UNIVERSITY OF
CAMBRIDGE

Foundations of Functional Programming

A twelve-lecture course (final version with minor revisions).

Alan Mycroft
Computer Laboratory, Cambridge University
`http://www.cl.cam.ac.uk/users/am/`

Lent 2007

Course Overview

- Sideways from the “Operational Semantics” course.
- Background implementation ideas for “Compiler Construction”.
- Rough area: λ-calculus and its applications
  $\lambda x.e$ [Church 1941] is parallel notation to ML “`fn x=>e`”
- Wikipedia articles:
  `http://en.wikipedia.org/wiki/Alonzo_Church`
  `http://en.wikipedia.org/wiki/Lambda_calculus`

(Pure) Lambda Calculus

$$e ::= x \mid e\,e' \mid \lambda x.e$$

Syntax:

- x variables
- $\lambda x.e$ (lambda-) abstraction
- $e\,e'$ (function) application

"The world's smallest programming language":

- α-, β-, η-reduction.
- when are λ-terms *equal*?
- choice of evaluation strategies.

Pure Lambda Calculus is Universal

Can encode:

- Booleans including Conditional
- Integers
- Pairs
- Disjoint Union
- Lists
- Recursion

within the λ-calculus.

Can even simulate a Turing Machine or Register Machine (see "Computation Theory") so "Turing Universal"

(Applied) Lambda Calculus

UNIVERSITY OF CAMBRIDGE

$$e ::= x \mid e\, e' \mid \lambda x.e \mid c$$

- x variables
- $\lambda x.e$ (lambda-) abstraction
- $e\, e'$ (function) application
- c constants

Elements of c can directly represent not only integers (etc.) but also function constants such as *addition* or *function composition*.

- "δ-reduction".

Combinators

$$e ::= e\,e' \mid c \qquad \text{(omit } x \text{ and } \lambda x.e\text{)}$$

- Remarkably: making $c = \{\mathbf{S}, \mathbf{K}\}$ regains the power of pure λ-calculus.
- Translation: λ-calculus to/from combinators, including (almost) equivalent reduction rules.

Evaluation Mechanisms/Facts

- Eager Evaluation
- Lazy Evaluation
- Confluence “There’s always a meeting place downstream”
- Implementation Techniques

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`

Wikipedia article:

- `http://en.wikipedia.org/wiki/Continuation`

Real Implementations of $\lambda-$expressions

- “Functional Languages”.
- Don’t do *substitution*, use *environments* instead.
- Haskell, ML.

SECD Machine

- An historic (but simple) abstract machine.
- 4 registers S, E, C, D
- Rules $(S, E, C, D) \longrightarrow (S', E', C', D')$

Types

- Most of this course is typeless.
- ML type system is a ‘sweet spot’ in some senses.

ML type system semi-formally

How to implement/understand ML types

- polymorphism for λ-calculus
- additional effects of `let`
- $\mathbf{Y}$ has a type, but cannot be expressed in ML

Part 1

Lambda-Calculus

Lambda Calculus

The *terms* of the λ-calculus, known as λ-*terms*, are constructed recursively from a given set of *variables* x, y, z, ... They may take one of the following forms:

x	variable
$(\lambda x.M)$	abstraction, where M is a term
(MN)	application, where M and N are terms

We use capital letters like L, M, N, ... for terms. We write $M \equiv N$ to state that M and N are identical λ-terms. The equality between λ-terms, $M = N$, will be discussed later.

Bound and Free Variables

BV(M), the set of all *bound variables* in M, is given by

$$\begin{aligned}
\mathrm{BV}(x) &= \emptyset \\
\mathrm{BV}(\lambda x.M) &= \mathrm{BV}(M) \cup \{x\} \\
\mathrm{BV}(MN) &= \mathrm{BV}(M) \cup \mathrm{BV}(N)
\end{aligned}$$

FV(M), the set of all *free variables* in M, is given by

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) \setminus \{x\} \\
\mathrm{FV}(MN) &= \mathrm{FV}(M) \cup \mathrm{FV}(N)
\end{aligned}$$

Substitution

$M[L/y]$, the result of substituting L for all free occurrences of y in M, is given by

$$\begin{aligned}
x[L/y] &\equiv \begin{cases} L & \text{if } x \equiv y \\ x & \text{otherwise} \end{cases} \\
(\lambda x.M)[L/y] &\equiv \begin{cases} (\lambda x.M) & \text{if } x \equiv y \\ (\lambda x.M[L/y]) & \text{otherwise} \end{cases} \\
(M\,N)[L/y] &\equiv (M[L/y]\ N[L/y])
\end{aligned}$$

Substitution notation is part of the *meta-language* (talks about the lambda calculus) *not* part of the lambda-calculus itself. (Later, we see *environments* as explicit representation of substitutions.)

Variable Capture – to be avoided

Substitution must not disturb variable binding. Consider $(\lambda x.(\lambda y.x))$. When applied to N it should give function $(\lambda y.N)$.

Fails for $N \equiv y$. This is *variable capture* – would make the lambda-calculus inconsistent mathematically.

The substitution $M[N/x]$ is safe provided the bound variables of M are disjoint from the free variables of N:

$$\mathrm{BV}(M) \cap \mathrm{FV}(N) = \emptyset.$$

We can always rename the bound variables of M, if necessary, to make this condition true.

Lambda Conversions

The idea that λ-abstractions represent functions is formally expressed through conversion rules for manipulating them. There are α-conversions, β-conversions and η-conversions.

The α-conversion $(\lambda x.M) \rightarrow_\alpha (\lambda y.M[y/x])$ renames the abstraction's bound variable from x to y. It is valid provided y does not occur (free or bound) in M. For example, $(\lambda x.(xz)) \rightarrow_\alpha (\lambda y.(yz))$. We shall usually ignore the distinction between terms that could be made identical by performing α-conversions.

The β-conversion $((\lambda x.M)N) \rightarrow_\beta M[N/x]$ substitutes the argument, N, into the abstraction's body, M. It is valid provided $\mathrm{BV}(M) \cap \mathrm{FV}(N) = \emptyset$. For example, $(\lambda x.(xx))(yz) \rightarrow_\beta ((yz)(yz))$.

β-conversion does all the 'real work'

Lambda Conversions (2)

The η-conversion $(\lambda x.(Mx)) \to_\eta M$ collapses the trivial function $(\lambda x.(Mx))$ down to M. It is valid provided $x \notin \mathrm{FV}(M)$. Thus, M does not depend on x; the abstraction does nothing but apply M to its argument. For example, $(\lambda x.((zy)x)) \to_\eta (zy)$.

Observe that the functions $(\lambda x.(Mx))$ and M always return the same answer, (MN), when applied to any argument N. The η-conversion rule embodies a principle of *extensionality*: two functions are equal if they always return equal results given equal arguments. In some situations, this principle (and η-conversions) are dispensed with.

[Think: is `fn x=>sin x` the same as `sin` in ML?]

Later, when we introduce constants into the lambda-calculus we will also add δ-reductions (zero or more for each constant) these also do ‘real work’.

Lambda Reductions

Lecture 2

Lambda Reduction (1)

Reductions: (non-α) conversions in *context*.

- Although we've introduced the conversions ($\rightarrow_\alpha$, $\rightarrow_\beta$, $\rightarrow_\eta$), they are not much use so far, e.g.

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

  does *not* convert. And it should!

- $((\lambda x.x)(\lambda y.y))$ already does though. Need to formalise.
- We also need to avoid people saying that $\lambda x.x$ reduces to $\lambda y.y$ reduces to $\lambda x.x$ … . It clearly does not *reduce* (think of 'make computational progress').

Lambda Reduction (2)

Fix 1. For each conversion (seen as an *axiom*) such as

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

we also add three *context rules* ($x = \alpha, \beta, \eta$)

$$\frac{M \rightarrow_x M'}{(\lambda x.M) \rightarrow_x (\lambda x.M')} \qquad \frac{M \rightarrow_x M'}{(MN) \rightarrow_x (M'N)} \qquad \frac{M \rightarrow_x M'}{(LM) \rightarrow_x (LM')}$$

(you've seen such rules before in "Operational Semantics").

BEWARE: the first of these context rules is very dodgy from a programming language viewpoint (a frequent source of diversion from the λ-calculus which has surprisingly little effect—because programming languages do not print or otherwise look inside functions, only apply them).

Lambda Reduction (3)

…we also add three *context rules*

$$\frac{M \rightarrow_x M'}{(\lambda x.M) \rightarrow_x (\lambda x.M')} \qquad \frac{M \rightarrow_x M'}{(MN) \rightarrow_x (M'N)} \qquad \frac{M \rightarrow_x M'}{(LM) \rightarrow_x (LM')}$$

FUTURE DIRECTION: we might also prioritise the 2nd and 3rd rules—lazy vs. eager evaluation.

E.g. $((\lambda x.xx)\,((\lambda y.y)(\lambda z.z)))$

Lambda Reduction (4)

Fix 2. (and DEFINITION) We say that $M \rightarrow N$, or M *reduces to* N, if $M \longrightarrow_\beta N$ or $M \longrightarrow_\eta N$. (Because α-conversions are not directional, and are not interesting, we generally ignore them—they don't do real work.)

BEWARE: η-reduction is also dodgy from a programming language point of view, but again it only affects functions and so tends to be harmless.

FUTURE: We treat reduction as including δ-reduction (for constants when we introduce them later).

Normal Forms

If a term admits no reductions then it is in *normal form*—think 'value'.

For example, $\lambda xy.y$ and xyz are in normal form.

To *normalise* a term means to apply reductions until a normal form is reached. A term *has a normal form* if it can be reduced to a term in normal form. For example, $(\lambda x.x)y$ is not in normal form, but it has the normal form y.

Many λ-terms cannot be reduced to normal form. For instance, $(\lambda x.xx)(\lambda x.xx)$ reduces to itself by β-conversion. This term is usually called $\mathbf{\Omega}$ ($\mathbf{\Omega}$ is an *abbreviation*—not part of the calculus).

Note that some terms which have normal forms, also have infinite reduction sequences, e.g. $(\lambda x.\lambda y.x)(\lambda z.z)\mathbf{\Omega}$.

Normal Forms (2)

For programming language purposes, the idea of *weak head normal form* (WHNF) is also of interest. A term is in WHNF if it admits no reduction (same as before) but when we *exclude* reductions reached by using the first context rule

$$\frac{M \rightarrow_x M'}{(\lambda x.M) \rightarrow_x (\lambda x.M')}$$

E.g. $\lambda x.\mathbf{\Omega}$ is in WHNF but not in NF. However $\mathbf{\Omega}$ is neither in NF nor in WHNF.

Currying

[You already know this from ML.]

Functions in the λ calculus take one argument and give one result. Two-argument (or more!) functions can be encoded via currying (actually due to Schönfinkel rather than Curry): if f abbreviates $\lambda x.\lambda y.L$ then

$$f\,M N \rightarrow_\beta (\lambda y.L[M/x]) \rightarrow_\beta L[M/x][N/y]$$

Multiple results (and an alternative way of doing arguments) can be achieved by pairing—see later how this can be simply encoded without additional magic.

Syntactic simplification

[You already know this from ML.]

Because λ-calculus evolved before programming languages, it is often presented (as we did) with excessive bracketing, e.g. $((\lambda x.(xy))\,(yz))$

However, omit brackets by use of operator precedence, treating:

- $\lambda xy.M$ as $(\lambda x.(\lambda y.M))$
- $L\,M\,N$ as $((L\,M)N)$
- $\lambda x.MN$ as $(\lambda x.(MN))$

BEWARE: $\lambda xyz.xyz$ is far from $(\lambda t.t)$. It is *literally* $(\lambda x.(\lambda y.(\lambda z.((xy)z))))$

Multi-step reduction

$$
\begin{array}{rcl}
M \twoheadrightarrow M' & \text{iff} & M \text{ can re-write to } M' \text{ using } k \geq 0\ (\rightarrow)\text{-steps} \\
 & \text{or} & \\
M \twoheadrightarrow M' & \text{iff} & (\exists k \geq 0, \exists M_0 \ldots M_k) M \equiv M_0 \rightarrow \cdots \rightarrow \equiv M_k \equiv M' \\
 & \text{or} & \\
(\twoheadrightarrow) & = & (\rightarrow)^* \quad \text{(reflexive/transitive closure)}
\end{array}
$$

Just like $\rightarrow$ this is not deterministic—consider $((\lambda x.xx)\ ((\lambda y.y)(\lambda z.z)))$ but *something* deterministic is happening (see confluence soon).

Equality

Two λ-terms, M and M', are *equal* $M = M'$ if there is a sequence of forwards and backwards reductions (backwards reductions are sometimes called expansions) from M to M'.

$$\begin{tikzcd}
M \arrow[dr] & & M_1 \arrow[dl] \arrow[dr] & & M_2 \cdots M_{k-1} \arrow[dl] \arrow[dr] & & M_k = M' \arrow[dl] \\
& N_1 & & N_2 & \cdots & N_k &
\end{tikzcd}$$

I.e.

$$\begin{aligned}
(=) &= ((\rightarrow) \cup (\rightarrow)^{-1})^* \quad \text{(reflexive/symmetric/transitive closure)} \\
&= ((\twoheadrightarrow) \cup (\twoheadrightarrow)^{-1})^*
\end{aligned}$$

Equality Properties

It's simple to show that equality satisfies all the expected things:

First of all, it is an *equivalence relation*—it satisfies the reflexive, symmetric and associative laws:

$$M = M \qquad \frac{M = N}{N = M} \qquad \frac{L = M \quad M = N}{L = N}$$

Furthermore, it satisfies congruence laws for each of the ways of constructing λ-terms:

$$\frac{M = M'}{(\lambda x.M) = (\lambda x.M')} \qquad \frac{M = M'}{(MN) = (M'N)} \qquad \frac{M = M'}{(LM) = (LM')}$$

I.e.

$$M = N \Rightarrow \mathcal{C}[M] == \mathcal{C}[N] \text{ for any context } \mathcal{C}[\cdots]$$

Equality (2)

Note $(M \twoheadrightarrow N \vee N \twoheadrightarrow M)$ suffices to imply $M = N$.

But $M = N$ does not imply $(M \twoheadrightarrow N \vee N \twoheadrightarrow M)$. (Exercise: why?)

Equality – bug in lecture notes

Definition 5 (page 8) is wrong. It needs to say ...

$$M = M \qquad \frac{M = N}{N = M} \qquad \frac{L = M \quad M = N}{L = N}$$

Furthermore, it satisfies congruence laws for each of the ways of constructing λ-terms:

$$\frac{M = M'}{(\lambda x.M) = (\lambda x.M')} \qquad \frac{M = M'}{(MN) = (M'N)} \qquad \frac{M = M'}{(LM) = (LM')}$$

Definition 5 *Equality of λ-terms* is the least relation satisfying the six rules above *and also* $M = N$ if there is a $(\alpha/\beta/\eta)$ conversion from M to N.

Church-Rosser Theorem – determinism

We saw the λ-calculus is non-deterministic locally, e.g. $((\lambda x.xx)\ ((\lambda y.y)(\lambda z.z)))$

But: "There's always a place (pub) to meet at downstream" (hence the alternative name "confluence").

Church-Rosser Theorem (BIG THEOREM): If $M = N$ then there exists L such that $M \twoheadrightarrow L$ and $N \twoheadrightarrow L$.

For instance, $(\lambda x.ax)((\lambda y.by)c)$ has two different reduction sequences, both leading to the same normal form. The affected subterm is underlined at each step:

$$\underline{(\lambda x.ax)((\lambda y.by)c)} \to a(\underline{(\lambda y.by)c}) \to a(bc)$$
$$(\lambda x.ax)(\underline{(\lambda y.by)c}) \to \underline{(\lambda x.ax)(bc)} \to a(bc)$$

Consequences

- If $M = N$ and N is in normal form, then $M \twoheadrightarrow N$; if a term can transform into normal form using reductions and expansions, then the normal form can be reached by reductions alone.
- If $M = N$ where both terms are in normal form, then $M \equiv N$ (up to renaming of bound variables). Conversely, if M and N are in normal form and are distinct, then $M \neq N$; there is no way of transforming M into N. For example, $\lambda xy.x \neq \lambda xy.y$.
- λ-calculus is *consistent* [nice to know!]

Diamond Properties

The key step in proving the Church-Rosser Theorem is demonstrating the diamond property—if $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$ then there exists a term L such that $M_1 \twoheadrightarrow L$ and $M_2 \twoheadrightarrow L$. Then 'tiling' suffices. Here is the diagram:

$$\begin{tikzcd}
 & M \arrow[dl, two heads] \arrow[dr, two heads] & \\
M_1 \arrow[dr, two heads] & & M_2 \arrow[dl, two heads] \\
 & L &
\end{tikzcd}$$

You might wonder that this would also be true of single-step reductions (as if there was independent concurrent evaluation in the two directions of the diamond). But...

Diamond Properties (2)

Note that $\rightarrow$ (one-step reduction) does *not* satisfy the diamond property

$$\begin{tikzcd} & M \arrow[dl] \arrow[dr] & \\ M_1 \arrow[dr] & & M_2 \arrow[dl] \\ & L & \end{tikzcd}$$

Consider the term $(\lambda x.xx)(\mathbf{I}\,a)$, where $\mathbf{I} \equiv \lambda x.x$. In one step, it reduces to $(\lambda x.xx)a$ or to $(\mathbf{I}\,a)(\mathbf{I}\,a)$.

Diamond Properties (3)

These both reduce eventually to aa, but there is no way to complete the diamond with a single-step reduction:

$$\begin{tikzcd}
 & (\lambda x.xx)(\mathbf{I}\,a) \arrow[dl] \arrow[dr] & \\
(\mathbf{I}\,a)(\mathbf{I}\,a) \arrow[dr, phantom, "\ddots"] & & (\lambda x.xx)a \arrow[dl] \\
 & aa &
\end{tikzcd}$$

Why? $(\lambda x.xx)$ replicates its argument, doubling the work needed.

Difficult cases involve one possible reduction contained inside another. Reductions that do not overlap, such as $M \to M'$ and $N \to N'$ in the term xMN, commute trivially to produce $xM'N'$. (Like concurrency.)

Confluence is trickily worded

Although different reduction sequences cannot yield different normal forms, they can yield completely different outcomes: one could terminate while the other runs forever!

For example, recall that $\mathbf{\Omega}$ reduces to itself, where $\mathbf{\Omega} \equiv (\lambda x.xx)(\lambda x.xx)$.

However, the reduction

$$(\lambda y.a)\mathbf{\Omega} \rightarrow a$$

reaches normal form, erasing the $\mathbf{\Omega}$. This corresponds to a *call-by-name* treatment of functions: the argument is not reduced but substituted 'as is' into the body of the abstraction.

Confluence is trickily worded (2)

Attempting to normalise the argument first generates a non-terminating reduction sequence:

$$(\lambda y.a)\mathbf{\Omega} \rightarrow (\lambda y.a)\mathbf{\Omega} \rightarrow \cdots$$

Evaluating the argument before substituting it into the body corresponds to a *call-by-value* treatment of function application. In this example, the call-by-value strategy never reaches the normal form.

So: *is there a best evaluation strategy, or do we have to search like Prolog does . . . ?*

Normal Order Reduction

Is there a best evaluation strategy (gives a normal form if one exists)?

Yes—Normal Order Reduction.

The *normal order* reduction strategy is, at each step, to perform the leftmost outermost β-reduction. (The η-reductions can be left until last.) *Leftmost* means, for instance, to reduce L before N in LN. *Outermost* means, for instance, to reduce $(\lambda x.M)N$ before reducing M or N.

Normal order reduction (*almost*) corresponds to call-by-name evaluation. The "Standardisation Theorem" states that it always reaches a normal form if one exists. As a flavour as to why: we have to erase as many computations as we can (so they don't explode)—so in LN we reduce L first because it may transform into an abstraction, say $\lambda x.M$. Reducing $(\lambda x.M)N$ may erase N.

Reduction Strategies

- Call-by-value (eager) evaluation, always evaluates the argument to a function exactly once. But may spuriously look.
- Call-by-name (normal order) evaluation, always gives a result if one exists (but can be exponentially slower as it evaluates function arguments $0, 1, 2, \ldots$ times.

Compromise?

- Call-by-need (lazy) evaluation, represent λ-terms as a (shared) graph. Evaluate the first time they are encountered and save the result (e.g. by over-writing) for subsequent uses. (Duplication of terms caused by β-reduction need not duplicate computation.)

Encoding Data in the λ-Calculus

Lecture 3

More on intuition behind the course

- λ-calculus is a very simple programming language which is the basis either explicitly (ISWIM, ML) or implicitly for many programming languages.
- λ-calculus is used as a (meta-language) notation for explaining various other bits of CS.
- ‘fun’ (or at least programming-like) things happen in it.
- a bridge towards compiler notions such as environment.

Encoding Data in the λ-Calculus

The λ-calculus is expressive enough to encode boolean values, ordered pairs, natural numbers and lists—all the data structures we may desire in a functional program. These encodings allow us to model virtually the whole of functional programming within the simple confines of the λ-calculus.

The encodings may not seem to be at all natural, and they certainly are not computationally efficient. In this, they resemble Turing machine encodings and programs. Unlike Turing machine programs, the encodings are themselves of mathematical interest, and return again and again in theoretical studies. Many of them involve the idea that the data can carry its control structure with it. (Think *iterators* in C++.)

Once understood, these constructs can be hard-coded (see later).

Encoding Booleans

It's not obvious that we can even encode if-then-else in λ-calculus (think about it).

An encoding of the booleans must define the terms **true**, **false** and **if**, satisfying (for all M and N)

$$\begin{aligned}\textbf{if}\ \textbf{true}\, M N &= M\\ \textbf{if}\ \textbf{false}\, M N &= N.\end{aligned}$$

Note **true**, **false** and **if**, are just abbreviations for λ-terms and not (yet!) constants or anything else within the λ-calculus.

Note also the use of '=' (recall what it means).

Encoding Booleans (2)

The following encoding is usually adopted:

$$\begin{aligned} \textbf{true} &\equiv \lambda xy.x \\ \textbf{false} &\equiv \lambda xy.y \\ \textbf{if} &\equiv \lambda pxy.pxy \end{aligned}$$

We have **true** $\neq$ **false** by the Church-Rosser Theorem, since **true** and **false** are distinct normal forms. As it happens, **if** is not even necessary. The truth values are their own conditional operators:

$$\textbf{true}\, MN \equiv (\lambda xy.x)MN \twoheadrightarrow M$$

$$\textbf{false}\, MN \equiv (\lambda xy.y)MN \twoheadrightarrow N$$

Encoding Booleans (3)

These reductions hold for all terms M and N, whether or not they possess normal forms (step through it!). Note that **if** $LMN \twoheadrightarrow LMN$; it is essentially an identity function on L. The equations given above even hold as reductions:

$$\begin{aligned} \textbf{if true}\, MN &\twoheadrightarrow M \\ \textbf{if false}\, MN &\twoheadrightarrow N. \end{aligned}$$

This is nice, because equations ('=') might mean forwards and backwards reduction. Using $\twoheadrightarrow$ means 'computation like'.

Indeed using *Normal Order Reduction* starting with (e.g.)

$$\textbf{if true}\, MN$$

will eventually (around 4 steps—try it) produce M to reduce next. This isn't true for all evaluation strategies.

More Booleans

Once we have defined the essential parts of an implementation of an ADT/interface additional operators can be defined in terms of the core ones. E.g. all the usual operations on truth values can be defined in terms of the conditional operator. Here are negation, conjunction and disjunction:

$$\begin{aligned} \mathbf{and} &\equiv \lambda pq.\,\mathbf{if}\; p\; q\; \mathbf{false} \\ \mathbf{or} &\equiv \lambda pq.\,\mathbf{if}\; p\; \mathbf{true}\; q \\ \mathbf{not} &\equiv \lambda p.\,\mathbf{if}\; p\; \mathbf{false}\; \mathbf{true} \end{aligned}$$

More Data Types

Idea: repeat the trick above, but at each stage we can use not only the λ-calculus itself, but the encodings already derived.

Encoding Pairs

Assume that **true** and **false** are defined as above. The function **pair**, which constructs pairs, and the projections **fst** and **snd**, which select the components of a pair, are encoded as follows:

$$\begin{array}{rcl}\textbf{pair} & \equiv & \lambda xyf.fxy \\ \textbf{fst} & \equiv & \lambda p.p\,\textbf{true} \\ \textbf{snd} & \equiv & \lambda p.p\,\textbf{false}\end{array}$$

See how **pair** squirrels away x and y waiting for an f to select one of them.

Encoding Sums—ML constructors

There are many ways of doing this, but a simple way is to encode the 'tag field' using a boolean:

$$\begin{aligned}\mathbf{inl} &\equiv \lambda x.\,\mathbf{pair}\;\mathbf{true}\,x\\ \mathbf{inr} &\equiv \lambda y.\,\mathbf{pair}\;\mathbf{false}\,y\\ \mathbf{case} &\equiv \lambda s f g = \mathbf{if}\,(\,\mathbf{fst}\,s)(f(\,\mathbf{snd}\,s))(g(\,\mathbf{snd}\,x))\end{aligned}$$

Encoding Natural Numbers

The following encoding of the natural numbers is the original one developed by Church. Alternative encodings are sometimes preferred today, but Church's numerals continue our theme of putting the control structure in with the data structure.

Such encodings are elegant; moreover, all these work in the second-order (typed) λ-calculus (presented in the Types course in CST Part II).

Encoding Natural Numbers (2)

The following encoding of the natural numbers is the original one
Define

$$\begin{aligned} \underline{0} &\equiv \lambda f x . x \\ \underline{1} &\equiv \lambda f x . f x \\ \underline{2} &\equiv \lambda f x . f(f x) \\ \vdots\ & \ \vdots \quad \vdots \\ \underline{n} &\equiv \lambda f x . \underbrace{f(\cdots(f}_{n \text{ times}} x)\cdots) \end{aligned}$$

Thus, for all $n \geq 0$, the Church numeral $\underline{n}$ is the function that maps f to f^n. Each numeral is an iteration operator (see how $\underline{n}$ acts as a sort of 'map' function with argument f).

Remember also the CST Part IA exercises using these?

Arithmetic on Church Numerals

Just having numbers isn't enough, we need to do arithmetic. Can Church numerals do this? Using this encoding, addition, multiplication and exponentiation can be defined immediately:

$$\begin{array}{rcl} \textbf{add} & \equiv & \lambda mnfx.mf(nfx) \\ \textbf{mult} & \equiv & \lambda mnfx.m(nf)x \\ \textbf{expt} & \equiv & \lambda mnfx.nmfx \end{array}$$

Addition is easy to validate:

$$\begin{array}{rcl} \textbf{add}\ \underline{m}\ \underline{n} & \twoheadrightarrow & \lambda fx.\underline{m}\,f(\underline{n}\,fx) \\ & \twoheadrightarrow & \lambda fx.f^m(f^n x) \\ & \equiv & \lambda fx.f^{m+n}x \\ & \equiv & \underline{m+n} \end{array}$$

More Arithmetic

Sadly, just being able to do addition, multiplication and exponentiation doesn't give full (Turing) computational power. Register machines (also Turing powerful) exploit increment, decrement and test-for-zero:

$$\begin{aligned} \mathbf{suc} &\equiv \lambda n f x. f(n f x) \\ \mathbf{iszero} &\equiv \lambda n. n(\lambda x.\, \mathbf{false}\,)\, \mathbf{true} \end{aligned}$$

The following reductions hold for every Church numeral $\underline{n}$:

$$\begin{aligned} \mathbf{suc}\ \underline{n} &\twoheadrightarrow \underline{n+1} \\ \mathbf{iszero}\ \underline{0} &\twoheadrightarrow \mathbf{true} \\ \mathbf{iszero}\,(\,\underline{n+1}\,) &\twoheadrightarrow \mathbf{false} \end{aligned}$$

But what about **pre**?

Encoding Predecessor

It's not obvious how we can convert a function which self-composes its argument n times into one which does so $n-1$ times (of course $n > 0$)

But it is possible. Here's my trick (different from the notes). Take an argument x, pair it with **true** (a 'first time' flag) and manufacture function g which flips the flag if it is true and *otherwise* applies f to the x component of the pair, finally dropping the flag before returning (thus doing f a total of $n-1$ times):

$$\begin{aligned}\mathbf{pre} \quad &\equiv \quad \lambda n f x.\, \mathbf{snd}\,(n \\ &\qquad (\lambda y.\, \mathbf{if}\,(\,\mathbf{fst}\, y)(\,\mathbf{pair\ false}\,(\,\mathbf{snd}\, y))(\,\mathbf{pair\ false}\,(f(\,\mathbf{snd}\, y)))) \\ &\qquad (\,\mathbf{pair\ true}\, x))\end{aligned}$$

Wheee! This is programming. (I could even use **case** here.)

Encoding Subtraction

Subtraction is just repeated predecessor:

$$\mathbf{sub} \quad \equiv \quad \lambda mn.n\,\mathbf{pre}\,m$$

You might want to think about what $\mathbf{sub}\ \underline{4}\ \underline{6}$ is. We know it is a λ-expression (there is nothing else yet it can be, no "bus error, core dumped" message). It may or may not have a normal form, and if it does, it may or may not represent a Church numeral. It's just not defined mathematically as we can't represent negative numbers yet, so getting a junk result is quite acceptable.

Of course, I could encode signed numbers by $\mathbf{pair}\,b\,\underline{n}$ where b is a boolean, and even IEEE floating point arithmetic, but that's perhaps a bit overkeen use of a coding demonstrating theoretical power rather than run-time efficiency!

Encoding Lists

We could encode the number n as the list $[(), \ldots, ()]$ in ML. Church numerals could similarly be generalized to represent lists. The list $[x_1, x_2, \ldots, x_n]$ would essentially be represented by the function that takes f and y to $fx_1(fx_2 \ldots (fx_n y) \ldots)$. Such lists would carry their own control structure with them.

But, we've defined pairing and sums (tagged union), so let's use those . . .

Encoding Lists (2)

Here is our encoding of lists:

$$\begin{aligned}
\mathbf{nil} &\equiv \lambda z.z \\
\mathbf{cons} &\equiv \lambda xy.\,\mathbf{pair}\ \mathbf{false}\,(\,\mathbf{pair}\,xy) \\
\mathbf{null} &\equiv \mathbf{fst} \\
\mathbf{hd} &\equiv \lambda z.\,\mathbf{fst}\,(\,\mathbf{snd}\,z) \\
\mathbf{tl} &\equiv \lambda z.\,\mathbf{snd}\,(\,\mathbf{snd}\,z)
\end{aligned}$$

This accidentally works; we really should have used

$$\mathbf{nil} \equiv \mathbf{pair}\ \mathbf{true}\,\lambda z.z$$

where the $\lambda z.z$ represents ‘()’ and won’t be used in the computation.

Encoding Lists (3)

The following properties are easy to verify; they hold for all terms M and N:

$$\begin{aligned}
\mathbf{null\ nil} &\twoheadrightarrow \mathbf{true}\\
\mathbf{null}\,(\mathbf{cons}\,M\,N) &\twoheadrightarrow \mathbf{false}\\
\mathbf{hd}\,(\mathbf{cons}\,M\,N) &\twoheadrightarrow M\\
\mathbf{tl}\,(\mathbf{cons}\,M\,N) &\twoheadrightarrow N
\end{aligned}$$

Note that $\mathbf{null\ nil} \twoheadrightarrow \mathbf{true}$ happens really by chance, while the other laws hold by our operations on pairs.

Recursion

We've defined almost everything used in programming, except recursion. How can recursive functions be defined?

Recursion and HNF

UNIVERSITY OF
CAMBRIDGE

Lecture 4

Recursion (2)

Recursion is obviously essential in functional programming. The traditional way to express recursion is a *fixed point combinator*, **Y** which essentially maps

$$\mathsf{letrec}\ f(x) = M\ \mathsf{in}\ N$$

into

$$\mathsf{let}\ f = \mathbf{Y}\,(\lambda f.\lambda x.M)\ \mathsf{in}\ N$$

or

$$(\lambda f.N)(\,\mathbf{Y}\,\lambda f.\lambda x.M)$$

By using this idea along with **suc** , **pre** and **iszero** we can encode any Register Machine from Computation Theory as a λ-term and hence the λ-calculus has at least the power of Turing machines and Register Machines (actually equipotent as the SECD machine given below can be coded as a Register Machine fairly simply).

Recursion (3)

Slick answer: we almost almost almost don't need recursion as things like Church numerals essentially code C++ iterators. Even Ackermann's function (see Computation Theory) can be defined:

$$\mathbf{ack} \equiv \lambda m.m(\lambda fn.nf(f\,\underline{1}))\,\mathbf{suc}$$

But hey, this is hard, and we did say "almost".

Recursion (4)

Why can't we just say:

$$\textbf{fact} \quad \equiv \quad \lambda f.\lambda x.\,\textbf{if}\,(\,\textbf{iszero}\,x)\,\underline{1}\,(\,\textbf{mult}\,x(\,\textbf{fact}\,(\,\textbf{pre}\,x)))?$$

Because **fact** would then have an infinite number of symbols and all λ-terms (just like programs) are of finite size.

(Why: the RHS has about 100 more characters in it than the LHS (**fact**) does.)

Recursion—the $\mathbf{Y}$ operator

More general answer: use a *fixed point combinator*—a term $\mathbf{Y}$ such that $\mathbf{Y}\,F = F(\mathbf{Y}\,F)$ for all terms F.

Terminology: A *fixed point* of the function F is any X such that $F X = X$; here, $X \equiv \mathbf{Y}\,F$. A *combinator* (here, but see later) is any λ-term containing no free variables (also called a closed term).

To code recursion, F represents the body of the recursive definition; the law $\mathbf{Y}\,F = F(\mathbf{Y}\,F)$ permits F to be unfolded as many times as necessary.

But does such a $\mathbf{Y}$ exist?

Recursion—the **Y** operator (2)

Example: consider (intuitively, or using Boolean and Church numeral encodings)

$$F \quad \equiv \quad \lambda f.\lambda x.\,\mathbf{if}\,(\,\mathbf{iszero}\,x)\,\underline{1}\,(\,\mathbf{mult}\,x(f(\,\mathbf{pre}\,x)))$$

and consider $F(\lambda x.\,\underline{1}\,)$.

$(\lambda x.\,\underline{1}\,)$ agrees with the factorial function for 0 and 1, but $F(\lambda x.\,\underline{1}\,)$ agrees with the factorial function for 0, 1 and 2.

Indeed, if f agrees with factorial for argument values $0..n$ then $F f$ agrees with factorial for $0..n+1$ (this is even true if f did not even agree with factorial at 0).

So the 'only' function on natural numbers satisfying $f = F f$ is factorial, hence we can define **fact** $=$ **Y** F.

But does such a **Y** exist?

Recursion—the Y operator (3)

More examples:

We shall encode the factorial function, the append function on lists, and the infinite list $[0, 0, 0, \ldots]$ in the λ-calculus, realising the recursion equations

$$\begin{array}{rcl} \mathbf{fact}\, N & = & \mathbf{if}\,(\,\mathbf{iszero}\, N)\,\underline{1}\,(\,\mathbf{mult}\, N(\,\mathbf{fact}\,(\,\mathbf{pre}\, N))) \\ \mathbf{append}\, ZW & = & \mathbf{if}\,(\,\mathbf{null}\, Z)W(\,\mathbf{cons}\,(\,\mathbf{hd}\, Z)(\,\mathbf{append}\,(\,\mathbf{tl}\, Z)W\,)) \\ \mathbf{zeroes} & = & \mathbf{cons}\ \underline{0}\ \mathbf{zeroes} \end{array}$$

These lines are *hopes*, not yet proper definitions, as they are self-referential.

Recursion—the Y operator (4)

To realize these, we simply put

$$\begin{aligned}
\textbf{fact} &\equiv \mathbf{Y}\,(\lambda gn.\,\textbf{if}\,(\,\textbf{iszero}\,n)\,\underline{1}\,(\,\textbf{mult}\,n(g(\,\textbf{pre}\,n))))\\
\textbf{append} &\equiv \mathbf{Y}\,(\lambda gzw.\,\textbf{if}\,(\,\textbf{null}\,z)w(\,\textbf{cons}\,(\,\textbf{hd}\,z)(g(\,\textbf{tl}\,z)w)))\\
\textbf{zeroes} &\equiv \mathbf{Y}\,(\lambda g.\,\textbf{cons}\,\underline{0}\,g)
\end{aligned}$$

These now *are* definitions, if **Y** exists.

But they won't have a NF!

Recursion—$\mathbf{Y}$ exists (1)

UNIVERSITY OF CAMBRIDGE

The combinator $\mathbf{Y}$ was discovered by Haskell B. Curry. It is defined by

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Let us calculate to show the fixed point property:

$$\begin{aligned} \mathbf{Y}\,F &\rightarrow (\lambda x.F(xx))(\lambda x.F(xx)) \\ &\rightarrow F((\lambda x.F(xx))(\lambda x.F(xx))) \\ &= F(\mathbf{Y}\,F) \end{aligned}$$

This consists of two β-reductions followed by a β-expansion. No reduction $\mathbf{Y}\,F \twoheadrightarrow F(\mathbf{Y}\,F)$ is possible!

Recursion—Y exists (2)

There are other fixed point combinators, such as Alan Turing's Θ:

$$\begin{aligned} A &\equiv \lambda xy.y(xxy) \\ \Theta &\equiv AA \end{aligned}$$

This has the reduction $\Theta F \twoheadrightarrow F(\Theta F)$:

$$\Theta F \equiv AAF \twoheadrightarrow F(AAF) \equiv F(\Theta F)$$

But **Y** has no Normal Form!

UNIVERSITY OF CAMBRIDGE

If $M = xM$ then M has no normal form. For if $M \twoheadrightarrow N$ where N is in normal form, then $N = xN$. Since xN is also in normal form, the Church-Rosser Theorem gives us $N \equiv xN$. But again N cannot contain itself as a subterm (count the number of its symbols)!

By similar reasoning, if $M = PM$ then M usually has no normal form, unless P is something like a constant function or identity function. So anything defined with the help of a fixed point combinator, such as **fact** , is unlikely to have a normal form.

Although **fact** has no normal form, we can still compute with it; **fact** $\underline{5}$ does have a normal form, namely $\underline{120}$. We can use infinite objects (including functions as above, and also lazy lists) by computing with them for a finite time and requesting a finite part of the result.

But **Y** has no Normal Form! (2)

The Goldilocks analogy:

- saying “nice λ-terms have a NF” is too strong (because we want **fact** to be considered nice).
- saying “nice λ-terms have a WHNF” is too weak (because we probably don’t want $\lambda x.\mathbf{\Omega}$ to be considered nice).
- Instead we say “nice λ-terms have a *head normal form* (HNF)”.

Digression – 5 slides

Only the definition of HNF (and NF, WHNF) is examinable this year!

Historically, people wanted to say "Nice terms have a NF, and their 'value' is that NF (which is unique)", and *then* all terms without an NF are nasty and should all be treated as equal".

fact lacks a normal form, as do similar (but different functions), such as the function to calculate the nth triangular number (same text as **fact** but with multiply replaced with addition).

So, failing to have a NF isn't too bad.

Digression continued

It turns out that terms without a WHNF (earlier) truly are useless—they all loop, and so after often equated (indeed courses write $\perp$ for this).

But historically, this isn't how things were done, and the concept of *head normal form* [intermediate in strength between HF and WNHF] captures things better.

Terms without a HNF can be seen as useless. Note that both **Y** and **fact** *do* have a HNF.

Intuition. Given a closed (no FV's) term in HNF we can always get a NF by applying it to suitable arguments. This is not true of $\lambda x.\mathbf{\Omega}$.

Head Normal Form

A term is in *head normal form* (HNF) if and only if it looks like this:

$$\lambda x_1 \ldots x_m . y M_1 \ldots M_k \qquad (m,\, k \geq 0)$$

Examples of terms in HNF include

$$x \qquad \lambda x.y\mathbf{\Omega} \qquad \lambda x\, y.x \qquad \lambda z.z((\lambda x.a)c)$$

But $\lambda y.(\lambda x.a)y$ is not in HNF because it admits the so-called *head reduction*

$$\lambda y.(\lambda x.a)y \to \lambda y.a.$$

Some obvious facts: a term in normal form is also in head normal form; a term in head normal form is also in weak head normal form.

Head Normal Form (2)

Some terms do not even have a head normal form. Recall $\mathbf{\Omega}$, defined by $\mathbf{\Omega} = (\lambda x.xx)(\lambda x.xx)$. A term is reduced to HNF by repeatedly performing leftmost reductions. With $\mathbf{\Omega}$ we can only do $\mathbf{\Omega} \to \mathbf{\Omega}$, which makes no progress towards an HNF. Another term that lacks an HNF is $\lambda y.\mathbf{\Omega}$; we can only reduce $\lambda y.\mathbf{\Omega} \to \lambda y.\mathbf{\Omega}$.

It can be shown that if MN has an HNF then so does M. Therefore, if M has no HNF then neither does any term of the form $MN_1N_2 \ldots N_k$. A term with no HNF behaves like a *totally undefined function*: no matter what you supply as arguments, evaluation never returns any information. It is not hard to see that if M has no HNF then neither does $\lambda x.M$ or $M[N/x]$, so M really behaves like a black hole. The only way to get rid of M is by a reduction such as $(\lambda x.a)M \to a$. This motivates the following definition of *definedness*.

Head Normal Form (3)

Some technical stuff – interesting to some people – background here.

A term is *defined* if and only if it can be reduced to head normal form; otherwise it is *undefined*.

A term M is called *solvable* if and only if there exist variables x_1, …, x_m and terms N_1, …, N_n such that

$$(\lambda x_1 \ldots x_m.M)N_1 \ldots N_n = \mathbf{I}\,.$$

Deep theorem: a term is defined iff is it solvable.

Recursion—Summary

To express a recursive definition, e.g.

$$\mathsf{letrec}\ d = M\ \mathsf{in}\ N$$

we treat it as

$$\mathsf{let}\ d = \ \mathbf{Y}\,(\lambda d.M)\ \mathsf{in}\ N$$

or

$$(\lambda d.N)(\,\mathbf{Y}\,\lambda d.M)$$

The special case of $M \equiv \lambda x.M'$ being a λ-abstraction is common (and is required in ISWIM/ML – modulo mutual recursion using pairing below).

Mutual recursion can be achieved by constructing a fixed point which is a pair (or triple etc.), often a pair (etc.) of functions.

ISWIM – λ-calculus as a Programming Language

Landin [1966] “The Next 700 Programming Languages”(!) observed that λ-calculus explained much of the existing 700 programming languages; proposed ISWIM “If you See What I Mean” as basis for the next 700.

Hugely influential – and ML is essentially ISWIM.

ISWIM—Syntax

ISWIM started with the λ-calculus:

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

It also allowed local declarations:

$\mathsf{let}\ x = M\ \mathsf{in}\ N$	simple declaration
$\mathsf{let}\ f\, x_1 \cdots x_k = M\ \mathsf{in}\ N$	function declaration
$\mathsf{letrec}\ f\, x_1 \cdots x_k = M\ \mathsf{in}\ N$	recursive declaration

ISWIM—Syntax (2)

Local declarations could be post-hoc:

$$N \ \textbf{where} \ x = M$$

$$N \ \textbf{where} \ f\, x_1 \cdots x_k = M$$

$$N \ \textbf{whererec} \ f\, x_1 \cdots x_k = M$$

Big language? No...

ISWIM—Syntactic Sugar

UNIVERSITY OF CAMBRIDGE

$$\begin{aligned}
N \ \textbf{where}\ x = M &\equiv \mathsf{let}\ x = M\ \mathsf{in}\ N \qquad \text{(etc.)}\\
\mathsf{let}\ x = M\ \mathsf{in}\ N &\equiv (\lambda x.N)M\\
\mathsf{let}\ f\,x_1\cdots x_k = M\ \mathsf{in}\ N &\equiv \mathsf{let}\ f = \lambda x_1\cdots x_k.M\ \mathsf{in}\ N\\
&\equiv (\lambda f.N)(\lambda x_1\cdots x_k.M)\\
\mathsf{letrec}\ f\,x_1\cdots x_k = M\ \mathsf{in}\ N &\equiv \mathsf{letrec}\ f = \lambda x_1\cdots x_k.M\ \mathsf{in}\ N\\
&\equiv \mathsf{let}\ f = \mathbf{Y}\,(\lambda f.\lambda x_1\cdots x_k.M)\ \mathsf{in}\ N\\
&\equiv (\lambda f.N)(\mathbf{Y}\,(\lambda f x_1\cdots x_k.M))
\end{aligned}$$

Desugaring explains new syntax by translation into a smaller core language; avoids unexpected behaviour from complex interactions of features.

ISWIM—Constants

Programmers did not have to use Church numerals either, constants were built-in which had the same effect. So core syntax is now

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

Constants include

$0\ 1\ {-1}\ 2\ {-2}\ \ldots$	integers
$+\ -\ \times\ /$	arithmetic operators
$=\ \neq\ <\ >\ \leq\ \geq$	relational operators
true **false**	booleans
and **or** **not**	boolean connectives

Extra ‘syntax’ (see later for desugaring):

if E then M else N conditional

ISWIM—Constants (2)

Constants reduce by each having one or more δ-reduction rules, e.g. $+\ 0\ 0 \rightarrow_\delta 0$ and $+\ 5\ 6 \rightarrow_\delta 11$.

We would want these to be nice (e.g. deterministic, i.e. forbidding things like $\oplus\ 0 \rightarrow_\delta 0$ with $\oplus\ 0 \rightarrow_\delta 1$); modelling λ-expressible things would guarantee this.

So, ISWIM is just the λ-calculus with constants, but historically with eager evaluation (call-by-value). ISWIM also acquired mutable operations like ML's (ref, :=, etc.) – it's hard to reason about these with lazy evaluation.

ISWIM—Constants (3)

The earlier primitives for if-then-else does not work properly with eager evaluation. Consider **if true** $\underline{1}\,\boldsymbol{\Omega}$.

So, adopt a new desugaring (assuming x does not appear in M,N):

$$\mathsf{if}\,E\ \mathsf{then}\ M\ \mathsf{else}\ N \quad\equiv\quad (\,\mathbf{if}\ \ E\ (\lambda x.M)\ (\lambda x.N))\ (\lambda z.z)$$

This is suitable for call-by-value. Note that we often (as in ML) replace x and $\lambda z.z$ with ()—once we have defined the empty tuple!

ISWIM—Pattern Matching

Pairing and unpairing is done by constants ((M, N) is sugar for **pair** $M\,N$)

(M, N)	pair constructor
fst **snd**	projection functions

For pattern-matching, $\lambda(p_1, p_2).E$ desugars to

$$\lambda z.(\lambda p_1\, p_2.E)(\,\mathbf{fst}\,z)(\,\mathbf{snd}\,z)$$

where p_1 and p_2 may themselves be patterns. Thus, we may write

let $(x, y) = M$ in E	taking apart M's value
let $f(x, y) = E$ in N	defining f on pairs

n-tuples $(M_1, ...M_n)$ desugar just to iterated pairs $(M_1, (M_2, ...M_n))$.

ISWIM—Mutual Recursion

Just use the desugarings so far:

$$\begin{array}{ll}
\mathsf{letrec} & f_1\,\vec{x}_1 = M_1 \\
\mathsf{and} & f_2\,\vec{x}_2 = M_2 \\
\vdots & \\
\mathsf{and} & f_k\,\vec{x}_k = M_k \\
\mathsf{in}\ \ N &
\end{array}$$

can be translated to an expression involving pattern-matching:

$$(\lambda(f_1,\ldots,f_k).N)(\,\mathbf{Y}\,(\lambda(f_1,\ldots,f_k).(\lambda\vec{x}_1.M_1,\lambda\vec{x}_2.M_2,\ldots,\lambda\vec{x}_k.M_k)))$$

So, all this 'normal ML-style language' is really only the λ-calculus with constants (and syntactic desugaring).

ISWIM—Mutability

Practically all programming language features, including goto statements and pointer variables, can be formally specified in the λ-calculus—*denotational semantics* uses is as its meta-language.

ISWIM is much simpler than that; it is programming directly in the λ-calculus. To allow imperative programming, we can even define sequential execution, letting $M; N$ abbreviate $(\lambda x.N)M$; the call-by-value rule will evaluate M before N. However, imperative operations must be adopted as primitive; they cannot be defined by *simple* translation into the λ-calculus.

We'll see how control-flow can be encoded within the λ-calculus when we study continuations, and explicit stores enable us to capture mutable objects.

Real λ-evaluators

Lecture 5

Real λ-evaluators

We want a real λ-evaluator to execute ISWIM, which is a real programming language.

What do we take as values: constants and functions. But for a programming language we don't want to reduce inside a function body until is it called. So 'function values' means λ-expressions in WHNF (not HNF or NF) which evaluate inside a function before it is called.

We don't use the β-rule's textual substitution on programs on efficiency grounds—instead we do *delayed* substitution by using an *environment* and ensure that the expressions encountered by the evaluator are all subterms of the original program (since there are only a finite number of these they may be compiled to (real or abstract) machine code. Later we will see combinator evaluation as another way to avoid variables and hence substitution.

Environments and Closures

Consider the (β-)reduction sequence

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

Substitution is too slow to be effective for parameter passing; instead, the SECD machine records $x = 3$ in an *environment*.

With curried functions, $(\lambda xy.x+y)\,3$ is a legitimate value. We represent it by a *closure*, packaging the λ-abstraction with its current environment:

$$\mathbf{Clo}(\underbrace{y}_{\uparrow\ \text{bound variable}},\ \underbrace{x+y}_{\uparrow\ \text{function body}},\ \underbrace{x=3}_{\uparrow\ \text{environment}})$$

Environments and Closures (2)

$$\begin{array}{lcccccc} \textbf{Clo}(& y & , & x+y & , & x=3 &) \\ & \uparrow & & \uparrow & & \uparrow & \\ & \text{bound variable} & & \text{function body} & & \text{environment} & \end{array}$$

When an interpreter applies this function value to the argument 5, it restores the environment to $x = 3$, adds the binding $y = 5$, and evaluates $x + y$ in this augmented environment.

A closure is so-called because it "closes up" the function body over its free variables. This operation is costly; most programming languages forbid using functions as values—indeed Lisp historically cheated here...

Static and Dynamic Binding

UNIVERSITY OF CAMBRIDGE

Traditionally, many versions of Lisp let a function's free variables pick up any values they happened to have in the environment of the call (not that of the function's definition!); with this approach, evaluating

$$\begin{array}{ll}\mathsf{let}\ x = 1 & \mathsf{in} \\ \mathsf{let}\ g(y) = x + y & \mathsf{in} \\ \mathsf{let}\ f(x) = g(1) & \mathsf{in} \\ f(17) & \end{array}$$

would return 18, using 17 as the value of x in g! This is *dynamic binding*, as opposed to the usual *static binding*. Dynamic binding is confusing because the scope of x in $f(x)$ can extend far beyond the body of f—it includes all code reachable from f (including g in this case).

The SECD Machine

UNIVERSITY OF CAMBRIDGE

A Virtual Machine for ISWIM (the IVM?!?)—also due to Landin!

The SECD machine has a state consisting of four components (think 'machine registers') S, E, C, D:

- The *Stack* is a list of values, typically operands or function arguments; it also returns the result of a function call.
- The *Environment* has the form $x_1 = a_1; \cdots ; x_n = a_n$, expressing that the variables $x_1, \ldots, x_n$ have the values $a_1, \ldots, a_n$, respectively.
- The *Control* is a list of commands. For the interpretive SECD machine, a command is a λ-term or the word **app**; the compiled SECD machine has many commands.

The SECD Machine (2)

- The *Dump* is 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), (S_2, E_2, C_2), $\ldots$, (S_n, E_n, C_n) and serves as the function call stack.

The SECD Machine—state transitions

Let us write SECD machine states as boxes:

Stack
Environment
Control
Dump

The SECD Machine—state transitions (2)

To evaluate the λ-term M, the machine begins execution an the *initial state* where M is the Control (left):

S	–
E	–
C	M
D	–

S	a
E	–
C	–
D	–

The right diagram shows a *final state* from which the result of the evaluation, say a, is obtained (the Control and Dump are empty, and a is the sole value on the Stack).

If the Control is non-empty, then its first command triggers a state transition. There are cases for constants, variables, abstractions, applications, and the **app** command.

The SECD Machine—state transitions (3)

A constant is pushed on to the Stack (left): the value of a variable is taken from the Environment and pushed on to the Stack. If the variable is x and E contains $x = a$ then a is pushed (right):

$$\begin{array}{|c|}\hline S\\\hline E\\\hline k;C\\\hline D\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline k;S\\\hline E\\\hline C\\\hline D\\\hline\end{array} \qquad\qquad \begin{array}{|c|}\hline S\\\hline E\\\hline x;C\\\hline D\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline a;S\\\hline E\\\hline C\\\hline D\\\hline\end{array}$$

The SECD Machine—state transitions (4)

A λ-abstraction is converted to a closure, then pushed on to the Stack. The closure contains the current Environment (left):

$$\begin{array}{|c|}\hline S\\\hline E\\\hline \lambda x.M;C\\\hline D\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline \mathbf{Clo}(x,M,E);S\\\hline E\\\hline C\\\hline D\\\hline\end{array} \qquad \begin{array}{|c|}\hline S\\\hline E\\\hline M\,N;C\\\hline D\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline S\\\hline E\\\hline N;M;\mathbf{app};C\\\hline D\\\hline\end{array}$$

A function application is replaced by code to evaluate the argument and the function, with an explicit **app** instruction (right).

The SECD Machine—state transitions (5)

The **app** command calls the function on top of the Stack, with the next Stack element as its argument. A primitive function, like $+$ or $\times$, delivers its result immediately:

$$\begin{array}{|c|}\hline f;a;S \\ \hline E \\ \hline \mathbf{app};C \\ \hline D \\ \hline \end{array} \quad \longmapsto \quad \begin{array}{|c|}\hline f(a);S \\ \hline E \\ \hline C \\ \hline D \\ \hline \end{array}$$

The SECD Machine—state transitions (6)

The closure $\mathbf{Clo}(x, M, E')$ is called by creating a new state to evaluate M in the Environment E', extended with a binding for the argument. The old state is saved in the Dump:

$$\begin{array}{|c|}\hline \mathbf{Clo}(x, M, E'); a; S \\ \hline E \\ \hline \mathbf{app}; C \\ \hline D \\ \hline \end{array} \longmapsto \begin{array}{|c|}\hline - \\ \hline x = a; E' \\ \hline M \\ \hline (S, E, C, D) \\ \hline \end{array}$$

The SECD Machine—state transitions (7)

The function call terminates in a state where the Control is empty but the Dump is not. To return from the function, the machine restores the state (S, E, C, D) from the Dump, then pushes a on to the Stack. This is the following state transition:

$$\begin{array}{|c|}\hline a \\ \hline E' \\ \hline - \\ \hline (S, E, C, D) \\ \hline \end{array} \longmapsto \begin{array}{|c|}\hline a; S \\ \hline E \\ \hline C \\ \hline D \\ \hline \end{array}$$

The case of where the LHS D component is empty halts the machine (see earlier).

The Compiled SECD Machine

It takes 17 steps to evaluate $((\lambda x\, y.x + y)\, 3)\, 5$ using the SECD machine (see notes)! Much faster execution is obtained by first compiling the λ-term. Write $[\![M]\!]$ for the list of commands produced by compiling M; there are cases for each of the four kinds of λ-term.

Constants are compiled to the **const** command, which will (during later execution of the code) push a constant onto the Stack:

$$[\![k]\!] = \mathbf{const}(k)$$

Variables are compiled to the **var** command, which will push the variable's value, from the Environment, onto the Stack:

$$[\![x]\!] = \mathbf{var}(x)$$

The Compiled SECD Machine (2)

Abstractions are compiled to the **closure** command, which will push a closure onto the Stack. The closure will include the current Environment and will hold M as a list of commands, from compilation:

$$[\![\lambda x.M]\!] = \mathbf{closure}(x, [\![M]\!])$$

Applications are compiled to the **app** command at compile time. Under the interpreted SECD machine, this work occurred at run time:

$$[\![MN]\!] = [\![N]\!]; [\![M]\!]; \mathbf{app}$$

The Compiled SECD Machine (3)

We could add further instructions, say for *conditionals*. Let **test**(C_1, C_2) be replaced by C_1 or C_2, depending upon whether the value on top of the Stack is **true** or **false**:

$$[\![\,\textbf{if}\ E\ \textbf{then}\ M\ \textbf{else}\ N]\!] = [\![E]\!]; \textbf{test}([\![M]\!], [\![N]\!])$$

The Compiled SECD Machine (4)

To allow built-in 2-place functions such as $+$ and $\times$ could be done in several ways. Those functions could be made to operate upon ordered pairs, constructed using a **pair** instruction. More efficient is to introduce arithmetic instructions such as **add** and **mult**, which pop both their operands from the Stack. Now $((\lambda x\, y.x + y)\, 3)\, 5$ compiles to

$$\mathbf{const}(5); \mathbf{const}(3); \mathbf{closure}(x, C_0); \mathbf{app}; \mathbf{app}$$

and generates two further lists of commands:

$$\begin{aligned} C_0 &= \mathbf{closure}(y, C_1) \\ C_1 &= \mathbf{var}(y); \mathbf{var}(x);\ \mathbf{add} \end{aligned}$$

The Compiled SECD Machine (5)

Many further optimisations can be made, leading to an execution model quite close to conventional hardware (certainly close to JVM). Variable names could be removed from the Environment, and bound variables referred to by depth rather than by name. Special instructions **enter** and **exit** could efficiently handle functions that are called immediately (say, those created by the declaration $\mathsf{let}\ x = N\ \mathsf{in}\ M$), creating no closure:

$$[\![(\lambda x.M)N]\!] = [\![N]\!]; \mathbf{enter}; [\![M]\!]; \mathbf{exit}$$

The Compiled SECD Machine (6)

Tail recursive (sometimes called *iterative*) function calls could be compiled to the **tailapp** command, which would cause the following state transition:

$$
\begin{array}{|c|}
\hline
\mathbf{Clo}(x, C, E'); a \\
\hline
E \\
\hline
\textbf{tailapp} \\
\hline
D \\
\hline
\end{array}
\quad \longmapsto \quad
\begin{array}{|c|}
\hline
- \\
\hline
x = a; E' \\
\hline
C \\
\hline
D \\
\hline
\end{array}
$$

The useless state $(-, E, -, D)$ is never stored on the dump, and the function return after **tailapp** is never executed—the machine jumps directly to C!

Compare this to the Part IA treatment of iteration vs. recursion.

The SECD Machine—Recursion

The usual fixed point combinator, **Y** , fails under the SECD machine; it always loops. A modified fixed point combinator, including extra λ's to delay evaluation (just like if-then-else earlier!), does work:

$$\lambda f.(\lambda x.f(\lambda y.x\,x\,y)(\lambda y.x\,x\,y))$$

But it is hopelessly slow! Recursive functions are best implemented by creating a closure with a pointer back to itself.

The SECD Machine—Recursion (2)

Suppose that $f(x) = M$ is a recursive function definition. The value of f is represented by $\mathbf{Y}\,(\lambda f\,x.M)$. The SECD machine should interpret $\mathbf{Y}\,(\lambda f x.M)$ in a special manner, applying the closure for $\lambda f\,x.M$ to a dummy value, $\perp$. If the current Environment is E then this yields the closure

$$\mathbf{Clo}(x,\ M,\ f = \perp; E)$$

Then the machine modifies the closure, replacing the $\perp$ by a pointer looping back to the closure itself:

$$\begin{tikzcd} \mathbf{Clo}(x,\ M,\ f = \cdot\ ; E) \arrow[loop above] \end{tikzcd}$$

The SECD Machine—Recursion (3)

When the closure is applied, recursive calls to f in M will re-apply the same closure. The cyclic environment supports recursion efficiently.

The technique is called "tying the knot" and works only for function definitions. It does not work for recursive definitions of data structures, such as the infinite list $[0, 0, 0, \ldots]$, defined as $\mathbf{Y}\,(\lambda l.\,\mathbf{cons}\;0\,l)$. Therefore strict languages like ML allow only functions to be recursive.

Relation to Operational Semantics

A virtual machine (like the SECD machine) has operational semantic rules which are all *axioms*

$$\frac{}{(S, E, C, D) \longmapsto (S', E', C', D')}$$

Often semantics are simpler to understand if the traversing of expressions in the Control C (done explicitly in the SECD machine) are done implicitly using rules rather than axioms, e.g.

$$\frac{M \to M'}{MN \to M'N}$$

This enables us to drop the axioms (SECD reductions, such as that for $MN; C$) which explicitly manipulate the structure of C. You might prefer the lambda interpreter written in ML (next few slides) which uses the ML call stack to traverse the structure of the expression to be evaluated—it's a big-step semantics.

A lambda-interpreter in ML

The SECD-machine, while conceptually “merely an abstract machine” does a fair amount of administration for the same reason (e.g. it has explicitly to save its previous state when starting to evaluate a λ-application).

Syntax of the λ-calculus with constants in ML as

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

Values are of course either integers or functions (closures):

```
datatype Val = IntVal of int | FnVal of string * Expr * Env;
```

A lambda-interpreter in ML (2)

Environments are just a list of (name,value) pairs (these represent delayed substitutions—we never actually do the substitutions suggested by β-reduction, instead we wait until we finally use a substituted name and replace it with the λ-value which would have been substituted at that point);

```
    datatype Env = Empty | Defn of string * Val * Env;
```

and name lookup is natural:

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

A lambda-interpreter in ML (3)

The main code of the interpreter is as follows:

```
fun eval(Name(s), r) = lookup(s, r)
  | eval(Numb(n), r) = IntVal(n)
  | eval(Plus(e, e'), r) =
      let val v = eval(e,r);
          val v' = eval(e',r)
      in case (v,v') of (IntVal(i), IntVal(i')) => IntVal(i+i')
                      | (v, v') => raise oddity("plus of non-number") e
  | eval(Fn(s, e), r) = FnVal(s, e, r)
  | eval(Apply(e, e'), r) =
      case eval(e, r)
        of IntVal(i) => raise oddity("apply of non-function")
         | FnVal(bv, body, r_fromdef) =>
             let val arg = eval(e', r)
             in eval(body, Defn(bv, arg, r_fromdef)) end;
```

A lambda-interpreter in ML (4)

Note particularly the way in which dynamic typing is handled (`Plus` and `Apply` have to check the type of arguments and make appropriate results). Also note the two different environments (`r`, `r_fromdef`) being used when a function is being called.

A fuller version of this code (with test examples and with the “tying the knot” version of **Y** appears on the course web page.

Do we really do environment name-lookup?

Our environments (for both SECD and AST-walking interpreters) consisted of name-value pairs. This is surely slow—both in practice and gives $O(n)$ variable access instead of $O(1)$.

This overhead is pretty unavoidable if interpreting, but in the compiled SECD machine (or any other way of compiling the λ-calculus) we change the access instruction from variable x from $\mathbf{var}(x)$ to $\mathbf{var}'(i)$ where i is the (compile-time calculable) offset of x in E. This means that we no longer have to store name-value pairs in E, just values.

Do we really do environment name-lookup (2)

By doing a bit more work when making a closure:

- loading and saving the values of all its free variables as a vector within the closure—rather than just saving E itself there; and
- leaving the argument(s) on S (an implicit cons, rather than an explicit one)

we can ensure that all variable access is $O(1)$.

This is what ML does.

Lazy evaluation systems

Lecture 6

Lazy evaluation systems

Idea 1: change SECD machine for call-by-need (lazy evaluation): as follows. When a function is called, its argument is stored *unevaluated* in a closure containing the current environment. Thus, the call MN is treated something like $M(\lambda u.N)$, where u does not appear in N. This closure is called a *suspension* or a *thunk*. When a strict, built-in function is called, such as $+$, its argument is evaluated in the usual way.

It is essential that no argument be evaluated more than once, no matter how many times it appears in the function's body:

$$\begin{array}{l}\mathsf{let}\ sqr\, n = n \times n\ \mathsf{in} \\ sqr(sqr(sqr\, 2))\end{array}$$

Lazy evaluation systems (2)

If this expression were evaluated by repeatedly duplicating the argument of *sqr*, the waste would be intolerable. Therefore, the lazy SECD machine updates the environment with the value of the argument, after it is evaluated for the first time. But the cost of creating suspensions makes this machine ten times slower than the strict SECD machine, according to David Turner.

Idea 2: Turner (1976) showed how Combinatory Logic (a variable-free system equivalent to λ-calculus) could do things better:

`en.wikipedia.org/wiki/Combinatory_logic`

`en.wikipedia.org/wiki/Graph_reduction`

Combinators and their Evaluators

Combinators exist in a system called *combinatory logic* (CL). This name derives from historical reasons—we will not treat it as a logic.

Simplest version:

$$P ::= c \mid PP' \quad \text{with} \quad c ::= \mathbf{K} \mid \mathbf{S}$$

(essentially the λ-calculus with constants, but *without* variables or λ-abstractions!)

We will use P, Q and R to range over combinator terms. (Note that now $\mathbf{K}$ and $\mathbf{S}$ are constants within the language rather than the use of symbols of this font-style as abbreviations earlier in the notes.)

Combinators and their Evaluators (2)

While combinatory terms do not formally contain variables, it is convenient to allow them as an extension (for example during intermediate stages of the translation of λ-terms to combinatory terms, e.g. $\mathbf{K}\,x(\,\mathbf{S}\;\mathbf{K}\,x)(\,\mathbf{K}\;\mathbf{S}\;\mathbf{K}\,y)\,\mathbf{S}\,.$ Although CL is not particularly readable, it is powerful enough to encode the λ-calculus and hence all the computable functions!

This is what we to exploit!

Combinators and their Evaluators (3)

The combinators obey the following (δ-)reductions:

$$\begin{aligned} \mathbf{K}\,P\,Q &\to_w P \\ \mathbf{S}\,P\,Q\,R &\to_w P\,R(Q\,R) \end{aligned}$$

Thus, the combinators could have been defined in the λ-calculus by as the following abbreviations

$$\begin{aligned} \mathbf{K} &\equiv \lambda x\,y.x \\ \mathbf{S} &\equiv \lambda f\,g\,x.(f\,x)(g\,x) \end{aligned}$$

But note that $\mathbf{S}\ \mathbf{K}$ does not reduce—because $\mathbf{S}$ requires three arguments — while the corresponding λ-term does. For this reason, combinator reduction is known as *weak reduction* (hence the “w” in $\to_w$). [This concept is distinct from that of WHNF earlier.]

Combinators and their Evaluators (4)

Here is an example of weak reduction:

$$\mathbf{S}\ \mathbf{K}\ \mathbf{K}\,P \rightarrow_w \mathbf{K}\,P(\mathbf{K}\,P) \rightarrow_w P$$

Thus $\mathbf{S}\ \mathbf{K}\ \mathbf{K}\,P \twoheadrightarrow_w P$ for all combinator terms P; we can define the identity combinator by $\mathbf{I} \equiv \mathbf{S}\ \mathbf{K}\ \mathbf{K}$.

Equivalently, we could introduce it as a constant with δ-reduction

$$\mathbf{I}\,P \quad \rightarrow_w \quad P$$

Many of the concepts of the λ-calculus carry over to combinators. A combinator term P is in *normal form* if it admits no weak reductions. Combinators satisfy a version of the Church-Rosser Theorem: if $P = Q$ (by any number of reductions, forwards or backwards) then there exists a term Z such that $P \twoheadrightarrow_w Z$ and $Q \twoheadrightarrow_w Z$.

Combinators encode λ

Any λ-term may be transformed into a roughly equivalent combinatory term. The key is the transformation of a combinatory term P into another combinator term, written as $\lambda^* x.P$ since it behaves like a λ-abstraction even though it is a metalanguage concept—it translates one combinatory term into another rather than being part of any combinatory term.

The operation $\lambda^* x$, where x is a variable, is defined recursively as follows:

$$\begin{array}{lcll}
\lambda^* x.x & \equiv & \mathbf{I} & \\
\lambda^* x.P & \equiv & \mathbf{K}\,P & (x \text{ not free in } P) \\
\lambda^* x.P\,Q & \equiv & \mathbf{S}\,(\lambda^* x.P)(\lambda^* x.Q) &
\end{array}$$

Example translation

For example:

$$\begin{array}{rcl}
\lambda^* x\,y.y\,x & \equiv & \lambda^* x.(\lambda^* y.y\,x) \\
 & \equiv & \lambda^* x.\,\mathbf{S}\,(\lambda^* y.y)(\lambda^* y.x) \\
 & \equiv & \lambda^* x.(\,\mathbf{S}\;\mathbf{I}\,)(\,\mathbf{K}\,x) \\
 & \equiv & \mathbf{S}\,(\lambda^* x.\,\mathbf{S}\;\mathbf{I}\,)(\lambda^* x.\,\mathbf{K}\,x) \\
 & \equiv & \mathbf{S}\,(\,\mathbf{K}\,(\,\mathbf{S}\;\mathbf{I}\,))(\,\mathbf{S}\,(\lambda^* x.\,\mathbf{K}\,)(\lambda^* x.x)) \\
 & \equiv & \mathbf{S}\,(\,\mathbf{K}\,(\,\mathbf{S}\;\mathbf{I}\,))(\,\mathbf{S}\,(\,\mathbf{K}\;\mathbf{K}\,)\,\mathbf{I}\,)
\end{array}$$

Beware: each λ^* operation can *double* the size of its operand. Hence removing n nested λs with λ^* gives exponential blow-up. Turner showed how to do better (later).

Combinators encode λ-reduction

Using $(\lambda^* x.P)x \twoheadrightarrow_w P$, we may derive an analogue of β-reduction for combinatory logic. We also get a strong analogue of α-conversion—changes in the abstraction variable are absolutely insignificant, yielding identical terms:

For all combinatory terms P and Q,

$$\begin{array}{rcll} (\lambda^* x.P)Q & \twoheadrightarrow_w & P[Q/x] & \\ \lambda^* x.P & \equiv & \lambda^* y.P[y/x] & \quad \text{if } y \notin \mathrm{FV}(P) \end{array}$$

Formal translation

The mapping $(\ )_{CL}$ converts a λ-term into a combinator term. It simply applies λ^* recursively to all the abstractions in the λ-term; note that the innermost abstractions must be performed first (because λ^* is not defined on terms with a λ-abstraction in them!).

The inverse mapping, $(\ )_\lambda$, converts a combinator term into a λ-term. Note that the latter is pretty trivial, we merely treat each use of $\mathbf{S}$ or $\mathbf{K}$ as if it were an abbreviation.

Formal translation (2)

UNIVERSITY OF CAMBRIDGE

The mappings $(\ )_{CL}$ and $(\ )_{\lambda}$ are defined recursively as follows:

$$\begin{aligned}
(x)_{CL} &\equiv x \\
(M\,N)_{CL} &\equiv (M)_{CL}(N)_{CL} \\
(\lambda x.M)_{CL} &\equiv \lambda^{*}x.(M)_{CL}
\end{aligned}$$

$$\begin{aligned}
(x)_{\lambda} &\equiv x \\
(\,\mathbf{K}\,)_{\lambda} &\equiv \lambda x\,y.x \\
(\,\mathbf{S}\,)_{\lambda} &\equiv \lambda x\,y\,z.x\,z(y\,z) \\
(P\,Q)_{\lambda} &\equiv (P)_{\lambda}(Q)_{\lambda}
\end{aligned}$$

Formal translation—properties

Different versions of combinatory abstraction yield different versions of $(\ )_{CL}$; the present one causes exponential blow-up in term size, but it is easy to reason about. Note that the reverse translation $(\ )_\lambda$ is just linear.

Let us abbreviate $(M)_{CL}$ as M_{CL} and $(P)_\lambda$ as P_λ. It is easy to check that $(\ )_{CL}$ and $(\ )_\lambda$ do not add or delete free variables:

$$\mathrm{FV}(M) = \mathrm{FV}(M_{CL}) \qquad \mathrm{FV}(P) = \mathrm{FV}(P_\lambda)$$

Formal translation—properties (2)

But, normal forms and reductions are not preserved. For instance, **S K** is a normal form of combinatory logic; no weak reductions apply to it. But the corresponding λ-term is not in normal form:

$$(\,\mathbf{S}\ \mathbf{K}\,)_\lambda \equiv (\lambda x\, y\, z.x\, z(y\, z))(\lambda x\, y.x) \twoheadrightarrow \lambda y\, z.z$$

Hence equality is not obviously preserved.

Formal translation—properties (3)

But, if we add extensionality (in combinatory logic, extensionality takes the form of a new rule for proving equality):

$$\frac{Px = Qx}{P = Q} \qquad (x \text{ not free in } P \text{ or } Q)$$

In the λ-calculus, extensionality is expressed by by η-reduction:

$$\lambda x.Mx \rightarrow_\eta M \qquad (x \text{ not free in } M)$$

or it can be represented directly by a similar rule to that in combinatory logic above.

Formal translation—properties (4)

Assuming extensionality, the mappings preserve equality [Barendregt]:

$$\begin{array}{rcll} (M_{CL})_\lambda & = & M & \text{in the } \lambda\text{-calculus} \\ (P_\lambda)_{CL} & = & P & \text{in combinatory logic} \\ M = N & \iff & M_{CL} = N_{CL} & \\ P = Q & \iff & P_\lambda = Q_\lambda & \end{array}$$

Using Combinators for Compilation

Combinator abstraction gives us a theoretical basis for removing variables from λ-terms, and will allow efficient graph reduction.

But first, we need a better mapping from λ-terms to combinators. The improved version of combinatory abstraction relies on two new combinators, $\mathbf{B}$ and $\mathbf{C}$, to handle special cases of $\mathbf{S}$:

$$\begin{aligned} \mathbf{B}\,P\,Q\,R &\rightarrow_w P(Q\,R) \\ \mathbf{C}\,P\,Q\,R &\rightarrow_w P\,R\,Q \end{aligned}$$

Note that $\mathbf{B}\,P\,Q\,R$ yields the function composition of P and Q.

Using Combinators for Compilation (2)

Let us call the new abstraction mapping λ^T, after David Turner, its inventor:

$$\begin{array}{lcll}
\lambda^T x.x & \equiv & \mathbf{I} & \\
\lambda^T x.P & \equiv & \mathbf{K}\,P & (x \text{ not free in } P) \\
\lambda^T x.P\,x & \equiv & P & (x \text{ not free in } P) \\
\lambda^T x.P\,Q & \equiv & \mathbf{B}\,P(\lambda^T x.Q) & (x \text{ not free in } P) \\
\lambda^T x.P\,Q & \equiv & \mathbf{C}\,(\lambda^T x.P)Q & (x \text{ not free in } Q) \\
\lambda^T x.P\,Q & \equiv & \mathbf{S}\,(\lambda^T x.P)(\lambda^T x.Q) & (x \text{ free in } P \text{ and } Q)
\end{array}$$

Using Combinators for Compilation (3)

Although λ^T is a bit more complicated than λ^*, it generates much better code (i.e. combinators). The third case, for $P\,x$, takes advantage of extensionality; note its similarity to η-reduction. The next two cases abstract over $P\,Q$ according to whether or not the abstraction variable is actually free in P or Q. Let us do our example again:

$$\begin{aligned}
\lambda^T x\,y.y\,x &\equiv \lambda^T x.(\lambda^T y.y\,x) \\
&\equiv \lambda^T x.\,\mathbf{C}\,(\lambda^T y.y)x \\
&\equiv \lambda^T x.\,\mathbf{C}\ \mathbf{I}\,x \\
&\equiv \mathbf{C}\ \mathbf{I}
\end{aligned}$$

The size of the generated code has decreased by a factor of four!

Using Combinators for Compilation (4)

Unfortunately, λ^T can still cause a quadratic blowup in code size; additional primitive combinators can be introduced; all the constants of the functional language—numbers, arithmetic operators, etc.—must be taken as primitive combinators.

Having more and more primitive combinators makes the code smaller and faster. This leads to the method of *super combinators*, where the set of primitive combinators is extracted from the program itself.

For a good set of combinators the translation if a λ-term of n *symbols* only requires $n \log n$ combinators. While this might sound wasteful, there is a cheating involved. The $\log n$ derives from the fact that we require $\log n$ operations (from any finite fixed set of operators) to select from n different variables in scope. However, to have n different variables in scope, a λ-term of n symbols needs $n \log n$ *characters*. So the translation is linear after all in the true size of a program.

Lazy Evaluation using Combinators

UNIVERSITY OF CAMBRIDGE

Lecture 7

Combinator Terms as Graphs

Consider the ISWIM program

$$\mathsf{let}\ sqr(n) = n \times n\ \mathsf{in}\ sqr(5)$$

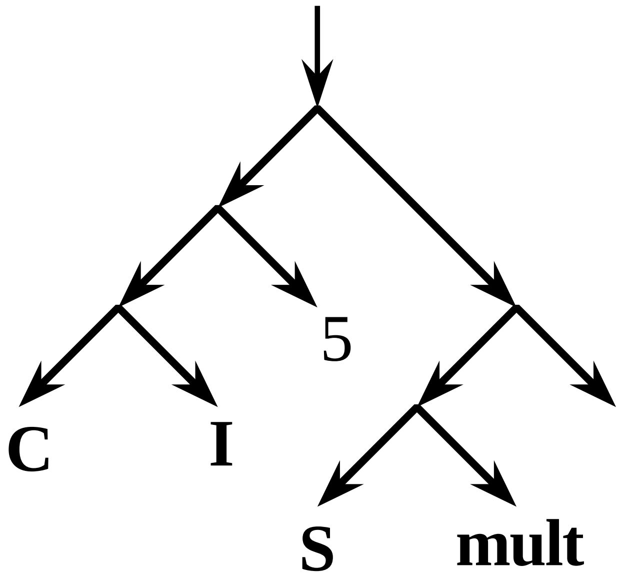
Let us translate it to combinators:

$$\begin{aligned}(\lambda^T f.f\,5)(\lambda^T n.\,\mathbf{mult}\,n\,n) &\equiv \mathbf{C}\,\mathbf{I}\,5\,(\,\mathbf{S}\,(\lambda^T n.\,\mathbf{mult}\,n)(\lambda^T n.n))\\ &\equiv \mathbf{C}\,\mathbf{I}\,5\,(\,\mathbf{S}\ \mathbf{mult}\ \mathbf{I}\,)\end{aligned}$$

We evaluate this by reducing it to normal form.

Combinator Terms as Graphs (2)

Graph reduction works on the combinator term's graph structure. This resembles a binary tree with branching at each application. The graph structure for **C I** 5 (**S mult I**) is as follows:



Repeated arguments cause sharing in the graph, ensuring that they are never evaluated more than once.

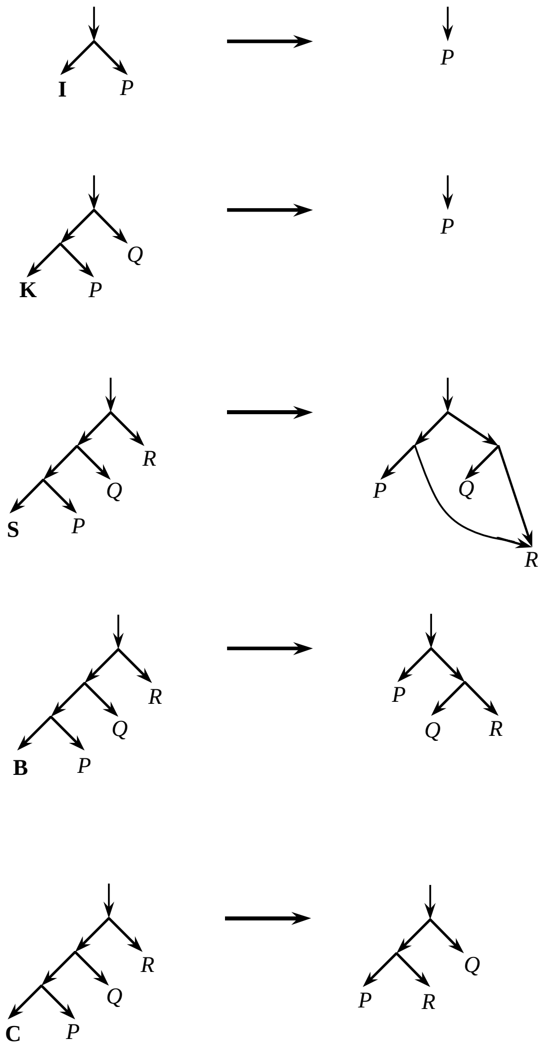
Combinator Terms as Graphs (2)

Graph reduction deals with terms that contain no variables. Each term, and its subterms, denote constant values. Therefore we may transform the graphs destructively—operands are never copied. The graph is *replaced* by its normal form!

Note that *replaced* means destructively updating the node pointed to (e.g. from an 'apply' node to a 'number' node or another 'apply' node). This type of replacement ('graph reduction') means that if several pointers point to one node then evaluating what one of them points to implies evaluating what they all point to.

See next slide. The sharing in the reduction for **S** is crucial, for it avoids duplicating R in the rule $\mathbf{S}\,P\,Q\,R \rightarrow_w P\,R(Q\,R)$.

Combinator Terms as Graphs (3)



Combinator Terms as Graphs (4)

We also require graph reduction rules for the built-in functions, such as **mult** . Because **mult** is a strict function, the graph for **mult** $P\,Q$ can only be reduced after P and Q have been reduced to numeric constants m and n. Then **mult** $m\,n$ is replaced by the constant whose value is $m \times n$. Graph reduction proceeds by walking down the graph's leftmost branch, seeking something to reduce. If the leftmost symbol is a combinator like **I** , **K** , **S** , **B** , or **C** , with the requisite number of operands, then it applies the corresponding transformation. If the leftmost symbol is a strict combinator like **mult** , then it recursively traverses the operands, attempting to reduce them to numbers.

Note that "leftmost-outermost" means the combinator found first on the leftmost spine of the graph.

Lazy evaluation systems (réprise)

Combinator graph rewriting systems are interesting, and were very promising for lazy evaluation systems. However, GHC (Glasgow Haskell Compiler)—the de facto standard nowadays—uses a more conventional approach.

Idea 3: values are still represented on the heap, but instead of an (interpreter-style) reduction system wandering over a graph structure, a lazy expression is represented as a structure containing code.

Branching to (i.e. an indirect subroutine call to) this code causes the associated expression to be evaluated and also to overwrite the node with the value of the expression (and the code pointer to be overwritten with a short subroutine which merely reloads the result).

Advantage: no time is spent examining graph nodes for what to do next.

Continuations

UNIVERSITY OF
CAMBRIDGE

Lecture 8

Continuations

A *continuation* is a value which represents the “rest of the program”; such values typically appear as function-like-things which never return. In many treatments (such as call/cc in Scheme (a form of Lisp)) they can seen as pretty much like exceptions.

For example, given ML

```
    let f b n g = 1 + (if b then n+10 else g(n+100))
```

if k represents the continuation “abandon the current computation, print the argument of k and return to top level” then `f true 0 k` would give 11, and `f false 0` (λ`x.x`) would give 101, but `f false 0 k` would give 100. (Formally we should be careful as to what ‘give’ means here.)

What’s this got to do with us?

Continuations (2)

In the previous slide, continuations were mixed with ordinary code to give behaviour outside our ideas of function call/return (admittedly useful to explain exceptions).

Let's us study how such non-standard control flow might be encoded directly in the λ-calculus.

The main thing is to show that (rather surprisingly) function return is rather redundant in λ-calculus. Indeed there is a Sussman and Steele paper "Lambda: the Ultimate Goto".

Then ideas of having multiple alternative continuations (rather than just monolithic "return") becomes more natural.,

Continuations (3)

How can function return be ‘inessential’?

Consider an ML function `f` : $A \to B$, and imagine an alternative top-level loop which just evaluates expressions an ignores their result.

Now `print(f(a))` does what ML normally does with `f(a)`, assuming `print` : $B \to$ `unit`

But what if we’re forbidden to use the result of `f`?

Define `g k x` with the *effect* of `k(f x)` and now write `g print a`.

Note that `f` : $A \to B$, but `g` : $(B \to$ `unit`$) \to (A \to$ `unit`$)$.

Say “`g` is the continuation passing form of `f`”.

Can we always define such a `g` with all return types being `unit`? [Yes].

Continuation Passing Style

A function which never returns (hence its return type may as well be `unit` were we to have types) but exits by calling another function (a continuation) which does not return, is said to be in *continuation passing style* (CPS).

The continuation it calls will typically be one of its parameter, but you also can see Prolog ‘backtracking points’ as being continuations stored away on a some form of stack. [Extended exercise.]

CPS transformation

Write $[\![\cdot]\!]$ for the following transformation (or translation or compilation) on the λ-calculus.

$$\begin{aligned}
[\![x]\!] &\equiv \lambda k.kx \\
[\![c]\!] &\equiv \lambda k.kc \\
[\![\lambda x.M]\!] &\equiv \lambda k.k(\lambda x.[\![M]\!]) \\
[\![M\ N]\!] &\equiv \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.(mn)k))
\end{aligned}$$

This is the *call-by-value* CPS transformation (there are variants for call-by-name) etc.

CPS transformation (2)

Note that the RHS can also be expressed using let using the standard sugaring $(\mathsf{let}\ x = e\ \mathsf{in}\ e') \equiv (\lambda x.e')e$:

$$[\![M\ N]\!] \quad \equiv \quad \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.(mn)k))$$

We can see continuation passing style also as "naming all intermediate results". Also, note that M and N are no longer quite as symmetric...

CPS encodes control

Note that the above translation says, given traditional ('direct-style') application MN, then do

$$\begin{aligned} [\![M\,N]\!] \;\equiv\; & \lambda k. \\ & [\![M]\!](\lambda m. \\ & [\![N]\!](\lambda n.(mn)k)) \end{aligned}$$

I.e. evaluate M before N. We could have equivalently written

$$\begin{aligned} [\![M\,N]\!] \;\equiv\; & \lambda k. \\ & [\![N]\!](\lambda n. \\ & [\![M]\!](\lambda m.(mn)k)) \end{aligned}$$

i.e. evaluate N first.

CPS encodes control (2)

Note that the CPS transformation produces a λ-expression which has essentially 'at most one reduction possible'—this then reduces to another λ-expression which has at most one reduction possible etc.

(This glib statement needs some tightening—the CPS translation introduces some spurious 'administrative redexes' which can be eliminated instead of being done at run-time; note also the programming language assumption that reductions are never done inside a λ-abstraction.)

This is what we mean by "CPS encodes control"—the $[\![\cdot]\!]$ translation hard-encodes an evaluation strategy into a λ-term.

CPS encodes control (3)

(Parenthetical slide)

We can take this further and give an alternative *call-by-name* CPS transformation which takes a λ-term and hard-encodes the call-by-name reduction strategy.

$$\begin{aligned}
[\![x]\!]_n &\equiv x \\
[\![c]\!]_n &\equiv \lambda k.kc \\
[\![\lambda x.M]\!]_n &\equiv \lambda k.k(\lambda x.[\![M]\!]_n) \\
[\![M\ N]\!]_n &\equiv \lambda k.[\![M]\!]_n(\lambda m.(m[\![N]\!]_n)k)
\end{aligned}$$

This is Plotkin's original translation; correction: it also has the "only one reduction possible" property. See [Danvy and Filinski 1992] for more on this.

Translating back?

Suppose $[\![M]\!] \equiv N$; then how do we get the value of M from N?

For such N the answer is simple: use $N(\lambda x.x)$. This acts as a 'do nothing' continuation. So writing $[\![N]\!]_{fromCPS} = N(\lambda x.x)$ suffices.

But what about the $M : A \rightarrow B$ and $N : (B \rightarrow \texttt{unit}) \rightarrow (A \rightarrow \texttt{unit})$ view? It turns out that $N : \forall\alpha.(B \rightarrow \alpha) \rightarrow (A \rightarrow \alpha)$ is a more general (and valid) type than the one above. Hence to apply N to $(\lambda x.x)$ we take $\alpha = B$ and get $N(\lambda x.x) : A \rightarrow B$ as expected.

Note: this use of $\lambda x.x$ only works for λ-terms produced by $[\![\cdot]\!]$—it does not necessarily work for continuations augmenting a conventional language (exceptions, or call/cc).

Didn't we cheat?

I said "only tail calls and all returns are **`unit`**" but wrote $(mn)k$ in

$$[\![M\,N]\!] \quad\equiv\quad \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.(mn)k))$$

and (mn) doesn't return **`unit`**.

One slick answer: all our translations get two curried arguments and use them both at once. Or...

Did we cheat (2)

Use pairs for encoding of arguments to functions (but not arguments to continuations):

$$
\begin{aligned}
[\![x]\!] &\equiv \lambda k.kx \\
[\![c]\!] &\equiv \lambda k.kc \\
[\![\lambda x.M]\!] &\equiv \lambda k.k(\lambda(k',x)([\![M]\!]k')) \\
[\![M\,N]\!] &\equiv \lambda k.[\![M]\!](\lambda m.[\![N]\!](\lambda n.m(k,n)))
\end{aligned}
$$

Note that this curries [modulo also swapping the argument order] $(B \to \texttt{unit}) \to (A \to \texttt{unit})$ into $((B \to \texttt{unit}) \times A) \to \texttt{unit}$. Look at this (typed assembly code) view: a function is just something you branch to which takes two arguments in registers. One is the real argument of type A and the other a 'return address' (type $B \to \texttt{unit}$) which is code to branch to which expects a B in a register.

CPS in practice

Suppose $fxy = g(h(x+1))(jy)$, then what we need to do is to capture the result of h, then the result of j and then pass the result of g to the continuation k. This results in

$$\begin{aligned} f'\ x\ y\ k = \quad & h'\ (x+1)\ (\lambda r_1. \\ & j'\ y\ (\lambda r_2. \\ & g'\ r_1\ r_2\ k)) \end{aligned}$$

Note how this parallels a machine-level implementation—first call h, then call j and finally call g. We have even had to decide whether to call h or j first—it might not matter from the programmer's point of view, but a machine implementation has to choose.

Side-effects and Monads

UNIVERSITY OF
CAMBRIDGE

Lecture 9

Side-effects in functional languages

Consider ML programs like:

```
val a = ref 1;
fun f(x,y) = (a := (!a)*2 + 1; x+y)
fun g(x) = (a := (!a)*3 + 1; x)
fun h(y) = (a := (!a)*5 + 1; y)
print f(g(1),h(2)) + !a;
```

The value printed depends critically on the order of evaluation of `f`, `g` and `h`. In ML the language specification says the order `g`, `h` then `f`.

The problem is the use of *implicit* side-effects. This isn't very *functional*—and what about I/O?

Side-effects—the ‘world’

UNIVERSITY OF CAMBRIDGE

How about (using `t`$\langle i \rangle$ for temporary variables):

```
type world = int;
fun f(x,y) w = (x+y, w*2 + 1)
fun g(x) w = (x, w*3 + 1)
fun h(y) w = (y, w*5 + 1)
print (let val w0 = 1;
           val (t1,w1) = g(1) w0
           val (t2,w2) = h(2) w1
           val (t3,w3) = f(t1,t2) w2
       in t3 + w3 end);
```

Note that each `world` is used exactly once (‘linearly’).

Side-effects—I/O

The side-effects (and interaction) of I/O can be encoded by returning a value of a datatype, which the read-eval-print loop serialises:

```
(* Note "unit->" trick is only needed for ML (because eager). *)
  datatype IO = TTwrite of char * (unit->IO)
              | TTread of (char -> IO)
              | Halt
  TTwrite('>', fn () =>
  TTread(fn c =>
  TTwrite(c+1, fn() =>
  Halt)));
```

Note the similarity to continuations. Note also that `IO` above is distinct from Haskell syntax (see later).

Side-effects and laziness

Note that the above encodings of side effects also work in lazy languages. You *may* be happy with ML's implicit treatment of side-effects cemented by its carefully specified order-of-evalation, but (from a programmer perspective) laziness gives fairly unpredictable order of evaluation and so *something else must be done*.

Before we dig into the Haskell solution, might we note that an overall-consistent view is that we can see most programming language features (concurrency being a notable exception) as encoded by λ-calculus:

Functions/procedures etc. are seen as taking two arguments: one is the true argument, one is the continuation: special continuations (like `TTread` above) encode "interact with the environment, possibly continuing with another part of the program afterwards".

Side-effects and laziness (2)

Big question: can we have a convenient “functions-as-functions” pure (and lazy) programming language in which side-effects code both neatly and efficiently?

Yes: Haskell is a proof-by-example.

Side-effects and Monads

Haskell encapsulates sequencing and I/O inside a *monad*. We will just use the IO monad (to avoid discussing Haskell 'typeclass').

Monads have two operations (we use the Haskell convention of using names like '`a`' and '`b`' for type variables, and '`::`' for type constraint)

```
return :: a -> IO a
>>=    :: IO a -> (a -> IO b) -> IO b
```

Note that `IO a` is the type of actions which when performed 'return' an `a` (but you need '`>>=`' to get at its value). It might help to see `IO a` is *approximately* the same as `a->IO` in ML earlier.

Note that

```
putChar :: Char -> IO ()      -- Haskell "()" is ML "unit"
getChar :: IO Char
```

Side-effects and Monads (2)

So we can write the 'prompting' example earlier as

```
putChar '>' >>= \() ->
getChar >>= \c ->
putChar (intToDigit((digitToInt c)+1))
```

Understanding how this works is important: values requesting actions return to top level where the evaluator acts on them and continues the computation. Each segment of computation between such top-level interaction is purely functional.

Why didn't we use `Halt` from earlier? Answer: Haskell `IO()` values are not simply constructors: both `putChar 'a'` and `return ()` have type `IO()`. Think of a value of `IO()` as an action which when performed does a sequence of zero or more IO operations before returning `()`.

Side-effects and Monads (3)

This can produces very readable code which has side-effects and does IO (using ghci, the interactive top-level loop):

```
Prelude> :type putChar 'a'  -- ask for the type of `putchar'
putChar 'a' :: IO ()

Prelude> let f(x) = if x then putChar '$' else return()

Prelude> f(True)             -- prints a '$' (see start of next line)
$Prelude> f(False)           -- does nothing
Prelude>
```

Side-effects and Monads (4)

This notation is still awkward, so Haskell provides '`do`' notation to sugar it little.

```
do { putChar '>';
     c <- getChar;
     putChar (intToDigit((digitToInt c)+1));
     return ()          --- optional
   }
```

This desugars to the previous slide.

Note that in both cases changing `return()` to `return 3` would make the expression return `IO Int` instead of `IO()`; we just tended not to use `return` in the 'prompting' example.

Side-effects and Monads (5)

The above introduced only the `IO` monad—this is interpreted specially by the Haskell top-level loop—and the builtin functions like `getChar` mean that `return` seems less useful.

`return` essentially performs the role of an identity to '`>>=`' the following three forms are equivalent (they all give a value of type `IO()` which causes 'a' to be printed):

```
putChar 'a'
return() >>= \()->putChar 'a'
putChar 'a' >>= \()->return()  -- [or putChar 'a' >>= return]
```

However, monads in general code up the idea of:

- `return` injects a value into a monad (representing a computation)
- `>>=` sequences two computations, giving back a single computation representing both

Side-effects and Monads (6)

Values can be 'extracted' (or at least passed on to the next computation) from a monad using the above '`>>=`$\lambda c.$' trick used in CPS.

The critical point about the `return` and `>>=` operators in a monad is that they encapsulate an object (e.g. state) and ensure it is threaded through the program like the `world` values were linearly threaded earlier.

The fact the the monad operators capture and manipulate a state in a linear manner explains their apparent initial complexity—interested readers should explore this more by writing some Haskell programs.

ML type system

Lecture 9.5

Type Systems for Functional Languages

UNIVERSITY OF CAMBRIDGE

The core ML type checker uses *unification*. For the λ-*calculus* we just

- give all λ-bound variables (unknown) type variables (note all variables are λ-bound as *programs* do not have free variables)
- give all constants their predeclared type
- propagate (equality) constraints imposed by application or use of constants.

Write `infer`(e, Γ, t) for $\Gamma \vdash e : t$ (see Part II course 'Types').

ML type checker in prolog

```
% Expressions:
% Expr ::= icon(int) | var(string) | lam(string, Expr)
%                                   | app(Expr, Expr)

% Type Expressions:
% Type ::= tint | tarrow(Type,Type)

% Type Environments:  list of (string,Type) pairs

% In 'lookup' the use of cut (!) ensures that we only find the most
% recent (i.e. non-scope-shadowed) version of a variable when the
% names of lambda-bound variables are not distinct.
lookup(X, [(X,T)|TEnv], T) :- !.
lookup(X, [(_,_)|TEnv], T) :-  lookup(X, TEnv, T).
```

ML type checker in prolog

```
infer(var(X), TEnv, T) :- lookup(X,TEnv,T).
infer(icon(_), TEnv, tint).
infer(lam(X,E), TEnv, arrow(T1,T2)) :- infer(E, [(X,T1)|TEnv], T2).
infer(app(F,E), TEnv, T2) :- infer(F, TEnv, arrow(T1,T2)),
                             infer(E, TEnv, T1).
% two tests:
typeofScombinator(T) :-
  infer(lam(f,lam(g,lam(x,
          app(app(var(f),var(x)), app(var(g),var(x)))))),
        [],
        T).

bug(T) :-
  infer(lam(x, app(var(x),var(x))), [], T).
```

But ML has let

The key invention of Milner (over Hindley's type checking encoded above in Prolog) is to handle let better.

Note the standard sugaring $(\mathsf{let}\ x = e\ \mathsf{in}\ e') \equiv (\lambda x.e')e$ does *not* respect ML type checking:

- $\mathsf{let}\ i = \lambda z.z\ \mathsf{in}\ i\,i$ type-checks in ML; and
- $(\lambda i.i\,i)(\lambda z.z)$ fails to type-check in ML

but they are equivalent semantically!

The idea is that let has special-case type-checking which gives a type more general than that of "apply-of-λ" (the former may have a type even when the latter fails to type-check as above).

Dealing with let

Variables bound by let are given *polymorphic* types; if the type of e is (say) $\alpha \rightarrow \mathsf{int}$ then the type of x in $\mathsf{let}\ x = e$ is treated as *generic*, i.e. $\forall\alpha.\alpha \rightarrow \mathsf{int}$.

Such *generic types* are instantiated by replacing all $\forall$-qualified type variables with a new (fresh) type variable at each use of x. (That's why the type-checking rules are more general than those those for λ which uses the same type variables for each use of x.)

The exact details of which type-variables in a let-bound variable are to be treated as generic is rather subtle—involving their use elsewhere (e.g. how any free variables of e is bound in an outer scope). The interesting corner case is the type of y in $\lambda x.\mathsf{let}\ y = (x, \lambda z.z)\ \mathsf{in}\ e'$.

See the "Types" Part II course.

Foundations of Functional Programming

UNIVERSITY OF CAMBRIDGE

The End

(Final version of these slides are now on the course website)