Functional Languages

According to Hudak, there are four features of functional languages:

1. Referential transparency (side-effect free)

   This is considered the *defining* feature by some.

   Because others do even consider it necessary (e.g., Scheme and ML are sometimes called functional), *purely* functional is used when you want to be sure the language has no side-effects.

2. Higher order functions

   Ability to passing functions as arguments and return as results.

   You may have higher order functsion without lambda – lambda allows for anonymous functions.

3. Lazy evaluation

   Using left-most outermost reduction semantics.

4. Data abstraction

Referential Transparency

Referential transparency is ability to substitute equals for equals.
Also, two occurrences of the same expressions in the same scope will always have the same value:

$$\begin{aligned}\texttt{let}\ x &= f\ a\ \texttt{in}\\ x + x &+ f\ a\end{aligned}$$

We know that x can be replaced by $f\ a$ and vice versa.
In compiler terminology, *common subexpression elimination* is possible without checking for dependencies.

The Untyped Lambda Calculus

Syntax (the same as the typed lambda calculus, except that types are omitted from the lambda expressions).

$$\begin{array}{ll} x \in Id & Identifiers \\ e \in Exp & Expressions \\ & \text{where } e ::= x \mid e_1\ e_2 \mid \lambda x.e \end{array}$$

Substitution $[e_1/x]e_2$ is defined as before.

Basic reduction rules, $alpha, \beta, \eta$, are also the same.

Some terms can be written in the untyped calculus that would be illegal in the typed. One example is the Y combinator.

Church-Rosser Theorems for the Untyped Lambda Calculus

Church-Rosser Theorem I: If $e_1 \twoheadleftarrow\!\!\!\twoheadrightarrow e_2$, then there exists an e_3 such that $e_1 \twoheadrightarrow e_3$ and $e_2 \twoheadrightarrow e_3$.

You can't take any wrong turns in the evaluation order.

The diagram from last time is uses another term, e_0.

In Diamond Lemma form: If $e_0 \twoheadrightarrow e_1$ and $e_0 \twoheadrightarrow e_2$, then there exists an e_3 such that $e_1 \twoheadrightarrow e_3$ and $e_2 \twoheadrightarrow e_3$.

Church-Rosser Theorem II: Restriction evaluations to normal order evaluation (left-most outermost), produces a normal form if anything does.

If $e_0 \twoheadleftarrow\!\!\!\twoheadrightarrow e_1$, then $e_0 \twoheadrightarrow^{left} e_1$.

Fixpoint Theorem for the Untyped Lambda Calculus

Theorem: Every Lambda expression e has a fixpoint e' such that

$$e\ e' \twoheadleftarrow\!\!\twoheadrightarrow\ e'$$

Proof: Let Y be the lambda term:

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

This is known as the Y combinator. Then take e' be $(Y\ e)$. To show this is a fixed point of e, evaluate it:

$$\begin{aligned}(Y\ e)\ &= (\lambda\ x\ .\ e\ (\ x\ x\ ))\ (\lambda\ x\ .\ e\ (\ x\ x\ ))\\ &= e\ ((\lambda\ x\ .\ e\ (\ x\ x\ ))\ (\lambda\ x\ .\ e\ (\ x\ x\ ))\\ &= e\ (Y\ e)\end{aligned}$$

Since $(Y\ e)$ is an input to e that produces itself, it is a fixpoint.

Church's Thesis

Effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus.

There is similar result for Turing machines, which makes the lambda calculus Turing-equivalent.

Much of recursive function theory was devised before the introduction of electronic, and long before high level programming languages.

Lisp Contributions (McCarthy)

1. Use of conditional in defining recursive functions.
2. Use of lists and operations like `mapcar`.
3. Idea of garbage collection.
4. The idea of program as data, i.e., S-expressions.

None of these consepts are supported in Fortran.

```
(define mapcar (fun lst)
     if (null lst)
         nill
        (cons (function (car lst))
              (mapcar fun (cdr list))))
```

Note: Lisp was not a pure functional language. It had things like `rplaca` and `rplacd` for modifying cons cells.

Church Representations

In the untyped lambda calculus, representations for all of the basic objects (natural numerals and booleans) and the operations on them (arithmetic and conditionals). May be defined in lambda terms.

$$\begin{aligned}
true &=^{def} \lambda\, x \,.\, \lambda\, y \,.\, x \\
false &=^{def} \lambda\, x \,.\, \lambda\, y \,.\, y \\
cond &=^{def} \lambda\, p \,.\, \lambda\, c \,.\, \lambda\, a \,.\, (p\ c\ a)
\end{aligned}$$

How do we know these work? Just have to prove they behave like booleans with respect to one another.

$$\begin{aligned}
(cond\ true\ M\ N) &= (\lambda\, p \,.\, \lambda\, c \,.\, \lambda\, a \,.\, (p\ c\ a))\ true\ M\ N \\
&= (true\ M\ N) \\
&= (\lambda\, x \,.\, \lambda\, y \,.\, x)\ M\ N \\
&= M
\end{aligned}$$

A similar proof can be done for $(cond\ false\ M\ N)$.

Church Representations

Natural numbers can also be expressed as untyped lambda terms.

$$
\begin{aligned}
0 &=^{def} \lambda\ f\ .\ \lambda\ a\ .\ a \\
1 &=^{def} \lambda\ f\ .\ \lambda\ a\ .\ f\ a \\
2 &=^{def} \lambda\ f\ .\ \lambda\ a\ .\ f\ f\ a \\
n &=^{def} \lambda\ f\ .\ \lambda\ a\ .\ f \ldots\ f\ a \quad \text{with } n \text{ applications of } f
\end{aligned}
$$

A successor function to increment by 1 is defined as:

$$succ \ =^{def} \lambda n.(\lambda f.\lambda a.f((nf)a))$$

Consider one instance of this:

$$
\begin{aligned}
(succ\ 0) &= (\lambda\ n\ .\ (\lambda\ f\ .\ \lambda\ a\ .\ f\ ((n\ f)\ a))) \\
&\quad\ \lambda\ f\ .\ \lambda\ a\ .\ a \\
&= (\lambda\ f\ .\ \lambda\ a\ .\ f\ (((\lambda\ f\ .\ \lambda\ a\ .\ a)\ f)\ a)) \\
&= (\lambda\ f\ .\ \lambda\ a\ .\ f\ ((\lambda\ a\ .\ a)\ a)) \\
&= (\lambda\ f\ .\ \lambda\ a\ .\ f\ a)
\end{aligned}
$$

Addition is defined as:

$$plus \ = \lambda\ m\ .\ \lambda\ n\ .\ n\ succ\ m$$

Evaluation Order

Strict, eager, or applicative evaluation order (sometimes called "call by value") is when arguments are completely evaluated before an the beta rule for application is applied.

Non-strict, lazy, or normal evaluation order (sometimes called "call by name") is when the beta rule is applied as soon as a lambda expression is available, before evaluating the arguments.

In a pure functional language, the difference is in termination. Non-strict may terminate when strict does not.

A language can be strict, non-strict, etc., or different constructs may have different evaluation orders. E.g., conditionals use non-strict evaluation on branches.

Lisp, `if` and `cond` are considered *special forms* whereas everything else is a function.

Structured Operational Semantics for PCF Conditional

Consider the given non-strict rules for conditionals:

$$(cond) \quad \texttt{if}\ true\ then\ M\ \texttt{else}\ N \ \rightarrow\ M$$
$$\texttt{if}\ false\ then\ M\ \texttt{else}\ N \ \rightarrow\ N$$

So `if` is strict in the boolean expression, but not in the two arms.

Here are (rather silly) rules that for the strict evaluation. Errors and non-termination in either of the branches would affect the entire program:

$$(cond) \quad \texttt{if}\ true\ then\ M\ \texttt{else}\ N \ \rightarrow\ M$$
where M and N are in normal form.
$$\texttt{if}\ false\ then\ M\ \texttt{else}\ N \ \rightarrow\ N$$
where M and N are in normal form.

Function Application Evaluation Order

Mitchell distinguishes between several evaluation orders:

- non-deterministic (any redex)
- left-most (left-most outermost redex)
- lazy (left-most outermost, but doesn't simplify subterms at all)
  Still Church-Rosser on complete programs, but not expressions
- eager (left-most innermost)

Lazy Evaluation

Here are some of the rule for lazy evaluation:

$$\frac{M \to N}{M \to^{lazy} N} M \to N \text{ is a reduction axiom}$$

The first rule means that we can reduce any expression of the form $(\lambda x.M)N$, since the beta rule exists.
We can also reduce an application (MN) by reducing the M, due to the following rule:

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

There are similar rules to evalate the boolean expression in a considitional as well as basic expressions.

Eager Evaluation

Define the idea of a *value* as a constant, variable, lambda abstraction, or pair of values. A value will be denoted V.

Functions

(β) $\qquad (\lambda x : \sigma . M)V \rightarrow^{eager} [V/x]M$

$$\frac{M \rightarrow^{eager} M'}{MN \rightarrow^{eager} M'N}$$

$$\frac{N \rightarrow^{eager} N'}{VN \rightarrow^{eager} VN'}$$

SOS for Expressions

$$\frac{E_1 \Rightarrow E_1'}{(+\ E_1\ E_2) \Rightarrow (+\ E_1'\ E_2)} \qquad \frac{E_2 \Rightarrow E_2'}{(+\ E_1\ E_2) \Rightarrow (+\ E_1\ E_2')}$$

$$(+\ N_1\ N_2) \Rightarrow N_3, \quad N_i \text{ are numerals and } N_3 = N_1 + N_2$$