Foundations of functional programming

Matthew Parkinson
12 Lectures (Lent 2009)
Overview

Foundations of CS

Compilers

Types

FofFP

Den Sem

Semantics
Materials

Previous lecturers notes are still relevant.

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) $\lambda$-calculus

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

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

World smallest programming language:
- $\alpha, \beta, \eta$ 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 $\lambda$-calculus

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

Syntax:
• $x$ variables
• $\lambda x . M$ (lambda) abstraction
• $M \ M$ (function) application
• $c$ (constants)

Elements of $c$ used to represent integers, and also functions such as addition
• $\delta$ reductions are added to deal with constants
Combinators

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

We just have \( c \in \{S, K\} \) regains power of \( \lambda \)-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#)
Abstract machine for executing the $\lambda$-calculus.

4 registers Stack, Environment, Control and Dump.
Continuations

• $\lambda$-expressions restricted to always return 
  "()" [continuations] can implement all $\lambda$-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 $\lambda$-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.
Syntax trees

\[ \lambda x.(x \; x) \]

\[
\text{app} \\
M \\
N \\
\text{fn} \\
\text{“x”} \\
M \\
\text{var} \\
\text{“x”} \\
\text{“x”} \\
\]

\[
\text{fn} \\
\text{“x”} \\
\text{app} \\
\text{var} \\
\text{“x”} \\
\text{var} \\
\text{“x”} \\
\]
Free variables and permutation

We define free variables of a \( \lambda \)-term as
- \( \text{FV}(M \ N) = \text{FV}(M) \cup \text{FV}(N) \)
- \( \text{FV}(\lambda x. M) = \text{FV}(M) \setminus \{x\} \)
- \( \text{FV}(x) = \{x\} \)

We define variable permutation as
- \( x < x \cdot z > = x < z \cdot x > = z \)
- \( x < y \cdot z > = x \) (provided \( x \neq y \) and \( x \neq z \))
- \( (\lambda x. M) < y \cdot z > = \lambda (x < y \cdot z >).(M < y \cdot z >) \)
- \( (M \ N) < y \cdot z > = (M < y \cdot z >)\ (N < y \cdot z >) \)
An equivalence relation is a reflexive, symmetric and transitive relation.

R is an equivalence relation if

- Reflexive
  \[ \forall x. \ x R x \]
- Transitive
  \[ \forall xyz. \ x R y \land y R z \Rightarrow x R z \]
- Symmetric
  \[ \forall xy. \ x R y \Rightarrow y R x \]
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 $M$.

- $(\lambda x. C)[N] = \lambda x. (C[N])$
- $(C \ M)[N] = (C[N]) \ M$
- $(M \ C)[N] = M \ (C[N])$
- $\bullet \ [N] = N$
A congruence relation is an equivalence relation, that is preserved by placing terms under contexts.

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.
\(\alpha\)-equivalence

Two terms are \(\alpha\)-equivalent if they can be made syntactically equal \((\equiv)\) by renaming bound variables.

\(\alpha\)-equivalence \((=_{\alpha})\) is the least congruence relation satisfying

\[\lambda x. M =_{\alpha} \lambda y. M<x\cdot y>\text{ where } y \notin \text{FV}(\lambda x. M)\]
Intuition of $\alpha$-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 $\alpha$–equivalence.
Are these alpha-equivalent?

\[ \lambda x.x =_\alpha \lambda y.y \]

\[ \lambda x.\lambda y.x =_\alpha \lambda y.\lambda x.y \]

\[ \lambda x.y =_\alpha \lambda y.y \]

\[ (\lambda x.x)(\lambda y.y) =_\alpha (\lambda y.y)(\lambda x.x) \]

\[ \lambda x.\lambda y. (x \ z \ y) =_\alpha \lambda z.\lambda y. (z \ z \ y) \]
\( \alpha \)-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 \( \lambda xs. M_2 =_\alpha \lambda ys. N_2 \)
Capture avoiding substitution

If $x \notin \text{FV}(M)$,
- $M \left[ L/x \right] = M$
otherwise:
- $(M \ N) \left[ L/x \right] = (M \left[ L/x \right] \ N \left[ L/x \right])$
- $(\lambda y. M) \left[ L/x \right] = (\lambda z. M<z \cdot y>[L/x])$
  where $z \notin \text{FV}(x, L, \lambda y.m)$
- $x \left[ L/x \right] = L$

Note: In the $(\lambda y. M)$ case, we use a permutation to pick an $\alpha$–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

\[ (\lambda x. x x) (\lambda x. x x) y z \equiv (((\lambda x. (x x)) (\lambda x. (x x))) y) z \]
\[ \lambda xyz.xyz \equiv \lambda x. (\lambda y. (\lambda z. ((xy) z))) \]
Extra brackets - again

\[ L \, M \, N \equiv (L \, M) \, N \]
\(\beta\eta\)-reduction

We define \(\beta\)-reduction as:

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

This is the workhorse of the \(\lambda\)-calculus.

We define \(\eta\)-reduction as: If \(x \notin \text{FV}(M)\), then

\[\lambda x. (M x) \rightarrow_\eta M\]

This collapses trivial functions.

Consider \((\text{fn } x \rightarrow \text{sin } x)\) is this the same as \text{sin} in ML?
$\beta \eta$ examples

$$(\lambda x. \, x \, y) \,(\lambda z. \, z) \rightarrow_\beta \lambda z. \, z \, y$$

$$(\lambda x. \, x \, y) \,(\lambda z. \, z) \rightarrow_\beta (\lambda z. \, z) \, y$$

$$\lambda x. \, M \, N \, x \rightarrow_\eta (\lambda x. \, M \, N)$$

$$(\lambda x. \, x \, x) \,(\lambda x. \, x \, x) \rightarrow_\beta (\lambda x. \, x \, x) \,(\lambda x. \, x \, x)$$

$$(\lambda xy. \, x) \,(\lambda x. \, y) \rightarrow_\beta (\lambda y x. \, y)$$
Reduction in a context

We actually define $\beta$-reduction as:

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

and $\eta$-reduction as:

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

where $C ::= \lambda x. C \mid C \ M \mid M \ C \mid \ast$ (from “Context and Congruence” slide)

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

\[ C[M] \xrightarrow{\beta} C \xrightarrow{\beta} C[N] \]

\[ M \xrightarrow{\beta} N \]
How many reductions?

\[(\lambda x.x) ((\lambda x.x) (\lambda x.x))\]

\[(\lambda x.x) (\lambda x.x) (\lambda x.x)\]

\[((\lambda xy.x) z) w\]

\[(\lambda xy.z) ((\lambda x.x) (\lambda x.x)) (\lambda x.x)\]

\[\lambda xy. z\]
Reduction and normal forms
Normal-form (NF)

A term is in normal form if it there are no $\beta$ or $\eta$ 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 $(\lambda x. y x)$

normal-form:
- $NF ::= \lambda x. NF$ (if $\forall M. NF \neq M x$ or $x \in FV(M)$)
  - $NF_1 NF_2$ (if $\forall M. NF_1 \neq \lambda x. M$)
  - $x$
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 \(\Omega\).

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

\[ C ::= C \mathsf{M} | \mathsf{M} C | \cdot \]

That is, we don’t reduce under a $\lambda$.

$\lambda x. \Omega$ is a WHNF, but not a NF.
Multi-step reduction

\[ M \rightarrow^* N \quad \text{iff} \]

- \[ M \rightarrow^\beta N \]
- \[ M \rightarrow^\eta N \]
- \[ M \equiv N \quad \text{(reflexive)} \]
- \[ \exists L. \ M \rightarrow^* L \quad \text{and} \quad L \rightarrow^* N \quad \text{(transitive)} \]

The transitive and reflexive closure of \( \beta \) and \( \eta \) reduction.
Equality

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

• $\alpha$-equivalence (implicitly)
• $\beta$-reduction
• $\eta$-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 \((M \rightarrow^* N \text{ or } N \rightarrow^* M)\), then \(M = N\).
The converse is not true (Exercise: why?)

If \(L \rightarrow^* M\) and \(L \rightarrow^* N\), then \(M = N\).

If \(M \rightarrow^* L\) and \(N \rightarrow^* L\), then \(M = N\).
Church-Rosser Theorem

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

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

- $(\lambda x.ax)((\lambda y.by)c) \rightarrow_\beta a((\lambda y.by)c) \rightarrow_\beta a(bc)$
- $(\lambda x.ax)((\lambda y.by)c) \rightarrow_\beta (\lambda x.ax)(bc) \rightarrow_\beta a(bc)$

Note: Underlined term is reduced.
Consequences

If $M=N$ and $N$ is in normal form, then $M \rightarrow^T N$.

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 $M \not= N$. For example, $\lambda xy.x \not= \lambda xy.y$. 
Diamond property

Key to proving Church–Rosser Theorem is demonstrating the diamond property:
• If $M \rightarrow^* N_1$ and $M \rightarrow^* N_2$, then there exists $L$ such that $N_1 \rightarrow^* L$ and $N_2 \rightarrow^* L$.

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

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

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

- \((\lambda x.a) \Omega \rightarrow_\beta a\)
- \((\lambda x.a) \Omega \rightarrow_\beta (\lambda x.a) \Omega\)

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

Perform leftmost, outermost \( \beta \)-reduction.
(leave \( \eta \)-reduction until the end)

Reduction context
- \( C ::= \lambda x. C \)
  - \( C M \) (if \( \forall C'. x. \ C \neq \lambda x. C' \))
  - \( \text{NF} \ C \) (if \( \forall M \ x. \ \text{NF} \neq \lambda x. M \))

where NF is from normal-form definition.

This definition is guaranteed to reach normal-form if one exists.
Example reduction: normal order

\[(\lambda x. x (\lambda y. y)) (\lambda y. (\lambda z. z z z) (y t))\]

\[\rightarrow (\lambda y. (\lambda z. z z z) (y t)) (\lambda y. y)\]

\[\rightarrow (\lambda z. z z z) ((\lambda y. y) t)\]

\[\rightarrow (\lambda y. y) t ((\lambda y. y) t) ((\lambda y. y) t) ((\lambda y. y) t)\]

\[\rightarrow t ((\lambda y. y) t) ((\lambda y. y) t) ((\lambda y. y) t)\]

\[\rightarrow t t ((\lambda y. y) t) ((\lambda y. y) t)\]

\[\rightarrow t t t ((\lambda y. y) t)\]

\[\rightarrow t t t t\]
Call-by-name

Do not reduce under $\lambda$ and do not reduce argument

- $C ::= C\ M \mid \cdot$
Example reduction: CBN

$$(\lambda x.x (\lambda y.y)) (\lambda y.(\lambda z.z z z z) (y t))$$
$$\rightarrow (\lambda y.(\lambda z.z z z z) (y t)) (\lambda y.y)$$
$$\rightarrow (\lambda z.z z z z) ((\lambda y.y) t)$$
$$\rightarrow (\lambda y.y) t ((\lambda y.y) t) ((\lambda y.y) t) ((\lambda y.y) t)$$
$$\rightarrow t ((\lambda y.y) t) ((\lambda y.y) t) ((\lambda y.y) t)$$
Call-by-value

\[ V ::= \, x \mid \lambda x. \, M \, \text{ (values)} \]
\[ C ::= C \, M \, \mid \, \cdot \, \mid (\lambda x. M) \, C \]
\[ C[ (\lambda x. M) \, V ] \to_\beta C[ M[V/x] ] \]

Do no reduce under \( \lambda \), and only apply function when its argument is a value.
Example reduction: CBV

\[(\lambda x. x (\lambda y. \lambda z. z z z) (y t)) \rightarrow (\lambda y. (\lambda z. z z z) (y t)) (\lambda y. y) \rightarrow (\lambda z. z z z) ((\lambda y. y) t) \rightarrow (\lambda z. z z z) t \rightarrow t t t t t\]
Call-by-normal-form

• \( V ::= x | \lambda x. M \) (values)
• \( C ::= C M \) (if \( \forall C' x. C \neq \lambda x.C' \))
  |  
  |  
  |  
  |  \( (\lambda x.M) C \)
  |  \( \lambda x.C \)
• \( C[ (\lambda x.M) \text{ NF } ] \rightarrow_\beta C[ M[\text{NF/}x] ] \)

Only apply function when its argument is a normal-form.
Example reduction: CB-NF

$$(\lambda x. x (\lambda y.y)) (\lambda y. (\lambda z.z z z z) (y t))$$

$$\rightarrow (\lambda x. x (\lambda y.y)) (\lambda y.y t (y t) (y t) (y t))$$

$$\rightarrow (\lambda y.y t (y t) (y t) (y t)) (\lambda y.y)$$

$$\rightarrow (\lambda y.y) t ((\lambda y.y) t) ((\lambda y.y) t) ((\lambda y.y) t)$$

$$\rightarrow t ((\lambda y.y) t) ((\lambda y.y) t) ((\lambda y.y) t)$$

$$\rightarrow t t ((\lambda y.y) t) ((\lambda y.y) t)$$

$$\rightarrow t t t ((\lambda y.y) t)$$

$$\rightarrow t t t t$$
All possible reductions

(λx.x (λy.y)) (λy.(λz.z z z) (y t))

(λy.(λz.z z z) (y t)) (λy.y)

(λz.z z z) ((λy.y) t)

(λy.y t (y t) (y t) (y t)) (λy.y)

(λy.y) t ((λy.y) t) ((λy.y) t) ((λy.y) t)

See next slide

t t t t
The complicated bit

\[(\lambda y. t ((\lambda y. t) ((\lambda y. t) ((\lambda y. t) ((\lambda y. t) t)) t)) t)\]
Encoding Data
Motivation

We want to use different datatypes in the $\lambda$-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 ≡ λm n. m
• FALSE ≡ λm n. n
• IF ≡ λb m n. b m n

TRUE and FALSE are both in normal-form, so by Church–Rosser, we know TRUE ≢ FALSE.

Note that, IF is not strictly necessary as
• ∀P. IF P = P (Exercise: show this).
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 ≡ \( \lambda xy. \text{IF } x \text{ y FALSE} \)
- OR ≡ \( \lambda xy. \text{IF } x \text{ TRUE } y \)
- NOT ≡ \( \lambda x. \text{IF } x \text{ FALSE TRUE } \)
Encoding pairs

Constructor:
• \( \text{PAIR} \equiv \lambda xy. fxy \)

Destructors:
• \( \text{FST} \equiv \lambda p.p \text{ TRUE} \)
• \( \text{SND} \equiv \lambda p.p \text{ FALSE} \)

Properties: \( \forall pq. \)
• \( \text{FST} \ (\text{PAIR} p q) = p \)
• \( \text{SND} \ (\text{PAIR} p q) = q \)
Encoding sums

Constructors:
• \( \text{INL} \equiv \lambda x. \text{PAIR TRUE} \ x \)
• \( \text{INR} \equiv \lambda x. \text{PAIR FALSE} \ x \)

Destructor:
• \( \text{CASE} = \lambda s \ f \ g. \text{IF} (\text{FST} \ s) (f (\text{SND} \ s)) (g (\text{SND} \ s)) \)

Properties:
• \( \text{CASE (INL x) f g} = f x \)
• \( \text{CASE (INR x) f g} = g x \)
Encoding sums (alternative defn)

Constructors:
• \text{INL} ≡ \lambda x \ f \ g. \ f \ x
• \text{INR} ≡ \lambda x \ f \ g. \ g \ x

Destructors:
• \text{CASE} ≡ \lambda s \ f \ g. \ s \ f \ g

As with booleans destructor unnecessary.
• \forall p. \text{CASE} \ p = p
Church Numerals

Define:
• \(0 \equiv \lambda f \ x. \ x\)
• \(1 \equiv \lambda f \ x. \ f \ x\)
• \(2 \equiv \lambda f \ x. \ f \ (f \ x)\)
• \(3 \equiv \lambda f \ x. \ f \ (f \ (f \ x))\)
• ...
• \(n \equiv \lambda f \ x. \ f(...(f \ x)...\)

That is, \(n\) takes a function and applies it \(n\) times to its argument: \(n \ f\) is \(f^n\).
Arithmetic

Definitions
• ADD $\equiv \lambda mnfx. \ m \ f \ (n \ f \ x)$
• MULT $\equiv \lambda mnfx. \ m \ (n \ f) \ x \ = \ \lambda mnf. \ m \ (n \ f)$
• EXP $\equiv \lambda mnfx. \ n \ m \ f \ x \ = \ \lambda mn. \ n \ m$

Example:
ADD $m \ n \ \rightarrow T \ \lambda f \ x. \ m \ f \ (n \ f \ x) \ \rightarrow T \ f^m \ (f^n \ x) \ \equiv f^{m+n} \ x$
More arithmetic

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

Properties
• SUC n = n+1
• ISZERO 0 = TRUE
• ISZERO (n+1) = FALSE

We also require decrement/predecessor!
# Building decrement

<table>
<thead>
<tr>
<th>n</th>
<th>PFN(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
</tr>
<tr>
<td>1</td>
<td>(1,0)</td>
</tr>
<tr>
<td>2</td>
<td>(2,1)</td>
</tr>
<tr>
<td>3</td>
<td>(3,2)</td>
</tr>
<tr>
<td>4</td>
<td>(4,3)</td>
</tr>
</tbody>
</table>

Successor transitions are indicated by arrows labeled "suc."
Decrement and subtraction

Definitions:
• $\text{PFN} \equiv \lambda n. n (\lambda p. \text{PAIR} (\text{SUC}(\text{FST } p)) (\text{FST } p)) (\text{PAIR } 0 \ 0)$
• $\text{PRE} \equiv \lambda n. \text{SND} (\text{PFN } n)$
• $\text{SUB} \equiv \lambda mn. n \ \text{PRE } m$

Exercise: Evaluate
• $\text{PFN } 5$
• $\text{PRE } 0$
• $\text{SUB } 4 \ 6$

Correction.
Using PAIR rather than $( , )$ notation.
Also, changed P to p
Lists

Constructors:
• \( \text{NIL} \equiv \text{PAIR TRUE (\( \lambda \)z.z) } \)
• \( \text{CONS} \equiv \lambda xy. \text{PAIR FALSE (PAIR x y)} \)

Destructors:
• \( \text{NULL} \equiv \text{FST} \)
• \( \text{HD} \equiv \lambda l. \text{FST (SND l)} \)
• \( \text{TL} \equiv \lambda l. \text{SND (SND l)} \)

Properties:
• \( \text{NULL NIL} = \text{TRUE} \)
• \( \text{HD (CONS M N)} = M \)
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:

\[
\text{letrec } f(x) = M \text{ in } N
\]

this is really

\[
\text{let } f = Y (\lambda f. \lambda x. M) \text{ in } N
\]

and hence

\[
(\lambda f. N) (Y \lambda f. \lambda x. M)
\]
Defining recursive function

Consider defining a factorial function with the following property:

$$\text{FACT} = \lambda n. (\text{ISZERO } n) \cdot 1 \cdot (\text{MULT } n \cdot (\text{FACT} \cdot (\text{PRE } n)))$$

We can define

$$\text{PREFACT} = \lambda f n. (\text{ISZERO } n) \cdot 1 \cdot (\text{MULT } n \cdot (f \cdot (\text{PRE } n)))$$

Properties

• Base case: $$\forall F. \text{PREFACT } F \cdot 0 = 1$$
• Inductive case: $$\forall F. \text{If } F \text{ behaves like factorial up to } n, \text{ then } \text{PREFACT } F \text{ behaves like factorial up to } n+1;$$
Fixed points

Discrete Maths: \( x \) is a fixed point of \( f \), iff \( f \cdot x = x \)

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

- \( Y \cdot f = f \cdot (Y \cdot f) \)

Hence, by using \( Y \) we can satisfy this property:

\[
\text{FACT} \equiv Y \cdot \text{PREFACT}
\]

Exercise: Show \( \text{FACT} \) satisfies property on previous slide.
General approach

If you need a term, M, such that
  • $M = P M$
Then $M \equiv YP$ suffices

Example:
  • $\text{ZEROES} = \text{CONS } 0 \text{ ZEROES} = (\lambda p. \text{CONS } 0 p) \text{ ZEROES}$
  • $\text{ZEROES} \equiv Y (\lambda p. \text{CONS } 0 p)$
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:
\[
L \equiv Y(\lambda p. \text{PAIR} (P (\text{FST} p) (\text{SND} p)) (Q (\text{FST} p) (\text{SND} p)))
\]
\[
M \equiv \text{FST} L
\]
\[
N \equiv \text{SND} L
\]

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:
• \( (\lambda xy.xyz) (\lambda xy.xyz) \)
(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.
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:
- \( x, \; \lambda y. x, \; \lambda z. z((\lambda x. a)c), \)
- \( \lambda f. f (\lambda x. f(xz)) (\lambda x. f(xz)) \)

Non-examples:
- \( \lambda y. (\lambda x. a) y \rightarrow \lambda y. a \)
- \( \lambda f. (\lambda x. f(xx)) (\lambda x. f(xx)) \)
Properties

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

- $C' ::= C' \ M | \cdot$
- $C ::= \lambda x. C | C'$

Therefore, $\Omega$ has no HNF (Exercise: prove this.)

If $M \ N$ has a HNF, then so does $M$. Therefore, if $M$ has no HNF, then $M \ N_1 \ldots N_k$ does not have a HNF. Hence, $M$ is a “totally undefined function”.
ISWIM

\( \lambda \)-calculus as a programming language
(The next 700 programming languages [Landin 1966])
ISWIM: Syntax

From the $\lambda$–calculus

• $x$ (variable)
• $\lambda x. M$ (abstraction)
• $M \ N$ (application)

Local declarations

• let $x = M$ in $N$ (simple declaration)
• let $f \ x_1 \ldots x_n = M$ in $N$ (function declaration)
• letrec $f \ x_1 \ldots x_n = M$ in $N$ (recursive declaration)

and post–hoc declarations

• $N$ where $x = M$
• ...

**ISWIM: Syntactic sugar**

\[ N \text{ where } x = M \equiv \text{let } x = M \text{ in } N \]
\[ \text{let } x = M \text{ in } N \equiv (\lambda x. N) M \]
\[ \text{let } f \ x_1 \ldots x_n = M \text{ in } N \equiv \text{let } f = \lambda x_1 \ldots x_n. M \text{ in } N \]
\[ \text{letrec } f \ x_1 \ldots x_n = M \text{ in } N \equiv \text{let } f = Y(\lambda f. \lambda x_1 \ldots x_n. M) \text{ in } N \]

Desugaring explains syntax purely in terms of \( \lambda \)-calculus.
ISWIM: Constants

\[ M ::= x \mid c \mid \lambda x. M \mid M \, N \]

Constants c include:
• 0 1 −1 2 −2 ... (integers)
• + − × / (arithmetic operators)
• = ≠ < > (relational operators)
• true false (booleans)
• and or not (boolean connectives)

Reduction rules for constants: e.g.
• + 0 0 \rightarrow_δ 0
Call-by-value and IF-THEN-ELSE

ISWIM uses the call-by-value $\lambda$-calculus.

Consider: \[
\text{IF TRUE 1 \Omega}
\]

\[
\text{IF E THEN M ELSE N} \equiv \text{IF E (} \lambda x. M \text{) (} \lambda x. N \text{)) (} \lambda z.z \text{)}
\]

where $x \notin \text{FV(M N)}$
Pattern matching

Has
• (M,N) (pair constructor)
• \(\lambda(p_1,p_2). M\) (pattern matching pairs)

Desugaring
• \(\lambda(p_1,p_2). M \equiv \lambda z.(\lambda p_1 p_2. M) (\text{fst } z) (\text{snd } z)\)
where \(z \not\in \text{FV}(M)\)
Real $\lambda$-evaluator

Don’t use $\beta$ and substitution

Do use environment of values, and delayed substitution.
Environments and Closures

Consider $\beta$-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:

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

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 \( \lambda \)-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
  \[(S_1,E_1,C_1,(S_2,E_2,C_2,...(S_n,E_n,C_n,−)...))\]
  It is essentially a list of triples \((S_1,E_1,C_1),..., (S_n,E_n,C_n)\) and serves as the function call stack.
State transitions: constant
State-transition: variable
State-transition: function

\[
\text{S} \quad \text{E} \\
\lambda x. M; C \\
\text{D}
\]

\[
\text{Clo(x, M, E); S} \\
\text{E} \\
\text{C} \\
\text{D}
\]
State-transition: application
State-transition: app primitive

<table>
<thead>
<tr>
<th>f; a; S</th>
<th>f(a); S</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>app; C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>
State-transition: app closure
State-transition: return

\[
\text{a} \\
\text{E'} \\
\text{-} \\
(S,E,C,D) \\
\rightarrow \\
\text{a; S} \\
\text{E} \\
\text{C} \\
\text{D}
\]
Final configuration

```
 a
 -
 -
 -
```
Compiled SECD machine

Inefficient as requires construction of closures. Perform some conversions in advance:
• $⟦c⟧ ≡ \text{const } c$
• $⟦x⟧ ≡ \text{var } x$
• $⟦M N⟧ ≡ ⟦N⟧ ; ⟦M⟧ ; \text{app}$
• $⟦\lambda x.M⟧ ≡ \text{Closure}(x,⟦M⟧)$
• $⟦M + N⟧ ≡ ⟦M⟧ ; ⟦N⟧ ; \text{add}$
• ...

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

We can see \(((λxy.x + y)\ 3\ )\ 5\) 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:
• \(\lambda f x. f (\lambda y. x x y) (\lambda y. x x y)\)
This is very inefficient.

Better approach to have closure with pointer to itself.
Recursive functions \((Y(\lambda fx.M))\)
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:

\[ P, Q, R ::= S \mid K \mid P \cdot Q \]

Reductions:

\[ K \cdot P \cdot Q \rightarrow_w P \]
\[ S \cdot P \cdot Q \cdot R \rightarrow_w (P \cdot R) \cdot (Q \cdot R) \]

Note that the term \( S \cdot K \) does not reduce: it requires three arguments. Combinator reductions are called “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 $P \rightarrow_{wT} R$ and $Q \rightarrow_{wT} R$
Encoding the \( \lambda \)-calculus

Use extended syntax with variables:

- \( P ::= S | K | P P | x \)

Define meta-operator on combinators \( \lambda^* \) by

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

\(\lambda\)-calculus to SK:
- \((\lambda x. M)_{CL} = (\lambda Tx. (M)_{CL})\)
- \((x)_{CL} = x\)
- \((M N)_{CL} = (M)_{CL} (N)_{CL}\)

SK to \(\lambda\)-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
• $\text{FV}(M) = \text{FV}(\ (M)_{\text{CL}} \ )$
• $\text{FV}(P) = \text{FV}(\ (P)_{\lambda} \ )$

Supports $\alpha$ and $\beta$ reduction:
• $(\lambda T \ x. P) \ Q \ \rightarrow_w T \ P[Q/x]$
• $(\lambda T \ x. P) \equiv \lambda T y. P<y \cdot x>$ (where $y \not\in \text{FV}(P)$)
Equality on combinators

Combinators don’t have an analogue of the $\eta$-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 $x \notin \text{FV}(P Q))$. \\
  $S K x y \rightarrow (K y) (K x) \rightarrow y \leftarrow I y \leftarrow K I x y$

Therefore, $SK = KI$. 
Properties

We get the following properties of the translation:

• \([(M)_{CL}]_{\lambda} = M\]
• \([(P)_{\lambda}]_{CL} = P\]
• \(M=N \iff (M)_{CL} = (N)_{CL}\)
• \(P=Q \iff (P)_{\lambda} = (Q)_{\lambda}\)
Aside: Hilbert style proof

In Logic and Proof you covered Hilbert style proof:

• Axiom K: \( \forall A B. A \rightarrow (B \rightarrow A) \)
• Axiom S: \( \forall A B C. (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)) \)
• Modus Ponens: If \( A \rightarrow B \) and \( A \), then \( B \)

Hilbert style proofs correspond to “Typed” combinator terms:

• \( S \ K : \forall A B. ((A \rightarrow B) \rightarrow (A \rightarrow A)) \)
• \( S \ K \ K : \forall A. (A \rightarrow A) \)

Logic, Combinators and the \( \lambda \)-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

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

(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 \(\lambda^*\) is exponential, while \(\lambda^T\) is only quadratic.
Example

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

This is wrong!!!!
Example

\[ \lambda^T f. (\lambda^T x. f (x \ x)) \]
\[ = \lambda^T f. B f (\lambda^T x. x \ x) \]
\[ = \lambda^T f. B f (S (\lambda^T x. x) (\lambda^T x. x)) \]
\[ = \lambda^T f. B f (S I (\lambda^T x. x)) \]
\[ = \lambda^T f. B f (S I I) \]
\[ = C (\lambda^T f. B f) (S I I) \]
\[ = C B (S I I) \]
Combinators as graphs

To enable lazy reduction, consider combinator terms as graphs.

S reduction creates two pointers to the same subterm.

Let’s consider
• let sqr x = mult x x in sqr 5 ≡ (λf. f 5)(λm. mult m m)
this translates to
• C I 5 (S mult I)

Exercise: Show this translation.
C 1 5 (S mult I)
Reduction: I
Reduction: K
Reduction: S
Reduction: B
Reduction: C
Recursion
Evaluation
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 $\lambda$-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 \( \exists \tau'. \, M : \tau' \rightarrow \tau \) and \( N : \tau' \)
- \( \lambda x. \, M : \tau \rightarrow \tau' \) iff \( \exists \tau. \, \text{if } x:\tau \text{ then } M : \tau' \)
- \( n : \text{int} \)

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

\[ \lambda x. \ x \ : \ \text{int} \rightarrow \text{int} \]

\[ \lambda x \ f \ x : \ \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \]

\[ \lambda fgx. \ fgx : (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_2 \rightarrow \tau_3) \rightarrow \tau_1 \rightarrow \tau_3 \]

\[ \lambda fgx. \ f(gx) : (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_2 \rightarrow \tau_3) \rightarrow \tau_1 \rightarrow \tau_3 \]

\[ \lambda f \ x. \ f(f \ x) : (\tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau \]
Types help find terms

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

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

Therefore \(M \equiv \lambda xy . N\) where \(x : \tau_2, y : \tau_1\) and \(N : \tau_3\).

Therefore \(N \equiv f y x\)

Therefore \(\lambda fxy . f y x : (\tau_1 \to \tau_2 \to \tau_3) \to \tau_2 \to \tau_1 \to \tau_3\)
Polymorphism and inference

ML type system supports polymorphism:
\[ \tau ::= \alpha \mid \forall \alpha. \tau \mid \ldots \]

Types can be inferred using unification.
Recap: Call-by-name

Do not reduce under $\lambda$ and do not reduce argument
• $C ::= C M \mid \cdot$
Recap: Call-by-value

- $V ::= x \mid \lambda x. M$ (values)
- $C ::= C \ M \mid \bullet \mid (\lambda x. M) \ C$
- $C[ (\lambda x. M) \ V ] \rightarrow_{\beta} C[ M[V/x] ]$

Do no reduce under $\lambda$, and only apply function when its argument is a value.
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. $\llbracket x \rrbracket_v(k) \equiv k \times$
2. $\llbracket c \rrbracket_v(k) \equiv k \cdot c$
3. $\llbracket \lambda x. M \rrbracket_v(k) \equiv k (\lambda(x,k'). \llbracket M \rrbracket_v(k'))$
4. $\llbracket M N \rrbracket_v(k) \equiv \llbracket M \rrbracket_v (\lambda m. \llbracket N \rrbracket_v(\lambda n. m(n,k)))$

Intuition:
• $\llbracket M \rrbracket_v(k)$ 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

\[
\begin{align*}
\llbracket \lambda x. y \rrbracket_v (k) &= k (\lambda (x, k'). \llbracket y \rrbracket_v (k')) \\
&= k (\lambda (x, k'). k' y) \\
\llbracket (\lambda x. y) z \rrbracket_v (k) &= \llbracket \lambda x. y \rrbracket_v (\lambda m. \llbracket z \rrbracket_v (\lambda n. m (n,k))) \\
&= \llbracket \lambda x. y \rrbracket_v (\lambda m. (\lambda n. m(n,k)) z) \\
&= (\lambda m. (\lambda n. m(n,k)) z) (\lambda (x, k'). k' y) \\
&\rightarrow (\lambda n. (\lambda (x, k'). k' y)) (n, k)) z \\
&\rightarrow (\lambda (x, k'). k' y)) (z, k) \\
&\rightarrow k y
\end{align*}
\]
Call-by-name

Definition:
• $\left\lfloor x \right\rfloor_n(k) \equiv x \, k$
• $\left\lfloor c \right\rfloor_n(k) \equiv k \, c$
• $\left\lfloor \lambda x. M \right\rfloor_n(k) \equiv k \, (\lambda(x,k'). \left\lfloor M \right\rfloor(k'))$
• $\left\lfloor M \, N \right\rfloor_n(k) \equiv \left\lfloor M \right\rfloor (\lambda m. m (\lambda k'. \left\lfloor N \right\rfloor(k'), k))$

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

For any closed term $M$ ($\text{FV}(M) = \{\} \}$)
- $M$ terminates with value $v$ in the CBV $\lambda$-calculus, iff $\llbracket M \rrbracket_v(\lambda x.x)$ terminates in both the CBV and CBN $\lambda$-calculus with value $v$.
- $M$ terminates with value $v$ in the CBN $\lambda$-calculus, iff $\llbracket M \rrbracket_n(\lambda x.x)$ terminates in both the CBV and CBN $\lambda$-calculus with value $v$. 
Encoding control

Consider trying to add an Exit instruction to the $\lambda$-calculus.

- $\text{Exit } M \rightarrow \text{Exit}$ (CBN and CBV)
- $(\lambda x. M) \text{ Exit} \rightarrow \text{Exit}$ (Just CBV)

When we encounter Exit execution is stopped.

- $(\lambda x. y) \text{ Exit} = \text{Exit}$ (CBV)
- $(\lambda x. y) \text{ Exit} = y$ (CBN)

Encode as

- $\llbracket \text{Exit} \rrbracket (k) = ()$ (Both CBV and CBN)
Example CBV

\[ [(\lambda x.y) \text{Exit}]_v(k) \]
\[ \equiv [(\lambda x.y)_v (\lambda m. [(\text{Exit})_v (\lambda n. m \ (n,k))])]_v \]
\[ \equiv [(\lambda x.y)_v (\lambda m. ())]_v \]
\[ \equiv (\lambda m. ()) (\lambda(x,k'). k' \ y) \]
\[ \rightarrow () \]
Example CBN

\[(\lambda x. y) \text{Exit} \]_n(k)

≡ \(\lambda x. y\) \(\lambda m. m (\lambda k'. [\text{Exit}]_n(k'), k)\)

≡ \(\lambda m. m (\lambda k'. [\text{Exit}]_n(k'), k)\) (\(\lambda (x, k'). y\)\( k'\))

→ \(\lambda (x, k'). y\)\( k'\) (\(\lambda k'. [\text{Exit}]_n(k'), k\))

→ \(y k\)
Order of evaluation

With CBV we can consider two orders of evaluation:

Function first:
$$\llbracket M N \rrbracket v_1(k) \equiv \llbracket M \rrbracket (\lambda m. \llbracket N \rrbracket (\lambda n. m (n,k)))$$

Argument first:
$$\llbracket M N \rrbracket v_2(k) \equiv \llbracket N \rrbracket (\lambda n. \llbracket M \rrbracket (\lambda m. m (n,k)))$$
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:
• \[ \text{Exit}_1 \text{ Exit}_2 \] v_1(k) = \[ \text{Exit}_1 \] v_1(k) (Function first)
• \[ \text{Exit}_1 \text{ Exit}_2 \] v_2(k) = \[ \text{Exit}_2 \] v_2(k) (Argument first)
Example (continued)

\[
\llbracket \text{Exit}_1 \text{ Exit}_2 \rrbracket_{v_1}(k) \\
\equiv \llbracket \text{Exit}_1 \rrbracket (\lambda m. \llbracket \text{Exit}_2 \rrbracket (\lambda n. m (n,k)))
\]
\[
\equiv 1 \equiv \llbracket \text{Exit}_1 \rrbracket (k)
\]

\[
\llbracket \text{Exit}_1 \text{ Exit}_2 \rrbracket_{v_2}(k) \\
\equiv \llbracket \text{Exit}_2 \rrbracket (\lambda n. \llbracket \text{Exit}_1 \rrbracket (\lambda m. m (n,k)))
\]
\[
\equiv 2 \equiv \llbracket \text{Exit}_2 \rrbracket (k)
\]
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:

1. \( [\tau_1 \rightarrow \tau_2]_v \equiv ([\tau_1]_v \times ([\tau_2]_v \rightarrow \perp)) \rightarrow \perp \)
2. \( [b]_v \equiv b \)

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

• $k : ([\tau_1 \rightarrow \tau_2]_v \rightarrow \bot)$
• $\lambda x.M : \tau_1 \rightarrow \tau_2$, hence $[M]_v : ([\tau_2]_v \rightarrow \bot) \rightarrow \bot$ if $x : [\tau_1]_v$

Find $N$ such that $kN : \bot$ therefore $N : [\tau_1 \rightarrow \tau_2]_v$

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

• $x : [\tau_1]_v$ and
• $k' : [\tau_2]_v \rightarrow \bot$

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

$[\lambda x.M]_v(k) \equiv k(\lambda(x,k'). [M]_v(k'))$
Types guide translation

Application translation (MN): Assume
• $k : (\llbracket \tau_2 \rrbracket_v \rightarrow \perp)$
• $M : \tau_1 \rightarrow \tau_2$, hence $\llbracket M \rrbracket_v : (\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_v \rightarrow \perp) \rightarrow \perp$
• $N : \tau_1$, hence $\llbracket N \rrbracket_v : (\llbracket \tau_1 \rrbracket_v \rightarrow \perp) \rightarrow \perp$

Find $L$ such that $\llbracket M \rrbracket_v L : \perp$ therefore $L : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket_v \rightarrow \perp$

So, $L \equiv \lambda m. L_1$, where $L_1 : \perp$ if $m : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket_v \equiv \llbracket \tau_1 \rrbracket_v \lambda (\llbracket \tau_2 \rrbracket_v \rightarrow \perp) \rightarrow \perp$

Find $L_2$ such that $\llbracket N \rrbracket_v L_2 : \perp$ therefore $L_2 : \llbracket \tau_1 \rrbracket_v \rightarrow \perp$

Therefore $L_2 \equiv \lambda n. L_3$ where $L_3 : \perp$ 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?

\[
\tau_1 \rightarrow \tau_2 \equiv ((T [\tau_1]_n) \times ([\tau_2]_n \rightarrow \bot)) \rightarrow \bot
\]

Typo T in notes should be ×
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 *       (* 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 (\ys. 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"]) (\xs. xs=[])
Example execution

\[ f (("a"; "a" | "a"); "a") ["a", "a"] (\lambda xs. xs=[]) \]
\[ \rightarrow f (("a"; "a" | "a"); "a") ["a", "a"] (\lambda xs. f "a" xs (\lambda xs.xs=[])) \]
\[ \rightarrow f ("a"; "a") ["a", "a"] (\lambda xs. f "a" xs (\lambda xs.xs=[])) \]
\[ \text{orelse } f "a" ["a", "a"] (\lambda xs. f "a" xs (\lambda xs.xs=[])) \]
\[ \rightarrow (\lambda xs. f "a" xs (\lambda xs.xs=[])) [] \]
\[ \text{orelse } f "a" ["a", "a"] (\lambda xs. f "a" xs (\lambda xs.xs=[])) \]
\[ \rightarrow \text{false} \]
\[ \text{orelse } f "a" ["a", "a"] (\lambda xs. f "a" xs (\lambda xs.xs=[])) \]
\[ \rightarrow f "a" ["a", "a"] (\lambda xs. f "a" xs (\lambda xs.xs=[])) \]
\[ \rightarrow (\lambda xs. f "a" xs (\lambda xs.xs=[])) ["a"] \]
\[ \rightarrow (\lambda xs.xs=[])) [] \rightarrow \text{true} \]
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.
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 $\lambda$-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 $s \times y = \lambda z. \text{IF } z=x \text{ THEN } y \text{ ELSE } s \times z$
- GET $s \times x = s \times x$

Note that, we ignore allocation in this encoding.
CPS and State

Definition: (This is a CBV translation.)

- $\llbracket x \rrbracket_v(k,s) \equiv k(x,s)$
- $\llbracket c \rrbracket_v(k,s) \equiv k(c,s)$
- $\llbracket \lambda x.M \rrbracket_v(k,s) \equiv k((\lambda(x,k',s'). \llbracket M \rrbracket(k',s'))), s)$
- $\llbracket M \, N \rrbracket_v(k,s) \equiv$
  $\llbracket M \rrbracket_v(\lambda(m,s'). \llbracket N \rrbracket_v(\lambda(n,s''). m(n,k,s''), s'), s)$
- $\llbracket !M \rrbracket_v(k,s) \equiv \llbracket M \rrbracket_v(\lambda(v,s'). k(\text{GET} s' v, s'), s)$
- $\llbracket [M]:=N \rrbracket_v(k,s) \equiv$
  $\llbracket M \rrbracket_v(\lambda(v,s'). \llbracket N \rrbracket_v(\lambda(v',s''). k((()), \text{SET} s'' v v'), s'), s)$
CPS and State

Definition: (This is a CBV translation.)

• $\llbracket x \rrbracket_v(s,k) \equiv k(s,x)$
• $\llbracket c \rrbracket_v(s,k) \equiv k(s,c)$
• $\llbracket \lambda x.M \rrbracket_v(s,k) \equiv k(s, (\lambda(s',x,k'). \llbracket M \rrbracket(s',k')))$
• $\llbracket M \ N \rrbracket_v(s,k) \equiv$
  $\llbracket M \rrbracket_v(s,\lambda(s',m). \llbracket N \rrbracket_v(s',\lambda(s'',n). m(s'',n,k)))$
• $\llbracket !M \rrbracket_v(s,k) \equiv \llbracket M \rrbracket_v(s, \lambda(s',v). k(s',\text{GET } s' \ v))$
• $\llbracket [M]:=N \rrbracket_v(s,k) \equiv$
  $\llbracket M \rrbracket_v(s, \lambda(s',v). \llbracket N \rrbracket_v(s',\lambda(s'',v').k(\text{SET } s'' \ v \ v'\),(),s))$
Exercises

• Extend encoding with sequential composition M;N
• Translate: \([x] := 1; \!x\)
• Translate: \((\lambda y.z)([x] := (!x+1))\)
• Redo translations above.
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 : \( \tau \rightarrow T \tau \)
• \( \gg= : T \tau \rightarrow (\tau \rightarrow T \tau') \rightarrow T \tau' \) [bind]
Option/Maybe Monad

Types
• Option $\tau$

Definition
• Option $\tau \equiv \text{unit} + \tau$

Operations
• return : $\tau \rightarrow \text{Option } \tau$
  return M $\equiv$ Some M
• $\gg=$ : Option $\tau \rightarrow (\tau \rightarrow \text{Option } \tau') \rightarrow \text{Option } \tau'$
  $\lambda$xy. case x of None $=>$ None | Some z $=>$ y z
Example

Imagine \texttt{findx} and \texttt{findy} are of type unit $\rightarrow$ Option $\tau$

\texttt{findx()} $\ggg \lambda x.\$
\texttt{findy()} $\ggg \lambda y.\$
return \((x,y)\)

This code is of type Option \((\tau \ T \ \tau)\).

ML code:
case \texttt{findx()} of
  None $\Rightarrow$ None
  | Some \(x\) $\Rightarrow$ case \texttt{findy()} of None $\Rightarrow$ None
  | | Some \(y\) $\Rightarrow$ Some \((x,y)\)
Do notation

findx() >>= \lambda x.
findy() >>= \lambda y.
return (x,y)

Haskell has syntax to make this even cleaner:

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

Types
  • State $\tau$

Definition
  • $\text{State} \: \tau \equiv s \rightarrow s^* \tau$  (s is some type for representing state, i.e. partial functions)

Operations
  • $\text{return} : \tau \rightarrow \text{State} \: \tau$
  • $\text{>>=} : \text{State} \: \tau \rightarrow (\tau \rightarrow \text{State} \: \tau') \rightarrow \text{State} \: \tau'$ (infix)
  • $\text{set} : \text{Loc} \rightarrow \text{Int} \rightarrow \text{State} ()$
  • $\text{get} : \text{Loc} \rightarrow \text{State} \: \text{Int}$
  • $\text{new} : () \rightarrow \text{State} \: \text{Loc}$
Haskell

Read up on Haskell if this interests you.
Concluding remarks
Where this course sits

- **Compilers**
  - PL
  - ASM

- **FoFP**
  - PL
  - λ

- **Semantics**
  - PL
  - OpSem
“Everything” can be encoded into the λ-calculus.

• Caveat: not concurrency!

Should we encode everything into λ-calculus?