Part II: Lambda Calculus

- Lambda Calculus is a foundation for functional programs.
- It's an operational semantics, based on term rewriting.
- Lambda Calculus was developed by Alonzo Church in the 1930's and 40's as a theory of computable functions.
- Lambda calculus is as powerful as Turing machines. That is, every Turing machine can be expressed as a function in the calculus and vice versa
- Church Hypothesis: Every computable algorithm can be expressed by a function in Lambda calculus.

Pure Lambda Calculus

- Pure Lambda calculus expresses only functions and function applications.
- Three term forms:

Names	$x, y, z \in \mathcal{N}$			
Terms	D, E, F	$::=$	x	names
		$\mid$	$\lambda x.E$	abstractions
		$\mid$	$D\,E$	applications

- Function-application is left-associative.
- The scope of a name extends as far to the right as possible.
- Example: $\lambda f.\lambda x.f\ E\ x \quad \equiv \quad (\lambda f.(\lambda x.((f\ E)\ x)))$.
- Often, one uses the term *variable* instead of *name*.

Evaluation of Lambda Terms

Evaluation of lambda terms is by the β-reduction rule.

$$\beta : \qquad (\lambda x.D)E \quad \rightarrow \quad [E/x]\ D$$

$[E/x]$ is substitution, which will be explained in detail later.

Example:

$$
\begin{array}{lcl}
(\lambda x.x)(\lambda y.y) & \rightarrow & \lambda y.y \\
(\lambda f.\lambda x.f\ (f\ x))(\lambda y.y)z & \rightarrow & (\lambda x.(\lambda y.y)(\lambda y.y)x)z \\
 & \rightarrow & (\lambda y.y)((\lambda y.y)z) \\
 & \rightarrow & (\lambda y.y)z \\
 & \rightarrow & z
\end{array}
$$

Term Equivalence

Question: Are these terms equivalent?

$$\lambda x.x \qquad \text{and} \qquad \lambda y.y$$

What about

$$\lambda x.y \qquad \text{and} \qquad \lambda x.z$$

?

Need to distinguish between *bound* and *free* names.

Free And Bound Names

Definition The free names $\mathrm{fn}(E)$ of a term E are those names which occur in E at a position where they are not in the scope of a definition in the same term. Formally, $\mathrm{fn}(E)$ is defined as follows.

$$
\begin{aligned}
\mathrm{fn}(x) &= \{x\} \\
\mathrm{fn}(\lambda x.E) &= \mathrm{fn}(E)\backslash\{x\} \\
\mathrm{fn}(F\,E) &= \mathrm{fn}(F) \cup \mathrm{fn}(E).
\end{aligned}
$$

All names which occur in a term E and which are not free in E are called *bound*.

A term without any free variables is called *closed*.

Renaming

- The spelling of bound names is not significant.
- We regard terms D and E which are convertible by renaming of bound names as equivalent, and write $D \equiv E$
- This is expressed formally by the following α-renaming rule:

$$\alpha : \qquad \lambda x.E \quad \equiv \quad \lambda y.[y/x]E \qquad (y \notin \mathrm{fn}(E))$$

Theorem: $\equiv$ is an equivalence relation.

Substitutions

- We now have the means to define substitution formally:

$$
\begin{array}{lcll}
[D/x]\ x & = & D & \\
[D/x]\ y & = & y & (x \neq y) \\
[D/x]\ \lambda x.E & = & \lambda x.E & \\
[D/x]\ \lambda y.E & = & \lambda y.[D/x]E & (x \neq y, y \notin \mathrm{fn}(D)) \\
[D/x]\ (F\ E) & = & ([D/x]F)\ ([D/x]E) &
\end{array}
$$

- Substitution affects only the free names of a term, not the bound ones.

Avoiding Name Capture

- We have to be careful that we do not bind free names of a substituted expression (this is called *name capture*).
- For instance,

$$[y/x]\lambda y.x \quad \not\equiv \quad \lambda y.y \qquad \text{!!!}$$

- We have to α-rename $\lambda y.x$ first before applying the substitution:

$$\begin{aligned} [y/x]\lambda y.x &\equiv [y/x]\lambda z.x && \text{by } \alpha \\ &\equiv \lambda z.y \end{aligned}$$

- In the following, we will always assume that terms are renamed automatically so as to make all substitutions well-defined.

Normal Forms

Definition: We write $\twoheadrightarrow$ for reduction in an arbitrary number of steps. Formally:

$$E \twoheadrightarrow E' \qquad \text{iff} \qquad \exists n \geq 0. E \equiv E_0 \rightarrow \ldots \rightarrow E_n \equiv E'$$

Definition: A *normal form* is a term which cannot be reduced further.

Exercise: Define:

$$\begin{aligned} S &\stackrel{\text{def}}{\equiv} \lambda f.\lambda g.\lambda x.fx(gx) \\ K &\stackrel{\text{def}}{\equiv} \lambda x.\lambda y.x \end{aligned}$$

Can SKK be reduced to a normal form?

Combinators

- Lambda calculus gives one the possibility to define new functions using λ abstractions.
- **Question:** Is that really necessary for expressiveness, or could one also do with a fixed set of functions?
- **Answer:** (by Haskell Curry) Every closed λ-definable function can be expressed as some combination of the *combinators* S and K.
- This insight has influenced the implementation of one functional language (Miranda).
- The Miranda compiler translates a source program to a combination of a handful of combinators (S, K, and a few others for "optimizations").
- A Miranda runtime system then only has to implement the handful of combinators.
- Very elegant, but "slow as continental drift".

Confluence

If a term had more than one normal form, we'd have to worry about an implementation finding "the right one".

The following important theorem shows that this case cannot arise.

Theorem: (Church-Rosser) Reduction in λ-calculus is *confluent*: If $E \twoheadrightarrow E_1$ and $E \twoheadrightarrow E_2$, then there exists a term E_3 such that $E_1 \twoheadrightarrow E_3$ and $E_2 \twoheadrightarrow E_3$.

Proof: Not easy.

Corollary: Every term can be reduced to at most one normal form.

Proof: Your turn.

Terms Without Normal Forms

- There are terms which do not have a normal form.
- Example: Let

$$\Omega \stackrel{\text{def}}{\equiv} (\lambda x.(xx))(\lambda x.(xx))$$

Then

$$\begin{aligned} \Omega &\rightarrow (\lambda x.(xx))(\lambda x.(xx)) \\ &\rightarrow (\lambda x.(xx))(\lambda x.(xx)) \\ &\rightarrow \ldots \end{aligned}$$

- Terms which cannot be reduced to a normal form are called *divergent*.

Evaluation Strategies

The existence of terms without normal forms raises the question of *evaluation strategies*.

For instance, let $I \stackrel{\text{def}}{\equiv} \lambda x.x$ and consider:

$$\begin{array}{ll} & (\lambda x.I)\ \Omega \\ \rightarrow & I \end{array}$$

in a single step. But one could also reduce:

$$\begin{array}{ll} & (\lambda x.I)\ \Omega \\ \rightarrow & (\lambda x.I)\ \Omega \\ \rightarrow & (\lambda x.I)\ \Omega \\ \rightarrow & \ldots \end{array}$$

by always doing the $\Omega \rightarrow \Omega$ reduction.

Complete Evaluation Strategies

An evaluation strategy is a decision procedure which tells us which rewrite step to choose, given a term where several reductions are possible.

Question 1: Is there a *complete* evaluation strategy, in the following sense:

> Whenever a term has a normal form, the reduction using the strategy will end in that normal form.

?

Weak Head Normal Forms

In practice, we are not so much interested in normal forms; only in terms which are not further reducible "at the top level".

That is, reduction would stop at a term of the form $\lambda x.E$ even if E was still reducible.

These terms are called *weak head normal forms* or *values*. They are characterized by the following grammar.

$$\text{Values} \quad V \quad ::= \quad x \mid \lambda x.E$$

We now reformulate our question as follows:

Question 2: Is there a (weakly) complete evaluation strategy, in the following sense:

> Whenever a term can be reduced to a value, the reduction using the strategy will end in that value.

Precise Definition of Evaluation Strategy

How can we define evaluation strategies formally?

Idea: Use *reduction contexts.*

Definition: A *context* C is a term where exactly one subterm is replaced by a "hole", written $[\,]$. $C[E]$ denotes the term which results if the hole of context C is filled with term E.

Examples of contexts: $[\,]$ $\quad \lambda x.\lambda y.[\,]$ $\quad \lambda x.f\ [\,]$

Previously, we have admitted reduction anywhere in a term without explicitly saying so. Let's formalize this:

Definition: A term E *reduces at top-level* to a term E', if E and E' are the left- and right-hand sides of an instance of rule β. We write in this case: $E \rightarrow_\beta E'$.

Definition: A term E *reduces* to a term E', written $E \to E'$ if there exists a context C and terms D, D' such that

$$
\begin{array}{lcl}
E & \equiv & C[D] \\
E' & \equiv & C[D'] \\
D & \to_\beta & D'
\end{array}
$$

So much for general reduction.

Now, to define an evaluation strategy, we *restrict* the possible set of contexts in the definition of $\to$.

The restriction can be expressed by giving a *grammar* which describes permissible contexts.

Such contexts are called *reduction contexts* and we let the letter R range over them

Call-By-Name

Definition: The *call-by-name* strategy is given by the following grammar for reduction-contexts:

$$R \quad ::= \quad [\,] \;\mid\; R\,E$$

Definition: A term E *reduces* to a term E' using the call-by-name strategy, written $E \rightarrow_{\text{cbn}} E'$ if there exists a reduction context R and terms D, D' such that

$$\begin{array}{lcl} E & \equiv & R[D] \\ E' & \equiv & R[D'] \\ D & \rightarrow_\beta & D' \end{array}$$

Deterministic Reduction Strategies

Definition: A reduction strategy is *deterministic* if for any term at most one reduction step is possible.

Proposition: The call-by-name strategy $\rightarrow_{\text{cbn}}$ is deterministic.

Proof: There is only one way a term can be split into a reduction context R and a subterm which is reducible at top-level.

Exercise: Reduce the term $K\ I\ \Omega$ with the call-by-name strategy, where

$$\begin{array}{lcl} K & \stackrel{\text{def}}{\equiv} & \lambda x.\lambda y.x \\ I & \stackrel{\text{def}}{\equiv} & \lambda x.x \\ \Omega & \stackrel{\text{def}}{\equiv} & (\lambda x.(xx))(\lambda x.(xx)) \end{array}$$

Theorem: (Standardization) Call-by-name reduction is weakly complete: Whenever $E \twoheadrightarrow V$ then $E \twoheadrightarrow_{\text{cbn}} V'$.

Proof: hard.

Question: Modify call-by-name reduction to *normal-order reduction*, which always reduces a term to a normal form, if it has one. Which changes to the definition of reduction contexts R are necessary?

- In practice, call-by-name is rarely used since it leads to duplicate evaluations of arguments. Example:

$$
\begin{array}{ll}
 & (\lambda f.f(fy))((\lambda x.x)(\lambda x.x)) \\
\to & (\lambda x.x)(\lambda x.x)((\lambda x.x)(\lambda x.x)y) \\
\to & (\lambda x.x)((\lambda x.x)(\lambda x.x)y) \\
\to & (\lambda x.x)((\lambda x.x)y) \\
\to & (\lambda x.x)y \\
\to & y
\end{array}
$$

- Note that the argument $(\lambda x.x)(\lambda x.x)$ is evaluated twice.

- A shorter reduction can often be achieved by evaluating function arguments before they are passed. In our example:

$$
\begin{array}{ll}
 & (\lambda f.f(fy))((\lambda x.x)(\lambda x.x)) \\
\rightarrow & (\lambda f.f(fy))(\lambda x.x) \\
\rightarrow & (\lambda x.x)((\lambda x.x)y) \\
\rightarrow & (\lambda x.x)y \\
\rightarrow & y
\end{array}
$$

Call-By-Value

The *call-by-value* strategy evaluates function arguments before applying the function.

It is often more efficient than the call-by-name strategy. However:

Proposition: The call-by-value strategy is not (weakly) complete.

Question: Name a term which can be reduced to a value following the call-by-name strategy, but not following the call-by-value strategy.

Hence we have a dilemma: One strategy is in practice too inefficient, the other is incomplete.

How to solve this?

First Solution: Call-By-Need Evaluation

- **Idea:** Rather than re-evaluating arguments repeatedly, save the result of the first evaluation and use that for subsequent evaluations.
- This technique is called *memoization*.
- It is used in implementations of *lazy* functional languages such as Miranda or Haskell.
- A formalization of call-by-need is possible, but beyond the scope of this course. See

  A Call-by-Need Lambda Calculus, Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky and Philip Wadler. *Proc. ACM Symposium on Principles of Programming Languages*, 1995. http://diwww.epfl.ch/~odersky/papers/#FP–Theory.

Exercise: What is a good data representation for call-by-need evaluation?

Second Solution: Call-By-Value Calculus

- Rather than tweaking the evaluation strategy to be complete with respect to a given calculus, we can also change the calculus so that a given evaluation strategy becomes complete with respect to it.
- This has been done by Gordon Plotkin, in the *call-by-value* lambda calculus.
- The *terms* and *values* of this calculus are defined as before. A more concise re-formulation is:

Terms	D, E, F	$::=$	$V \mid D\ E$
Values	V, W	$::=$	$x \mid \lambda x.E$

- As reduction rule, we have:

$$\beta_{\mathbf{V}}: \qquad (\lambda x.D)V \quad \rightarrow \quad [V/x]\ D$$

- As reduction contexts, we have:

$$R_V \quad ::= \quad [\,] \mid R_V\ E \mid V\ R_V$$

- Let $\rightarrow_V$ be general reduction of terms with the β_V rule, and let $\rightarrow_{\text{cbv}}$ be β_V reduction only at the holes of call-by-value reduction contexts R_V. Then we have:

Theorem: (Plotkin) $\rightarrow_V$ reduction is confluent.

Theorem: (Plotkin) $\rightarrow_{\text{cbv}}$ is weakly complete with respect to $\rightarrow_V$.

Church Encodings

- The treatment so far covered *pure* lambda calculus which consists of just functions and their applications.
- Actual programming languages add to this primitive data types and their operations, named value and function definitions, and much more.
- We can model these constructs by extending the basic calculus.
- But it is also possible to *encode* these constructs in the basic calculus itself.
- These encodings will be presented in the following.
- We will assume in general call-by-name evaluation, but will also work out modifications needed for call-by-value.

Encoding of Booleans

- An abstract type of booleans is given by the two constants true and false as well as the conditional **if**.
- Other constructs can be written in terms of these primitives. E.g.

  not x $=$ **if** (x) false **else** true
  x || y $=$ **if** (x) true **else** y
  x && y $=$ **if** (x) y **else** false

- **Idea**: The encoding of a boolean value $\mathsf{B} \in \{\mathsf{true},\mathsf{false}\}$ is the binary function

  λx.λy. **if** (B) x **else** y

- That is:

  true $\stackrel{\text{def}}{\equiv}$ λx. λy. x
  false $\stackrel{\text{def}}{\equiv}$ λx. λy. y
  if c x y $\stackrel{\text{def}}{\equiv}$ c x y

Example:

$$\begin{array}{lcl} \textbf{if}\ (\mathsf{true})\ \mathsf{D}\ \textbf{else}\ \mathsf{E} & \stackrel{\text{def}}{\equiv} & \mathsf{true\ D\ E} \\ & \stackrel{\text{def}}{\equiv} & (\lambda \mathsf{x}\ .\lambda \mathsf{y}.\ \mathsf{x})\ \mathsf{D\ E} \\ & \rightarrow & (\lambda \mathsf{y}\ .\ \mathsf{D})\ \mathsf{E} \\ & \rightarrow & \mathsf{D} \end{array}$$

Question: What changes to this encoding are necessary if the evaluation strategy is call-by-value?

Encoding of Lists

The encoding of Booleans can be generalized to arbitrary algebraic data types.

Example: Consider the type of lists (as defined in Haskell):

```
data List a = Nil | Cons a (List a)
```

This defines a type of lists with (nullary) constructor Nil and (curried binary) constructor Cons.

A list xs can be accessed using a case-expression

```
case xs of
   Nil        ⇒ E1
|  Cons x xs  ⇒ E2
```

Here, the expression of the second branch, E_2, can refer to the variables x and xs defined in the Cons pattern.

All other functions over lists can be written in terms of the case-expression.

For instance, function car which equals head except that it avoids errors, can be written as:

```
car xs =
    case xs of
        Nil        ⇒ Nil
    |   Cons y ys  ⇒ x
```

Question: How can lists be encoded?

Same principle as before: Equate a list with the case-expression that accesses it.

$$\mathsf{xs} \stackrel{\text{def}}{\equiv} \lambda\mathsf{a}.\lambda\mathsf{b}.\textbf{case}\ \mathsf{xs}\ \mathsf{of}\ \mathsf{Nil} \Rightarrow \mathsf{a}\ \mid \mathsf{Cons}\ \mathsf{x}\ \mathsf{xs} \Rightarrow \mathsf{b}\ \mathsf{x}\ \mathsf{xs}$$

That is:

$$\mathsf{Nil} \stackrel{\text{def}}{\equiv} \lambda \mathsf{a}.\lambda \mathsf{b}.\ \mathsf{a}$$
$$\mathsf{Cons\ x\ xs} \stackrel{\text{def}}{\equiv} \lambda \mathsf{a}.\lambda \mathsf{b}.\ \mathsf{b\ x\ xs}$$

or, equivalently:

$$\mathsf{Cons} \stackrel{\text{def}}{\equiv} \lambda \mathsf{x}.\lambda \mathsf{xs}.\lambda \mathsf{a}.\lambda \mathsf{b}.\ \mathsf{b\ x\ xs}$$

The pattern-bound names x and xs are now passed as parameters to the case branch that accesses them.

Example: : car is coded as follows:

$$\mathsf{car} \stackrel{\text{def}}{\equiv} \lambda \mathsf{xs}.\ \mathsf{xs\ Nil}\ (\lambda \mathsf{y}.\lambda \mathsf{ys}.\mathsf{y})$$

Exercise: Church-encode function isEmpty which returns true iff the given list is empty.

Encoding of Numbers

The encoding for lists generalizes to arbitrary data types which are defined in terms of a finite number of constructors.

For instance, whole numbers don't present any new difficulties. To see this, note that natural numbers can be coded as algebraic data types as follows:

$$\mathsf{data\ Nat} \;=\; \mathsf{Zero} \mid \mathsf{Succ\ Nat}$$

Hence:

$$\mathsf{Zero} \stackrel{\text{def}}{\equiv} \lambda \mathsf{a}.\lambda \mathsf{b}.\mathsf{a}$$
$$\mathsf{Succ\ x} \stackrel{\text{def}}{\equiv} \lambda \mathsf{a}.\lambda \mathsf{b}.\mathsf{b\ x}$$

Note: Church encodings do not reflect types. In fact Zero, Nil, and true are all mapped to the same term!

Encoding of Definitions

A non-recursive value definition **val** x = D ; E can be encoded as:

$$\textbf{val}\ \mathsf{x} = \mathsf{D}\ ;\ \mathsf{E} \quad \stackrel{\text{def}}{\equiv} \quad (\lambda \mathsf{x}.\mathsf{E})\ \mathsf{D}$$

Caveat: With a call-by-name strategy, D might be evaluated more than once.

Let's try an analogous principle for function definitions:

$$\begin{aligned} \textbf{def}\ \mathsf{f}\ \mathsf{x} = \mathsf{D}\ ;\ \mathsf{E} \quad &\stackrel{\text{def}}{\equiv} \quad \textbf{val}\ \mathsf{f} = \lambda \mathsf{x}.\mathsf{D}\ ;\ \mathsf{E} \\ &\stackrel{\text{def}}{\equiv} \quad (\lambda \mathsf{f}.\mathsf{E})\ (\lambda \mathsf{x}.\mathsf{D}) \end{aligned}$$

But this fails if f is used recursively in D! (Why?)

Fixed Points to the Rescue

If we have a recursive definition of

```
val f = E
```

where E refers to f, we can interpret this as a solution to the equation

$$f \quad = \quad E$$

Another way to characterize solutions to this equation is to say that these solutions are fixed points of the function $\lambda f.E$.

Definition: A *fixed point* of a function f is a value x such that

$$f\,x \quad = \quad x$$

Proposition: The solutions of $f = E$ are exactly the fixed points of $\lambda f.E$

Proof: F is a solution of the equation

$$f \quad = \quad E$$

iff

$$F \quad = \quad [F/f]E$$

iff

$$F \quad = \quad (\lambda f.E)\ F$$

iff F is a fixed point of $\lambda f.E$.

Fixed Point Operators

Let's assume the existence of a *fixed point operator* Y. For every function f, $Y f$ evaluates to a fixed point of f. That is,

$$Y f \quad = \quad f\,(Y f)$$

Then we can encode potentially recursive definitions as follows:

$$\begin{array}{lcl} \textbf{def}\ \mathsf{f\ x = D\ ;\ E} & \stackrel{\text{def}}{\equiv} & \textbf{val}\ \mathsf{f = Y\ (\lambda f.\lambda x.D)\ ;\ E} \\ & \stackrel{\text{def}}{\equiv} & \mathsf{(\lambda f.E)\ (Y\ (\lambda f.\lambda x.D))} \end{array}$$

Remains the question whether Y exists.

Proposition: Let

$$Y \stackrel{\text{def}}{\equiv} \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

Then Y is a fixed point operator:

$$Y f = f\,(Y f)$$

Proof: By repeated β-reduction.

Least Fixed Points

In fact, an equation will in general have several solutions, and a function will in general have several fixed points.

Example: The equation

$$f \quad = \quad f$$

has every λ-term as a solution.

Can we characterize the fixed point computed by Y?

Proposition: Among all the fixed points of a function f, $Y f$ will return the one which diverges most often. This is also called the *least fixed point* of the function f.

Exercise: Find the least fixed point of $\lambda f.f$ (which is also the least solution of the equation $f \ = \ f$).

Connection to Domain Theory

- The definition of least fixed points is made precise in the field of *domain theory*.
- Domain theory gives λ-terms meaning by mapping them to mathematical functions.
- Divergent terms are modeled by a value $\perp$, which stands for "undefined".
- Domain theory introduces a partial ordering on values which makes $\perp$ smaller than any defined value.
- The fixed points computed by Y are the smallest with respect to this ordering.

Summary

- We have seen the basic theory of λ-calculus, and how it can express functional programming.
- Two main variants: Call-by-value and call-by-name.
- In each case, evaluation is described by reduction of function applications, using rule β (or β_V).
- λ-calculus has two important properties, which make it well suited as a basis of deterministic programming languages:
  - **Confluence:** Every term can be reduced to at most one value.
  - **Standardization:** There exists a deterministic reduction strategy which always reduces a term to a value, provided it can be done at all.

Outlook

- λ-calculus is ideally suited as a basis for functional programming.
- But it is less well suited as basis for imperative programming with side effects (essentially, need to introduce and carry along a data structure describing global state).
- It is not suitable at all as a basis for reactive systems with concurrent evaluation.
- Later on, we will extend λ-calculus to *join calculus* which can express these additional concepts.
- The price we will have to pay for the generalization is the loss of the confluence and standardization properties.