- (M M)   (function) application
- (λx.M)   (lambda) abstraction

World smallest programming language:
- α,β,η reductions
- when are two programs equal?
- choice of evaluation strategies

Pure λ-calculus is universal

Can encode:
- Booleans
- Integers
- Pairs
- Disjoint sums
- Lists
- Recursion

within the λ–calculus.

Can simulate a Turing or Register machine (Computation Theory), so is universal.

Applied λ-calculus

$$M ::= x \mid \lambda x.M \mid M\ M \mid c$$

Syntax:
- x   variables
- λx.M   (lambda) abstraction
- M M   (function) application
- c   (constants)

Elements of c used to represent integers, and also functions such as addition
- δ reductions are added to deal with constants

Combinators

$$M ::= M\ M \mid c \quad \text{(omit } x \text{ and } \lambda x.M\text{)}$$

We just have $c \in \{S, K\}$ regains power of λ–calculus.

Translation to/from lambda calculus including almost equivalent reduction rules.

Evaluation mechanisms/facts

Eager evaluation (Call-by-value)

Lazy evaluation (Call-by-need)

Confluence "There's always a meeting place downstream"

Implementation Techniques

Real implementations

- "Functional Languages"
- Don't do substitution, use environments instead.
- Haskell, ML, F# (, Visual Basic, C#)

SECD

Abstract machine for executing the λ-calculus.

4 registers Stack, Environment, Control and Dump.

Continuations

- λ-expressions restricted to always return "()" [continuations] can implement all λ-expressions
- Continuations can also represent many forms of non-standard control flow, including exceptions
- call/cc

State

How can we use state and effects in a purely functional language?

Types

This course is primarily untyped.

We will mention types only where it aids understanding.

Pure λ-calculus

Syntax

Variables: $x, y, z, \ldots$

Terms:

$$M, N, L, \ldots \ ::= \ \lambda x.M \mid M\,N \mid x$$

We write $M \equiv N$ to say M and N are syntactically equal.

Free variables and permutation

We define free variables of a λ–term as

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

We define variable permutation as

- $x <x \cdot z> \ = \ x <z \cdot x> \ = \ z$
- $x <y \cdot z> \ = \ x$ (provided $x \neq y$ and $x \neq z$)
- $(\lambda x.M) <y \cdot z> \ = \ \lambda(x<y \cdot z>).(M <y \cdot z>)$
- $(M\ N) <y \cdot z> \ = \ (M<y \cdot z>)\ (N<y \cdot z>)$

Recap: Equivalence relations

An equivalence relation is a reflexive, symmetric and transitive relation.

R is an equivalence relation if

- Reflexive
  $$\forall x.\ \ x\ R\ x$$
- Transitive
  $$\forall xyz.\ \ x\ R\ y \wedge y\ R\ z \Rightarrow x\ R\ z$$
- Symmetric
  $$\forall xy.\ x\ R\ y \Rightarrow y\ R\ x$$

Congruence and Contexts

A congruence relation is an equivalence relation, that is preserved by placing terms under contexts.

Context (term with a single hole ($\bullet$)):

$$C ::= \lambda x.C \mid C\ M \mid M\ C \mid \bullet$$

Context application C[M] fills hole ($\bullet$) with M.

R is a compatible relation if

- $\forall M\ N\ C.\ \ \ M\ R\ N \ \Rightarrow\ C[M]\ R\ C[N]$

R is a congruence relation if it is both an equivalence and a compatible relation.

α-equivalence

Two terms are α–equivalent if they can be made syntactically equal ($\equiv$) by renaming bound variables

α–equivalence ($=_\alpha$) is the least congruence relation satisfying

- $\lambda x.\ M =_\alpha \lambda y.\ M<x \cdot y>$ where $y \notin FV(\lambda x.\ M)$

Intuition of α-equivalence

Consider

$\lambda x.\ \lambda y.\ x\ y\ z\ x$

We can see this as

$\lambda x.\ \lambda y.\ x\ y\ z\ x$

and hence the bound names are irrelevant

$\lambda \cdot .\ \lambda \cdot .\ \cdot\ \cdot\ z\ \cdot$

We only treat terms up to α−equivalence.

Are these alpha-equivalent?

$\lambda x.x =_\alpha \lambda y.y$	☐
$\lambda x.\lambda y.x =_\alpha \lambda y.\lambda x.y$	☐
$\lambda x.y =_\alpha \lambda y.y$	☐
$(\lambda x.x)\ (\lambda y.y) =_\alpha (\lambda y.y)\ (\lambda x.x)$	☐
$\lambda x.\ \lambda y.\ (x\ z\ y) =_\alpha \lambda z.\ \lambda y.\ (z\ z\ y)$	☐

α-equivalence (alternative defn)

Use $\lambda xs.M$ as a shorthand, where

- $xs ::= xs,x \mid []$
- $\lambda[].M \equiv M$
- $\lambda xs,x.M \equiv \lambda xs.\ \lambda x.\ M$

Definition

- $\lambda[].x =_\alpha \lambda[].x$
- $\lambda xs,x_1.x_2 =_\alpha \lambda ys,y_1.y_2$
  if $(x_1 \equiv x_2$ and $y_1 \equiv y_2)$
  or $(x_1 \not\equiv x_2$ and $y_1 \not\equiv y_2$ and $\lambda xs.x_2 =_\alpha \lambda ys.y_2)$
- $\lambda xs.\ M_1\ M_2 =_\alpha \lambda ys.\ N_1\ N_2$
  iff $\lambda xs.M_1 =_\alpha \lambda ys.N_1$ and $\lambda xs.M_2 =_\alpha \lambda ys.N_2$

Correction

Capture avoiding substitution

If $x \notin FV(M)$,

- $M\,[L/x] = M$

otherwise:

- $(M\ N)\,[L/x] = (M\,[L/x]\ \ N\,[L/x])$
- $(\lambda y.M)\,[L/x] = (\lambda z.\ M\langle z \cdot y\rangle[L/x])$
  where $z \notin FV(x, L, \lambda y.m)$
- $x\,[L/x] = L$

Note: In the $(\lambda y.M)$ case, we use a permutation to pick an α−equivalent term that does not capture variables in L.

$(x\ y)[L/y] = x\ L$

$(\lambda x.\ y)\ [x/w] = \lambda x.\ y$

$(\lambda x.\ (x\ y))\ [L/x] = (\lambda x.\ (x\ y))$

$(\lambda x.\ y)\ [x/y] = (\lambda z.x)$

$(\lambda y.\ (\lambda x.\ z))\ [x\ w/\ z] = (\lambda y.(\lambda x.\ (x\ w)))$

Extra brackets

To simplify terms we will drop some brackets:

$$\lambda xy.\ M \equiv \lambda x.\ (\lambda y.\ M)$$
$$L\ M\ N \equiv (L\ M)\ N$$
$$\lambda x.\ M\ N \equiv \lambda x.\ (M\ N)$$

Some examples

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)\ y\ z \equiv (((\lambda x.(x\ x))\ (\lambda x.(x\ x)))\ y)\ z$$
$$\lambda xyz.xyz \equiv \lambda x.(\lambda y.(\lambda z.\ ((xy)\ z)))$$

βη-reduction

We define β–reduction as:

$$(\lambda x.M)\ N \to_\beta M\ [N/x]$$

This is the workhorse of the λ–calculus.

We define η–reduction as: If $x \notin FV(M)$, then

$$\lambda x.\ (M\ x) \to_\eta M$$

This collapses trivial functions.

Consider (fn x=> sin x) is this the same as sin in ML?

βη examples

$(\lambda x.\ x\ y\ )\ (\lambda z.\ z) \to_\beta \lambda z.\ z\ y$

$(\lambda x.\ x\ y)\ (\lambda z.\ z) \to_\beta (\lambda z.\ z)\ y$

$\lambda x.\ M\ N\ x \to_\eta (M\ N)$

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \to_\beta (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$(\lambda xy.\ x)\ (\lambda x.\ y) \to_\beta (\lambda yx.\ y)$

Reduction in a context

We actually define β–reduction as:

$$C[\ (\lambda x.M)\ N\ ] \rightarrow_\beta C[\ M\ [N/x]\ ]$$

and η–reduction as:

$$C[\ (\lambda x.(M\ x))\ ] \rightarrow_\eta C[\ M\ ]\ \ (\text{where } x \notin FV(M))$$

where $C ::= \lambda x.C \mid C\ M \mid M\ C \mid \bullet$ (from "Context and Congruence" slide)

Note: to control evaluation order we can consider different contexts.

How many reductions?

$(\lambda x.x)\ ((\lambda x.x)\ (\lambda x.x))$	☐
$(\lambda x.x)\ (\lambda x.x)\ (\lambda x.x)$	☐
$((\lambda xy.x)\ z)\ w$	☐
$(\lambda xy.z)\ ((\lambda x.x)\ (\lambda x.x))\ (\lambda x.x)$	☐
$\lambda xy.\ z$	☐

Lecture 2

Normal-form (NF)

A term is in normal form if it there are no β or η reductions that apply.

Examples in NF:

- x; $\lambda x.y$; and $\lambda xy.\ x\ (\lambda x.y)$

and not in NF:

- $(\lambda x.x)\ y$; $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$; and $(\lambda x.\ y\ x)$

β–normal–form:

- $NF ::= \lambda x.\ NF \mid NF_2$
- $NF_2 ::= NF_2\ NF \mid x$

Normal-forms

A term has a normal form, if it can be reduced to a normal form:

- $(\lambda x.x)\ y$ has normal form y
- $(\lambda x.\ y\ x)$ has a normal form y
- $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ does not have a normal form

Note: $(\lambda x.xx)\ (\lambda x.xx)$ is sometimes denoted Ω.

Note: Some terms have normal forms and infinite reduction sequences, e.g. $(\lambda x.\ y)\ \Omega$.

Weak head normal form

A term is in WHNF if it cannot reduce when we restrict the context to

$$C ::= C\ M \mid M\ C \mid \bullet$$

That is, we don't reduce under a λ.

$\lambda x.\ \Omega$ is a WHNF, but not a NF.

Multi-step reduction

$M \rightarrow^* N$ iff

- $M \rightarrow_\beta N$
- $M \rightarrow_\eta N$
- $M \equiv N$ (reflexive)
- $\exists L.\ \ M \rightarrow^* L$ and $L \rightarrow^* N$ (transitive)

The transitive and reflexive closure of β and η reduction.

Equality

We define equality on terms, =, as the least congruence relation, that additional contains

- α–equivalence (implicitly)
- β–reduction
- η–reduction

Sometimes expressed as M=M' iff there exists a sequence of forwards and backwards reductions from M to M':

- $M \rightarrow^* N_1 {}^*\!\leftarrow M_1 \rightarrow^* N_2 {}^*\!\leftarrow \ldots \rightarrow^* N_k {}^*\!\leftarrow M'$

Exercise: Show these are equivalent.

Equality properties

If ($M \to^* N$ or $N \to^* M$), then $M = N$.
The converse is not true (Exercise: why?)

If $L \to^* M$ and $L \to^* N$, then $M = N$.

If $M \to^* L$ and $N \to^* L$, then $M = N$.

Church-Rosser Theorem

Theorem: If $M=N$, then there exists L such that $M \to^* L$ and $N \to^* L$.

Consider $(\lambda x.ax)((\lambda y.by)c)$:

- $\underline{(\lambda x.ax)((\lambda y.by)c)} \to_\beta a(\underline{(\lambda y.by)c}) \to_\beta a(bc)$
- $(\lambda x.ax)(\underline{(\lambda y.by)c}) \to_\beta (\lambda x.ax)\,(bc) \to_\beta a(bc)$

Note: Underlined term is reduced.

Consequences

If $M=N$ and N is in normal form, then $M \to^* N$.

If $M=N$ and M and N are in normal forms, then $M =_\alpha N$. Conversely, if M and N are in normal forms and are distinct, then $M \neq N$. For example, $\lambda xy.x \neq \lambda xy.y$.

Diamond property

Key to proving Church-Rosser Theorem is demonstrating the diamond property:

- If $M \to^* N_1$ and $M \to^* N_2$, then there exists L such that $N_1 \to^* L$ and $N_2 \to^* L$.

Exercise: Show how this property implies the Church-Rosser Theorem.

Proving diamond property

The diamond property does not hold for the single step reduction:

- If $M \rightarrow_\beta N_1$ and $M \rightarrow_\beta N_2$, then there exists L such that $N_1 \rightarrow_\beta L$ and $N_2 \rightarrow_\beta L$.

Proving diamond property

Consider $(\lambda x.xx)\ (I\ a)$ where $I = \lambda x.x$. This has two initial reductions:

- $(\lambda x.xx)\ \underline{(I\ a)} \rightarrow_\beta (\lambda x.xx)\ a \rightarrow_\beta a\ a$
- $\underline{(\lambda x.xx)\ (I\ a)} \rightarrow_\beta (I\ a)\ (I\ a)$

Now, the second has two possible reduction sequences:

- $(I\ a)\ (I\ a) \rightarrow_\beta a\ (I\ a) \rightarrow_\beta a\ a$
- $(I\ a)\ (I\ a) \rightarrow_\beta (I\ a)\ a \rightarrow_\beta a\ a$

Proving diamond property

Strip lemma:

- If $M \rightarrow_\beta N_1$ and $M \rightarrow^* N_2$, then there exists L such that $N_1 \rightarrow^* L$ and $N_2 \rightarrow^* L$

Proof: Tedious case analysis on reductions.

Note: The proof is beyond the scope of this course.

Reduction order

Consider $(\lambda x.a)\ \Omega$ this has two initial reductions:

- $\underline{(\lambda x.a)\ \Omega} \rightarrow_\beta a$
- $(\lambda x.a)\ \underline{\Omega} \rightarrow_\beta (\lambda x.a)\ \Omega$

Following first path, we have reached normal-form, while second is potentially infinite.

Normal order reduction

Perform leftmost, outermost β–reduction.
(leave η–reduction until the end)

Reduction context

- C ::= λx.C | C'
- C'::= C' M | NF_2 C | •

typo in printout

where NF_2 is from β–normal–form definition.

This definition is guaranteed to reach normal–form if one exists.

Other reduction orders

Call–by–name:

- C ::= C M | •

Call–by–value:

- V ::= x | λx. M (values)
- C ::= C M | • | (λx.M) C
- C[(λx.M) V] $\rightarrow_\beta$ C[M[V/x]]

typo in printout

Encoding Data

Motivation

We want to use different datatypes in the λ–calculus.

Two possibilities:

- Add new datatypes to the language
- Encode datatypes into the language

Encoding makes program language simpler, but less efficient.

Encoding booleans

To encode booleans we require IF, TRUE, and FALSE such that:

IF TRUE M N = M
IF FALSE M N = N

Here, we are using = as defined earlier.

Encoding booleans

Definitions:

- TRUE $\equiv \lambda$m n. m
- FALSE $\equiv \lambda$m n. n
- IF $\equiv \lambda$b m n. b m n

TRUE and FALSE are both in normal-form, so by Church-Rosser, we know TRUE$\neq$FALSE.

Note that, IF is not strictly necessary as

- $\forall$P. IF P = P (Exercise: show this).

Encoding booleans

Exercise: Show

- If L=TRUE then IF L M N = M.
- If L=FALSE then IF L M N = N.

Logical operators

We can give AND, OR and NOT operators as well:

- AND $\equiv \lambda$xy. IF x y FALSE
- OR $\equiv \lambda$xy. IF x TRUE y
- NOT $\equiv \lambda$x. IF x FALSE TRUE

Encoding pairs

Constructor:
- PAIR ≡ λxyf. fxy

Destructors:
- FST ≡ λp.p TRUE
- SND ≡ λp.p FALSE

Properties: ∀pq.
- FST (PAIR p q) = p
- SND (PAIR p q) = q

Encoding sums

Constructors:
- INL ≡ λx. PAIR TRUE x
- INR ≡ λx. PAIR FALSE x

Destructor:
- CASE = λs f g. IF (FST s) (f (SND s)) (g (SND s))

Properties:
- CASE (INL x) f g = f x
- CASE (INR x) f g = g x

Encoding sums (alternative defn)

Constructors:
- INL ≡ λx f g. f x
- INR ≡ λx f g. g x

Destructors:
- CASE ≡ λ s f g. s f g

As with booleans destructor unnecessary.
- ∀p. CASE p = p

Church Numerals

Define:
- $\underline{0}$ ≡ λf x . x
- $\underline{1}$ ≡ λf x. f x
- $\underline{2}$ ≡ λf x. f (f x)
- $\underline{3}$ ≡ λf x. f (f (f x))
- ...
- $\underline{n}$ ≡ λf x. f(...(f x)...)

That is, $\underline{n}$ takes a function and applies it n times to its argument: $\underline{n}$ f is f^n.

Arithmetic

typo in printout

Definitions
- ADD ≡ λmnfx. m f (n f x)
- MULT ≡ λmnfx. m (n f) x = λmnf. m (n f)
- EXP ≡ λmnfx. n m f x = λmn. n m

Example:
ADD $\underline{m}$ $\underline{n}$ $\rightarrow^*$ λf x. $\underline{m}$ f ($\underline{n}$ f x) $\rightarrow^*$ f^m (f^n x) ≡ f^{m+n} x

More arithmetic

Definitions
- SUC ≡ λn f x. f (n f x)
- ISZERO ≡ λn. n (λx.FALSE) TRUE

Properties
- SUC $\underline{n}$ = $\underline{n+1}$
- ISZERO $\underline{0}$ = TRUE
- ISZERO ($\underline{n+1}$) = FALSE

We also require decrement/predecessor!

Building decrement

n	PFN(n)
0	(0,0)
	suc I
1	(1,0)
	suc I
2	(2,1)
	suc I
3	(3,2)
	suc I
4	(4,3)

Decrement and subtraction

Definitions:
- PFN ≡ λn. n (λp. (SUC(FST P), FST P)) (PAIR $\underline{0}$ $\underline{0}$)
- PRE ≡ λn. SND (PFN n)
- SUB ≡ λmn. n PRE m

Exercise: Evaluate
- PFN $\underline{5}$
- PRE $\underline{0}$
- SUB $\underline{4}$ $\underline{6}$

Lists

Constructors:
- NIL ≡ PAIR TRUE (λz.z)
- CONS ≡ λxy. PAIR FALSE (PAIR x y)

Destructors:
- NULL ≡ FST
- HD ≡ λl. FST (SND l)
- TL ≡ λl. SND (SND l)

Properties:
- NULL NIL = TRUE
- HD (CONS M N) = M

Recursion

How do we actually iterate over a list?

Recursion

Fixed point combinator (Y)

We use a fixed point combinator Y to allow recursion.

In ML, we write:

letrec f(x) = M in N

this is really

let f = Y (λf.λx. M) in N

and hence

(λf.N) (Y λf. λx. M)

Defining recursive function

Consider defining a factorial function with the following property:

FACT = λn.(ISZERO n) $\underline{1}$ (MULT n (FACT (PRE n)))

We can define

PREFACT = λfn. (ISZERO n) $\underline{1}$ (MULT n (f (PRE n)))

Properties

- Base case: ∀F. PREFACT F $\underline{0}$ = $\underline{1}$
- Inductive case: ∀F. If F behaves like factorial up to n, then PREFACT F behaves like factorial up to n+1;

Fixed points

Discrete Maths: x is a fixed point of f, iff f x = x

Assume, Y exists (we will define it shortly) such that

- Y f = f (Y f)

Hence, by using Y we can satisfy this property:

FACT ≡ Y (PREFACT)

Exercise: Show FACT satisfies property on previous slide.

General approach

If you need a term, M, such that

- M = P M

Then M ≡ YP suffices

Example:

- ZEROES = CONS $\underline{0}$ ZEROES = (λp.CONS $\underline{0}$ p) ZEROES
- ZEROES ≡ Y (λp.CONS $\underline{0}$ p)

Mutual Recursion

Consider trying to find solutions M and N to:

- M = P M N
- N = Q M N

We can do this using pairs:

L ≡ Y(λp. PAIR (P (FST p) (SND p)) (Q (FST p) (SND p)))

M ≡ FST L

N ≡ SND L

Exercise: Show this satisfies equations given above.

Y

Definition (Discovered by Haskell B. Curry):

- $Y \equiv \lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))$

Properties

$YF \equiv (\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))) F$
$\rightarrow (\lambda x. F(xx)) (\lambda x. F(xx))$
$\rightarrow F ((\lambda x. F(xx)) (\lambda x.F(xx)))$
$\leftarrow F ((\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))) F) \equiv F(YF)$

There are other terms with this property:

- $(\lambda xy.xyx) (\lambda xy.xyx)$

(see wikipedia for more)

Y has no normal form

We assume:

- M has no normal form, iff M x has no normal form. (Exercise: prove this)

Proof of Y has no normal form:

- Y f = f (Y f) (by Y property)
- Assume Y f has a normal form N.
- Hence f (Y f) can reduce to f N, and f N is also a normal form.
- Therefore, by Church Rosser, f N ≡ N, which is a contradiction, so Y f cannot have a normal form.
- Therefore, Y has no normal form.

Head normal form

How can we characterise well-behaved λ-terms?

- Terms with normal forms? (Too strong, FACT does not have normal form)
- Terms with weak head normal form (WHNF)? (Too weak, lots of bad terms have this, for example $\lambda x.\Omega$).
- New concept: Head normal form.

HNF

A term is in head normal form, iff it looks like
$\lambda x_1 \ldots x_m.\ y\ M_1 \ldots M_k \quad (m,k \geq 0)$

Examples:

- x, $\lambda xy.x$, $\lambda z.z((\lambda x.a)c)$,
- $\lambda f. f (\lambda x. f(xx)) (\lambda x. f(xx))$

Non-examples:

- $\lambda y.(\lambda x.a)\ y \quad \rightarrow \lambda y.a$
- $\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))$

Properties

Head normal form can be reached by performing head reduction (leftmost)

- C' ::= C' M | •
- C ::= λx.C | C'

Therefore, Ω has no HNF (Exercise: prove this.)

If M N has a HNF, then so does M. Therefore, if M has no HNF, then M N_1 ... N_k does not have a HNF. Hence, M is a "totally undefined function".

ISWIM

λ-calculus as a programming language
(The next 700 programming languages [Landin 1966])

ISWIM: Syntax

From the λ–calculus

- x (variable)
- λx.M (abstraction)
- M N (application)

Local declarations

- let x = M in N (simple declaration)
- let f x_1 ... x_n = M in N (function declaration)
- letrec f x_1 ... x_n = M in N (recursive declaration)

and post–hoc declarations

- N where x = M
- ...

ISWIM: Syntactic sugar

N where x=M	$\equiv$	let x = M in N
let x = M in N	$\equiv$	(λx.N) M
let f x_1...x_n =M in N	$\equiv$	let f = $\lambda x_1...x_n$.M in N
letrec f x_1...x_n =M in N	$\equiv$	let f = Y(λf.$\lambda x_1...x_n$.M) in N

Desugaring explains syntax purely in terms of λ–calculus.

ISWIM: Constants

M ::= x | c | λx.M | M N

Constants c include:

- 0 1 −1 2 −2 ... (integers)
- + − x / (arithmetic operators)
- = ≠ < > (relational operators)
- true false (booleans)
- and or not (boolean connectives)

Reduction rules for constants: e.g.

- $+\ 0\ 0 \rightarrow_\delta 0$

Call-by-value and IF-THEN-ELSE

ISWIM uses the call−by−value λ−calculus.

Consider: IF TRUE 1 Ω

IF E THEN M ELSE N ≡ (IF E (λx.M) (λx.N)) (λz.z)
where $x \notin FV(M\ N)$

Pattern matching

Has

- (M,N) (pair constructor)
- $\lambda(p_1,p_2)$. M (pattern matching pairs)

Desugaring

- $\lambda(p_1,p_2).\ M \equiv \lambda z.(\lambda p_1 p_2.\ M)\ (\mathrm{fst}\ z)\ (\mathrm{snd}\ z)$

where $z \notin FV(M)$

Real λ-evaluator

Don’t use β and substitution

Do use environment of values, and delayed substitution.

Environments and Closures

Consider β–reduction sequence

$$(\lambda xy.x + y)\ 3\ 5 \rightarrow (\lambda y.3 + y)\ 5 \rightarrow 3 + 5 \rightarrow 8.$$

Rather than produce $(\lambda y.3+y)$ build a closure:

$$\text{Clo}(\quad y \quad , \quad x+y \quad , \quad x=3)$$

The arguments are

- bound variable;
- function body; and
- environment.

SECD Machine

Virtual machine for ISWIM.

The SECD machine has a state consisting of four components S, E, C and D:

- S: The “stack” is a list of values typically operands or function arguments; it also returns result of a function call;
- E: The “environment” has the form $x_1=a_1;\ldots;x_n=a_n$, expressing that the variables $x_1,\ldots,x_n$ have values $a_1\ldots a_n$ respectively; and
- C: The “control” is a list of commands, that is λ–terms or special tokens/instructions.

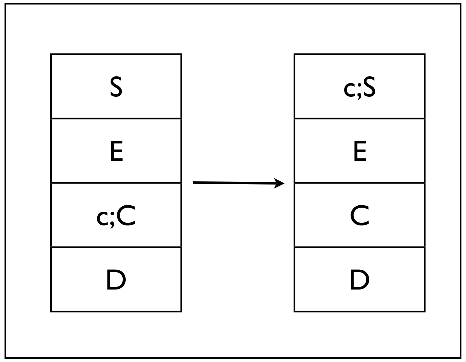
SECD Machine

- D: The “dump” is either empty (–) or is another machine state of the form (S,E,C,D). A typical state looks like

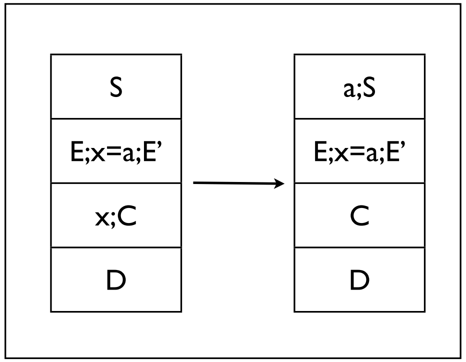
$$(S_1,E_1,C_1,(S_2,E_2,C_2,\ldots(S_n,E_n,C_n,-)\ldots))$$

It is essentially a list of triples $(S_1,E_1,C_1),\ldots,(S_n,E_n,C_n)$ and serves as the function call stack.

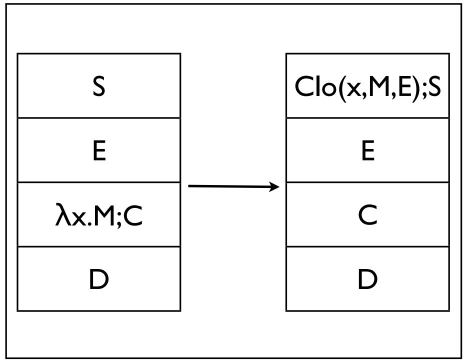
State transitions: constant



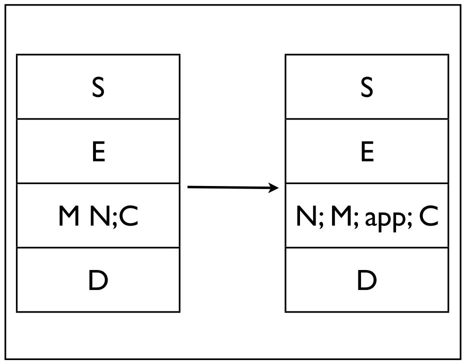
State-transition: variable



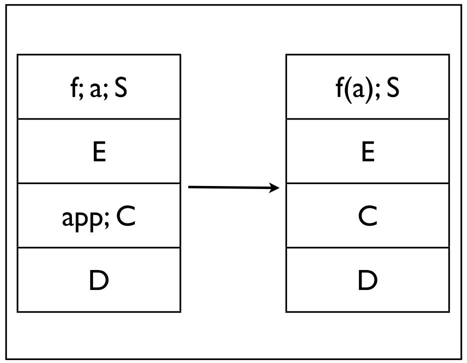
State-transition: function



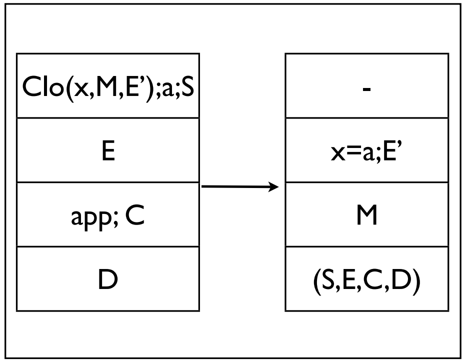
State-transition: application



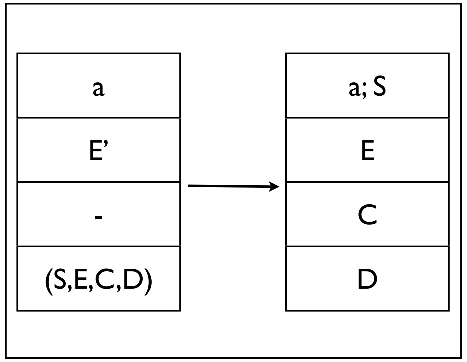
State-transition: app primitive



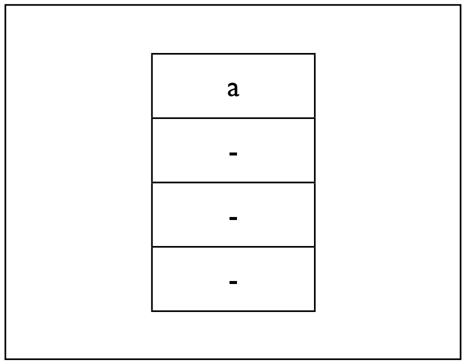
State-transition: app closure



State-transition: return



Final configuration



Compiled SECD machine

Inefficient as requires construction of closures.
Perform some conversions in advance:

- $[\![c]\!] \equiv$ const c
- $[\![x]\!] \equiv$ var x
- $[\![M\ N]\!] \equiv [\![N]\!] ; [\![M]\!]$; app
- $[\![\lambda x.M]\!] \equiv$ Closure(x,$[\![M]\!]$)
- $[\![M + N]\!] \equiv [\![M]\!] ; [\![N]\!]$; add
- ...

More intelligent compilations for “let” and tail recursive functions can also be constructed.

Example

We can see $((\lambda xy.x + y)\ 3\ )\ 5$ compiles to

- const 5; const 3; Closure(x,C_0); app; app

where

- $C_0 \equiv$ Closure(y,C_1)
- $C_1 \equiv$ var x; var y; add

Recursion

The usual fixpoint combinator fails under the SECD machine: it loops forever.

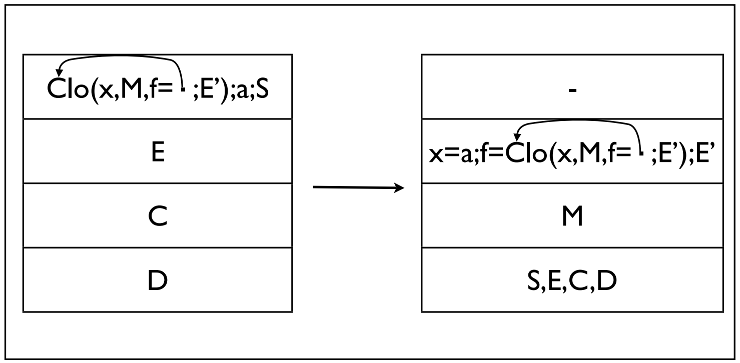
A modified one can be used:

- λfx. f (λy. x x y) (λy.x x y)

This is very inefficient.

Better approach to have closure with pointer to itself.

Recursive functions (Y(λfx.M))



Implementation in ML

SECD machine is a small-step machine.

Next we will see a big-step evaluator written in ML.

Implementation in ML

```
datatype Expr = Name of string
  | Numb of int
  | Plus of Expr  * Expr
  | Fn of string * Expr
  | Apply of Expr * Expr

datatype Val =
    IntVal of val
  | FnVal of string * Expr * Env

and Env = Empty | Defn of string * Val * Env
```

Implementation in ML

```
fun lookup (n, Defn (s,v,r)) =
    if s=n then v else lookup(n,r)
  | lookup(n, Empty)  = raise oddity()
```

Implementation in ML

```
fun eval (Name(s), r) = lookup(s,r)
  | eval(Fn(bv,body),r) = FnVal(bv,body,r)
  | eval(Apply(e,e'), r) =
    case eval(e,r)
      of IntVal(i) => raise oddity()
        | FnVal(bv,body,env) =>
          let val arg = eval(e',r) in
          eval(body, Defn(bv,arg,env))
    ...
```

Exercises

How could we make it lazy?

Combinators

Combinator logic

Syntax:

$$P,Q,R ::= S \mid K \mid P\,Q$$

Reductions:

$$K\,P\,Q \rightarrow_w P$$
$$S\,P\,Q\,R \rightarrow_w (P\,R)\,(Q\,R)$$

Note that the term S K does not reduce: it requires three arguments. Combinator reductions are call "weak reductions".

Identity combinator

Consider the reduction of, for any P

- $S\,K\,K\,P \rightarrow_w K\,P\,(K\,P) \rightarrow_w P$

Hence, we define $I \equiv S\,K\,K$, where I stands for identity.

Church-Rosser

Combinators also satisfy Church-Rosser:

- if $P = Q$, then exists R such that $P \rightarrow_w^* R$ and $Q \rightarrow_w^* R$

Encoding the λ-calculus

Use extended syntax with variables:

- P ::= S | K | P P | x

Define meta-operator on combinators λ* by

- $\lambda^*x.x \equiv I$
- $\lambda^*x.P \equiv K\ P$ (where $x \notin FV(P)$)
- $\lambda^*x.P\ Q \equiv S\ (\lambda^*x.P)\ (\lambda^*x.Q)$

Example translation

$(\lambda^*x.\lambda^*y.\ y\ x)$
$\equiv \lambda^*x.\ S\ (\lambda^*y.y)\ (\lambda^*y.x)$
$\equiv \lambda^*x.\ (S\ I)\ (K\ x)$
$\equiv S\ (\lambda^*x.(S\ I))\ (\lambda^*x.K\ x)$
$\equiv S\ (K\ (S\ I))\ (S\ (\lambda^*x.K)\ (\lambda^*x.x))$
$\equiv S\ (K\ (S\ I))\ (S\ (K\ K)\ I)$

There and back again

λ–calculus to SK:

- $(\lambda x.M)_{CL} = (\lambda^*x.\ (M)_{CL})$
- $(x)_{CL} = x$
- $(M\ N)_{CL} = (M)_{CL}\ (N)_{CL}$

SK to λ–calculus:

- $(x)_\lambda = x$
- $(K)_\lambda = \lambda xy.x$
- $(S)_\lambda = \lambda xyz.\ x\ z\ (y\ z)$
- $(P\ Q)_\lambda = (P)_\lambda\ (Q)_\lambda$

Properties

Free variables are preserved by translation

- $FV(M) = FV(\ (M)_{CL}\ )$
- $FV(P) = FV(\ (P)_\lambda\ )$

Supports α and β reduction:

- $(\lambda^*\ x.P)\ Q \rightarrow_w^*\ P\,[Q/x]$
- $(\lambda^*\ x.P) \equiv \lambda^*y.\ P\langle y \cdot x\rangle$ (where $y \notin FV(P)$)

Equality on combinators

Combinators don't have an analogue of the η-reduction rule.

- $(SK)_\lambda = (KI)_\lambda$, but SK and KI are both normal forms

To define equality on combinators, we take the least congruence relation satisfying:

- weak reductions, and
- functional extensionality: If $P\ x = Q\ x$, then $P = Q$ (where $x \notin FV(P\ Q)$).

$$S\ K\ x\ y \rightarrow (K\ y)\ (K\ x) \rightarrow y \leftarrow I\ y \leftarrow K\ I\ x\ y$$

Therefore, $SK = KI$.

Properties

We get the following properties of the translation:

- $((M)_{CL})_\lambda = M$
- $((P)_\lambda)_{CL}) = P$
- $M{=}N \Leftrightarrow (M)_{CL} = (N)_{CL}$
- $P{=}Q \Leftrightarrow (P)_\lambda = (Q)_\lambda$

Aside: Hilbert style proof

In Logic and Proof you covered Hilbert style proof:

- Axiom K: $\forall AB.\ A \rightarrow (B \rightarrow A)$
- Axiom S: $\forall ABC.\ (A{\rightarrow}(B{\rightarrow}C)) \rightarrow ((A{\rightarrow}B) \rightarrow (A{\rightarrow}C))$
- Modus Ponens : If $A \rightarrow B$ and A, then B

Hilbert style proofs correspond to "Typed" combinator terms:

- $S\ K : \forall AB.\ ((A \rightarrow B) \rightarrow (A \rightarrow A))$
- $S\ K\ K : \forall A.\ (A \rightarrow A)$

Logic, Combinators and the λ-calculus are carefully intertwined. See Types course for more details.

Compiling with combinators

The translation given so far is exponential in the number of lambda abstractions.

Add two new combinators

- $B\ P\ Q\ R \rightarrow_w P\ (Q\ R)$
- $C\ P\ Q\ R \rightarrow_w P\ R\ Q$

Exercise: Encode B and C into just S and K.

Advanced translation

- $\lambda^T x.x \equiv I$
- $\lambda^T x.P \equiv KP \quad (x \notin FV(P))$
- $\lambda^T x.Px \equiv P \quad (x \notin FV(P))$
- $\lambda^T x.PQ \equiv B\, P(\lambda^T x.Q) \quad (x \notin FV(P) \text{ and } x \in FV(Q))$
- $\lambda^T x.PQ \equiv C(\lambda^T x.P)Q \quad (x \in FV(P) \text{ and } x \notin FV(Q))$
- $\lambda^T x.PQ \equiv S(\lambda^T x.P)(\lambda^T x.Q) \quad (x \in FV(P), x \in FV(Q))$

(Invented by David Turner)

Example

$(\lambda^T x.\lambda^T y.\ y\ x)$
$\equiv (\lambda^T x.C\ (\lambda^T y.\ y)\ x)$
$\equiv (\lambda^T x.C\ I\ x)$
$\equiv C\ I$

Compared to $(\lambda^* x.\lambda^* y.\ y\ x) \equiv S\ (K\ (S\ I))\ (S\ (K\ K)\ I)$

Translation with λ^* is exponential, while λ^T is only quadratic.

Example

$\lambda^T f.\lambda^T x.\ f\ (x\ x)$
$\equiv \lambda^T f.\ B\ (f\ (\lambda^T x.x\ x))$
$\equiv \lambda^T f.\ B\ (f\ (S\ (\lambda^T x.x)\ (\lambda^T x.\ x)))$
$\equiv \lambda^T f.\ B\ (f\ (S\ I\ I))$
$\equiv B\ B\ (\lambda^T f.\ f\ (S\ I\ I))$
$\equiv B\ B\ (C\ (\lambda^T f.\ f)\ (S\ I\ I))$
$\equiv B\ B\ (C\ I\ (S\ I\ I))$

Combinators as graphs

To enable lazy reduction, consider combinator terms as graphs.

S reduction creates two pointers to the same subterm.

Typo

Let's consider

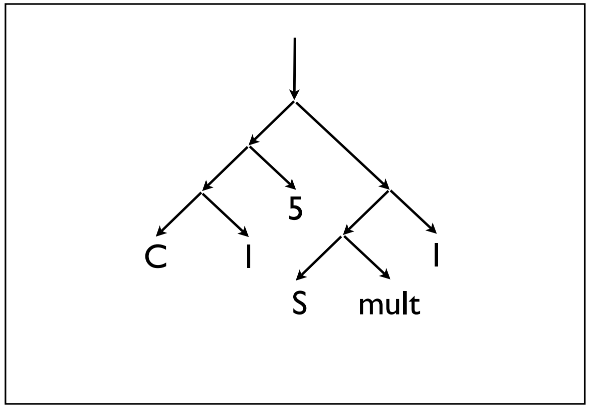
- let sqr x=mult x x in sqr 5 $\equiv (\lambda f.\ f\ 5)(\lambda m.\ \text{mult}\ m\ m)$

this translates to

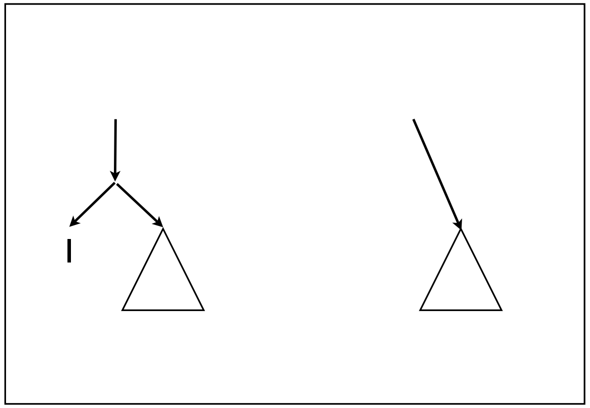
- $C\ I\ 5\ (S\ \text{mult}\ I)$

Exercise: Show this translation.

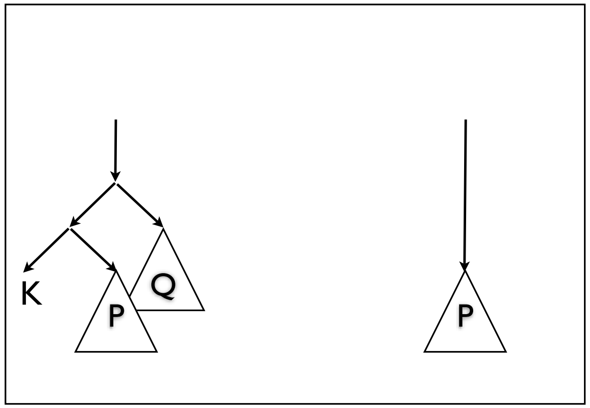
C I 5 (S mult I)



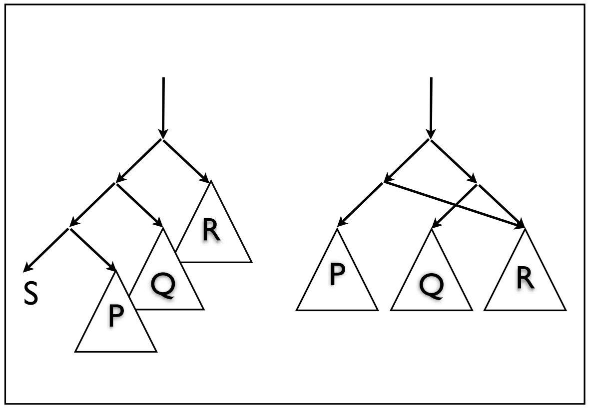
Reduction: I



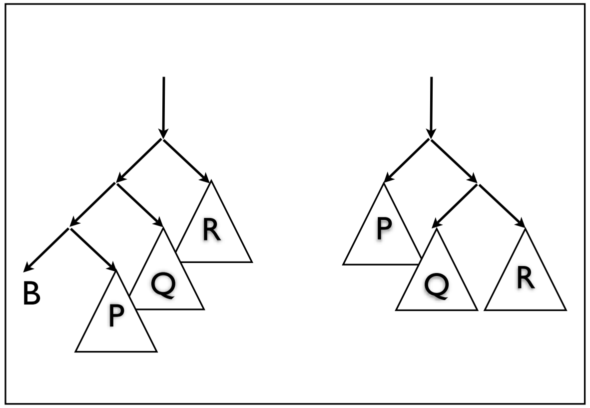
Reduction: K



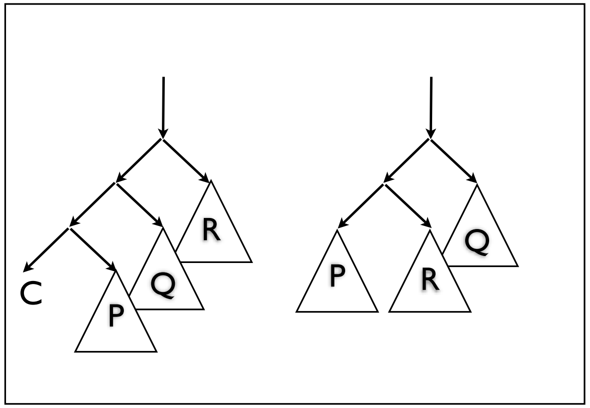
Reduction: S



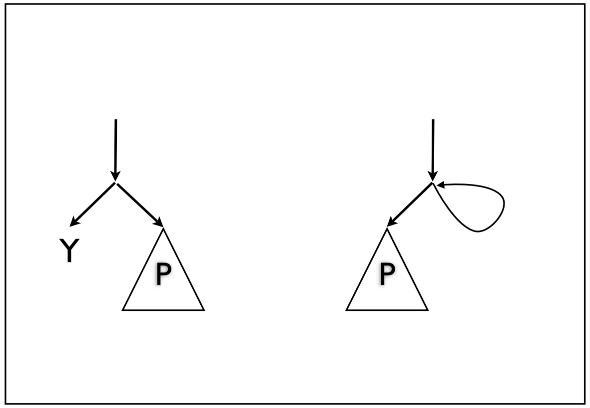
Reduction: B



Reduction: C



Recursion



Comments

If 5 was actually a more complex calculation, would only have to perform it once.

Lazy languages such as Haskell, don't use this method.

Could we have done graphs of λ-terms? No. Substitution messes up sharing.

Example using recursion in Paulson's notes.

Types

Simply typed λ-calculus

Types

$$\tau ::= \text{int} \mid \tau \to \tau$$

Syntactic convention

$$\tau_1 \to \tau_2 \to \tau_3 \equiv \tau_1 \to (\tau_2 \to \tau_3)$$

Simplifies types of curried functions.

Type checking

We check

- $M\,N : \tau$ iff $\exists \tau'.\ M : \tau' \to \tau$ and $N : \tau'$
- $\lambda x.\ M : \tau \to \tau'$ iff $\exists \tau.$ if $x{:}\tau$ then $M : \tau'$
- $n : \text{int}$

Semantics course covers this more formally, and types course next year in considerably more detail.

Type checking

$\lambda x.\ x \ : \text{int} \to \text{int}$

$\lambda x\ f.\ f\ x : \text{int} \to (\text{int} \to \text{int}) \to \text{int}$

$\lambda fgx.\ fgx : (\tau_1 \to \tau_2) \to (\tau_2 \to \tau_3) \to \tau_1 \to \tau_3$

$\lambda fgx.\ f(gx) : (\tau_1 \to \tau_2) \to (\tau_2 \to \tau_3) \to \tau_1 \to \tau_3$

$\lambda f\ x.\ f\ (f\ x) : (\tau \to \tau) \to \tau \to \tau$

Types help find terms

Consider type $(\tau_1 \to \tau_2 \to \tau_3) \to \tau_2 \to \tau_1 \to \tau_3$

Term λf.M where f : $(\tau_1 \to \tau_2 \to \tau_3)$ and M : $\tau_2 \to \tau_1 \to \tau_3$

Therefore M $\equiv$ λxy . N where x:τ_2, y:τ_1 and N:τ_3.

Therefore N $\equiv$ f y x

Therefore λfxy. f y x : $(\tau_1 \to \tau_2 \to \tau_3) \to \tau_2 \to \tau_1 \to \tau_3$

Polymorphism and inference

ML type system supports polymorphism:

$$\tau \ ::= \ \alpha \ | \ \forall\alpha.\,\tau \ | \ \ldots$$

Types can be inferred using unification.

Continuations

Overview

Encode evaluation order.

Encode control flow commands: for example Exit, exceptions, and goto.

Enables backtracking algorithms easily.

Key concept:

- don’t return, pass result to continuation. (This is what you did with the MIPS JAL (Jump And Link.) instruction.)

Call-by-value

Definition:

1. $[\![x]\!]_v(k) \equiv k\ x$
2. $[\![c]\!]_v(k) \equiv k\ c$
3. $[\![\lambda x.M]\!]_v(k) \equiv k\ (\lambda(x,k').\ [\![M]\!](k'))$
4. $[\![M\ N]\!]_v(k) \equiv [\![M]\!]\ (\lambda m.\ [\![N]\!]\ (\lambda n.\ m\ (n,k)))$

Intuition:

- $[\![M]\!](k)$ means evaluate M and then pass the result to k.
- k is what to do next.

Pairs not essential, but make the translation simpler.

Example: CBV

$[\![\lambda x.y]\!]_v\ (k)$
$\equiv k\ (\lambda(x,k').\ [\![y]\!]\ (k'))$
$\equiv k\ (\lambda(x,k').\ k'\ y)$

$[\![(\lambda x.y)\ z]\!]_v(k)$
$\equiv [\![\lambda x.y]\!](\ \lambda m.\ [\![z]\!]\ (\lambda n.\ m\ (n,k)))$
$\equiv [\![\lambda x.y]\!](\ \lambda m.\ (\lambda n.\ m(n,k))\ z)$
$\equiv (\lambda m.\ (\lambda n.\ m(n,k))\ z)\ (\lambda(x,k').\ k'\ y)$
$\rightarrow (\lambda n.\ (\lambda(x,k').\ k'\ y))\ (n,k))\ z$
$\rightarrow (\lambda(x,k').\ k'\ y))\ (z,k)$
$\rightarrow k\ y$

Call-by-name

Definition:

- $[\![x]\!]_n(k) \equiv x\ k$
- $[\![c]\!]_n(k) \equiv k\ c$
- $[\![\lambda x.M]\!]_n(k) \equiv k\ (\lambda(x,k').\ [\![M]\!](k'))$
- $[\![M\ N]\!]_n(k) \equiv [\![M]\!]\ (\lambda m.\ m\ (\lambda k'.[\![N]\!](k'),k))$

Only application and variable are different. Don't have to evaluate N before putting it into M.

Mistake

CBN and CBV

For any closed term M (FV(M) = {})

- M terminates with value v in the CBV λ-calculus, iff $[\![M]\!]_v(\lambda x.x)$ terminates in both the CBV and CBN λ-calculus with value v.
- M terminates with value v in the CBN λ-calculus, iff $[\![M]\!]_n(\lambda x.x)$ terminates in both the CBV and CBN λ-calculus with value v.

Encoding control

Consider trying to add an Exit instruction to the λ-calculus.

- Exit M → Exit (CBN and CBV)
- M Exit → Exit (Just CBV)

When we encounter Exit execution is stopped.

- (λx.y) Exit = Exit (CBV)
- (λx.y) Exit = y (CBN)

Encode as

- ⟦ Exit ⟧(k) = () (Both CBV and CBN)

Example CBV

$[\![(\lambda x.y)\ \mathrm{Exit}]\!]_v(k)$

$\equiv\ [\![\lambda x.y]\!]_v\ (\lambda m.\ [\![\mathrm{Exit}]\!]_v\ (\lambda n.\ m\ (n,k)))$

$\equiv\ [\![\lambda x.y]\!]_v\ (\lambda m.\ ())$

$\equiv\ (\lambda m.\ ())\ (\lambda(x,k').\ k'\ y)$

$\rightarrow\ ()$

Example CBN

$[\![(\lambda x.y)\ \mathrm{Exit}]\!]_n(k)$

$\equiv\ [\![\lambda x.y]\!]_n\ (\lambda m.\ m\ (\lambda k'.[\![\mathrm{Exit}]\!]_n(k'),k))$

$\equiv\ (\lambda m.\ m\ (\lambda k'.[\![\mathrm{Exit}]\!]_n(k'),k))\ (\lambda(x,k').\ y\ k')$

$\rightarrow\ (\lambda(x,k').\ y\ k')\ (\lambda k'.[\![\mathrm{Exit}]\!]_n(k'),k)$

$\rightarrow\ y\ k$

Order of evaluation

With CBV we can consider two orders of evaluation:

Function first:

$$[\![\ M\ N\ ]\!]_{v2}(k)\ \equiv\ [\![\ M\ ]\!]\ (\lambda m.\ [\![\ N\ ]\!]\ (\lambda n.\ m\ (n,k)))$$

Argument first:

$$[\![\ M\ N\ ]\!]_{v1}(k)\ \equiv\ [\![\ N\ ]\!]\ (\lambda n.\ [\![\ M\ ]\!]\ (\lambda m.\ m\ (n,k)))$$

Example

Consider having two Exit expressions

- $[\![\text{Exit}_1]\!] (k) = 1$
- $[\![\text{Exit}_2]\!] (k) = 2$

Now, we can observe the two different translations by considering $\text{Exit}_1\ \text{Exit}_2$:

- $[\![\text{Exit}_1\ \text{Exit}_2]\!]_{v1}(k) = [\![\text{Exit}_1]\!]_{v2}(k)$ (Function first)
- $[\![\text{Exit}_1\ \text{Exit}_2]\!]_{v2}(k) = [\![\text{Exit}_2]\!]_{v2}(k)$ (Argument first)

Example (continued)

$[\![\text{Exit}_1\ \text{Exit}_2]\!]_{v1}(k)$

$\equiv [\![\text{Exit}_1]\!]\ (\lambda m.\ [\![\text{Exit}_2]\!]\ (\lambda n.\ m\ (n,k))$

$\equiv 1 \equiv [\![\text{Exit}_1]\!](k)$

$[\![\text{Exit}_1\ \text{Exit}_2]\!]_{v2}(k)$

$\equiv [\![\text{Exit}_2]\!]\ (\lambda n.\ [\![\text{Exit}_1]\!]\ (\lambda m.\ m\ (n,k)))$

$\equiv 2 \equiv [\![\text{Exit}_2]\!](k)$

Typed translation: CBV

Consider types:

$$\tau ::= b \mid \tau \rightarrow \tau \mid \perp$$

Here b is for base types of constants, $\perp$ for continuation return type.

We translate:

- $[\![\tau_1 \rightarrow \tau_2]\!]_v \equiv ([\![\tau_1]\!]_v * ([\![\tau_2]\!]_v \rightarrow \perp)) \rightarrow \perp$
- $[\![b]\!]_v \equiv b$

If $M : \tau$ then $\lambda k.\ [\![M]\!]_v(k) : ([\![\tau]\!]_v \rightarrow \perp) \rightarrow \perp$

Sometimes, we write $T\ \tau$ for $(\tau \rightarrow \perp) \rightarrow \perp$

Types guide translation

For function translation: Assume

- $k : ([\![\tau_1 \rightarrow \tau_2]\!]_v \rightarrow \perp)$
- $\lambda x.M : \tau_1 \rightarrow \tau_2$, hence $[\![M]\!]_v : ([\![\tau_2]\!]_v \rightarrow \perp) \rightarrow \perp$ if $x : [\![\tau_1]\!]_v$

Find N such that $k\ N : \perp$ therefore $N : [\![\tau_1 \rightarrow \tau_2]\!]_v$

So, $N \equiv \lambda(x,k').\ L$, where $L : \perp$ if

- $x : [\![\tau_1]\!]_v$ and
- $k' : [\![\tau_2]\!]_v \rightarrow \perp$

Therefore $L \equiv [\![M]\!]_v(k')$

$$[\![\lambda x.M]\!]_v(k) \equiv k\ (\lambda(x,k').\ [\![M]\!](k')\ )$$

Types guide translation

Application translation (MN): Assume

- k : $(\llbracket \tau_2 \rrbracket_v \to \bot)$
- M : $\tau_1 \to \tau_2$, hence $\llbracket M \rrbracket_v : (\llbracket \tau_1 \to \tau_2 \rrbracket_v \to \bot) \to \bot$
- N : τ_1, hence $\llbracket N \rrbracket_v : (\llbracket \tau_1 \rrbracket_v \to \bot) \to \bot$

Find L such that $\llbracket M \rrbracket_v L : \bot$ therefore $L : \llbracket \tau_1 \to \tau_2 \rrbracket_v \to \bot$

So, $L \equiv \lambda m.\ L_1$, where $L_1 : \bot$ if $m : \llbracket \tau_1 \to \tau_2 \rrbracket_v \equiv \llbracket \tau_1 \rrbracket_v * (\llbracket \tau_2 \rrbracket_v \to \bot) \to \bot$

Find L_2 such that $\llbracket N \rrbracket_v L_2 : \bot$ therefore $L_2 : \llbracket \tau_1 \rrbracket_v \to \bot$

Therefore $L_2 \equiv \lambda n.\ L_3$ where $L_3 : \bot$ if $n : \llbracket \tau_1 \rrbracket_v$.

Therefore $L_3 \equiv m\ (n,k)$

$$\llbracket M\ N \rrbracket_v(k) \equiv \llbracket M \rrbracket\ (\lambda m.\ \llbracket N \rrbracket\ (\lambda n.\ m\ (n,k)))$$

Other encodings

We can encode other control structures:

- Exceptions (2 continuations: normal and exception)
- Breaks and continues in loops (3 continuations: normal, break, and continue)
- Goto, jumps and labels
- call/cc (passing continuations into programs)
- backtracking

Exercises

- Find an example that evaluates differently for each of the three encodings, and demonstrate this.
- How would you perform a type call-by-name translation?

$$\llbracket \tau_1 \to \tau_2 \rrbracket_n \equiv ((T\ \llbracket \tau_1 \rrbracket_n)^*\ (\llbracket \tau_2 \rrbracket_n \to \bot)) \to \bot$$

Aside: backtracking

Continuations can be a powerful way to implement backtracking algorithms. (The following is due to Olivier Danvy.)

Consider implementing regular expression pattern matcher in ML:

```
datatype re =
   Char of char        (*  "c"  *)
 | Seq of re * re      (*  re1 ; re2  *)
 | Alt of re * re      (*  re1 | re2  *)
 | Star of re * re     (*  re1 *   *)
```

Implementation

Plan: use continuations to enable backtracking:

```
fun
   f (“c”) (a::xs) k =   if a=c then (k xs) else false
 | f (“c”)  [] k  = false
 | f (re1 ; re2) xs k =  f re1 xs (λys. f re2 ys k)
 | f (re1 | re2) xs k = (f re1 xs k) orelse (f re2 xs k)
 | f (re1 *) xs k =
          (k xs) orelse (f (re1 ; re1*) xs k
```

Exercise: execute

f ((“a” | “a”; “b”; “c”); “b”) [“a”, “b”, “c”] (λxs. xs=[])

Example execution

```
f ((“a”; “a” | “a”); “a”)  [“a”, “a”] (λxs. xs=[])
→ f (“a”; “a” | “a”)  [“a”, “a”] (λxs. f “a” xs (λxs.xs=[]))
→ f  (“a”; “a”)  [“a”, “a”] (λxs. f “a” xs (λxs.xs=[]))
   orelse f “a”  [“a”, “a”] (λxs. f “a” xs (λxs.xs=[]))
→ (λxs. f “a” xs (λxs.xs=[])) []
   orelse f “a”  [“a”, “a”] (λxs. f “a” xs (λxs.xs=[]))
→ false
   orelse f “a”  [“a”, “a”] (λxs. f “a” xs (λxs.xs=[]))
→ f “a”  [“a”, “a”] (λxs. f “a” xs (λxs.xs=[]))
→ (λxs. f “a” xs (λxs.xs=[]))  [“a”]
→ (λxs.xs=[]) []  →   true
```

Exercise

How could you extend this to

- count the number of matches; and
- allow matches that don’t consume the whole string?

Remove use of orelse by building a list of continuations for backtracking.

Comments

Not the most efficient regular expression pattern matching, but very concise code.

This style can implement efficient lazy pattern matchers or unification algorithms.

State

Encoding state

Now, we can consider extending the λ-calculus with
- Assignment M := N
- Read !M

How can we do this by encoding?

ML Program

```
val a = ref 1;
fun g(x) = (a := (!a)*2; x+1)
fun h(y) = (a := (!a)+3; y*2)
print g(1) + h(3) + !a
```

```
fun g(x,w) = (x+1,w*2)
fun h(y,w) = (y*2,w + 3)
val w0 = 1
val (g',w1) = g(1,w0)
val (h',w2) = h(3,w1)
print g' + h' + w2
```

Comments

Evaluation order made explicit (CPS transform).

Parameter used to carry state around.

We use the following encoding of state functions,
- SET s x y = λz. IF z=x THEN y ELSE s x
- GET s x = s x

Note that, we ignore allocation in this encoding.

CPS and State

Replaced by CBV on next slide.
(This slide is not examinable)

Definition: (This is a CBN translation.)

- $[\![x]\!]_n(k,s) \equiv x\,(k,s)$
- $[\![c]\!]_n(k,s) \equiv k\,(c,s)$
- $[\![\lambda x.M]\!]_n(k,s) \equiv k\,((\lambda(x,k',s).\ [\![M]\!](k',s)\ ),s)$
- $[\![M\ N]\!]_n(k,s) \equiv [\![M]\!]\,((\lambda(m,s').\ m\,([\![N]\!],k,s')),s)$
- $[\![!M]\!]_n(k,s) \equiv [\![M]\!]((\lambda(v,s').\ k\,(GET\ s'\ v,\ s')),\ s)$
- $[\![[M]:=N]\!]_n(k,s) \equiv [\![M]\!](\lambda(v,s').\ k\,((),SET\ s'\ v\ [\![N]\!]),s)$

CPS and State

Definition: (This is a CBV translation.)

- $[\![x]\!]_n(k,s) \equiv k\,(x,s)$
- $[\![c]\!]_n(k,s) \equiv k\,(c,s)$
- $[\![\lambda x.M]\!]_n(k,s) \equiv k\,((\lambda(x,k',s').\ [\![M]\!](k',s')\ ),\ s)$
- $[\![M\ N]\!]_n(k,s) \equiv$
  $[\![M]\!]\,((\lambda(m,s').\ [\![N]\!](\lambda(n,s'').\ m(n,k,s''),\ s'),s)$
- $[\![!M]\!]_n(k,s) \equiv [\![M]\!](\lambda(v,s').\ (k\,(GET\ s'\ v,\ s'),\ s)$
- $[\![[M]:=N]\!]_n(k,s) \equiv$
  $[\![M]\!](\lambda(v,s').\ ([\![N]\!](\lambda(v',s'').k((),SET\ s''\ v\ v'),s'),s)$

Let notation

We can simplify the presentation by using let notation:

- $[\![M]\!]\,((\lambda(m,s').\ [\![N]\!](\lambda(n,s'').\ m(n,k,s''),\ s'),s)$

would be

$let_{CPS}\ (m,s') = [\![M]\!](s)$ in
$let_{CPS}\ (n,s'') = [\![N]\!](s')$ in $m(n,k,s'')$

where

$let_{CPS}\ (m,s') = [\![M]\!](s)$ in T

means

$[\![M]\!](\lambda(m,s').\ T,\ s)$

Note: This differs from the use of "let" in ISWIM.

Exercises

- Extend encoding with sequential composition M;N
- Translate: [x]:=1; !x
- Translate: (λy.z)([x]:=(!x+1))
- Alter translation for CBV (assignment, application and variable cases need changing.)
- Redo translations above.

Mistake

It's getting complicated

Common theme, we are threading "stuff" through the evaluation:

- continuations
- state

If we add new things, for example IO and exceptions, we will need even more parameters.

Can we abstract the idea of threading "stuff" through evaluation?

Monad (Haskell)

Haskell provides a syntax and type system for threading "effects" through code.

Two required operations

- return : $\tau \to T\,\tau$
- >>= : $T\,\tau \to (\tau \to T\,\tau') \to T\,\tau'$ [bind]

Option/Maybe Monad

Types

- Option τ

Definition

- Option $\tau \equiv$ unit + τ

Operations

- return : $\tau \to$ Option τ

  Typo

  return M $\equiv$ Some M
- >>= : Option $\tau \to (\tau \to$ Option $\tau') \to$ Option τ'

  λxy. case x of None => None | Some z => y z

Example

Imagine findx and findy are of type unit $\to$ Option τ

```
findx() >>= λx.
findy() >>= λy.
return (x,y)
```

This code is of type Option $(\tau * \tau)$.

ML code:

```
case findx() of
   None => None
  | Some x => case findy() of None => None
                           |  Some y => Some (x,y)
```

Do notation

findx() >>= λx.
findy() >>= λy.
return (x,y)

Haskell has syntax to make this even cleaner:

```
do {
  x ← findx();
  y ← findy();
  return (x,y)
}
```

State monad

Types

- State τ

Definition

- State τ $\equiv$ s $\rightarrow$ s * τ (s is some type for representing state, i.e. partial functions)

Operations

- return : $\tau \rightarrow$ State τ
- >>= : State $\tau \rightarrow (\tau \rightarrow$ State $\tau') \rightarrow$ State τ' (infix)
- set : Loc $\rightarrow$ Int $\rightarrow$ State ()
- get : Loc $\rightarrow$ State Int
- new : () $\rightarrow$ State Loc

Typo

Example: swap

Assume x and y are two locations.

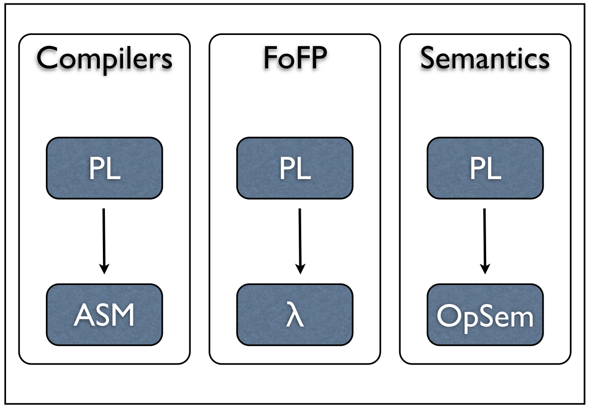
```
do {
  z ← get x;
  w ← get y;
  set y z;
  set x w
}
```

Haskell

Read up on Haskell if this interests you.

Concluding remarks

Where this course sits



Summary

“Everything” can be encoded into the λ-calculus.

- Caveat: not concurrency!

Should we encode everything into λ-calculus?

The end