# G6021 Comparative Programming

Part 3 - Foundations

# The Lambda Calculus

- A computational model based on the notion of a function.
- Defined by Alonzo Church in the 1930's, as a precise notation for anonymous functions.

The $\lambda$-calculus is used to:

- study computability (as an alternative to Turing Machines),
- define models (denotational semantics) of programming languages,
- study strategies and implementation techniques for functional languages (abstract machines),
- encode proofs in a variety of logics,
- design automatic theorem provers and proof assistants.

# $\lambda$-calculus: Syntax

Definition:

*Assume an infinite set $\mathcal{X}$ of variables denoted by $x$, $y$, $z$, ..., then the set of $\lambda$-terms is the least set satisfying:*

$$M ::= \mathcal{X} \mid (\lambda \mathcal{X}.M) \mid (MM)$$

*which are called variable, abstraction and application.*

Some examples

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

An intuition. The following functions are all the same:

- $f\ x\ y = x + y$
- $f\ x = \lambda y.x + y$
- $f = \lambda x.\lambda y.x + y$

## $\lambda$-calculus: Conventions

- write as few parentheses as possible:

$$(\lambda y.(xy)) = \lambda y.xy$$

- application associates to the *left*:

$$xyz = ((xy)z)$$

- abstractions bind as far as possible to the *right*

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

- abstractions can be abbreviated:

$$\lambda x.\lambda y.M = \lambda xy.M$$

# Examples of $\lambda$-terms

- $x$, $\lambda x.x$, $xy$, $\lambda x.z$, $xz(yz)$, $\lambda x.\lambda y.yx$
- $\lambda xy.x$, $\lambda xyz.z(xy)$, $\lambda xyz.xz(yz)$
- $\lambda x.\lambda y.x$, $\lambda x.\lambda y.y$
- $(\lambda x.x)y$, $(\lambda x.\lambda y.xy)(\lambda x.x)$
- $\lambda f.\lambda x.x$, $\lambda f.\lambda x.fx$, $\lambda f.\lambda x.f(fx)$, $\lambda f.\lambda x.f(f(fx))$
- $\lambda x.xx$

Note: Haskell syntax:

`\x -> M` is the same as $\lambda x.M$

Exercise: Write the above examples in Haskell syntax. Are they all valid in Haskell?

# Variables

A variable is *free* in a $\lambda$-term if it is not bound by a $\lambda$.

More precisely, the set of free variables of a term is defined as:

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.M) & = & FV(M) - \{x\} \\
FV(MN) & = & FV(M) \cup FV(N)
\end{array}
$$

Terms without free variables are called *closed terms*.

We can define:

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

Question: What is *BV* defining?

# $\alpha$-conversion

$\lambda$-terms that differ only in the names of their bound variables will be equated. More precisely: If $y$ is not free in $M$:

$$\lambda x.M =_\alpha \lambda y.M\{x \mapsto y\}$$

where $M\{x \mapsto y\}$ is the term $M$ where each occurrence of $x$ is replaced by $y$ (i.e. we rename every free occurrence of $x$ to $y$).

IMPORTANT:

- $\lambda$-terms are defined modulo $\alpha$-conversion, so $\lambda x.x$ and $\lambda y.y$ are the SAME term.
- $\alpha$-equivalent terms represent the same computation (see below).

# Computation

- Abstractions represent functions, which can be applied to arguments.
- The main computation rule is $\beta$*-reduction*, which indicates how to find the result of the function for a given argument.
- A *redex* is a term of the form: $(\lambda x.M)N$
- It reduces to the term $M\{x \mapsto N\}$ where $M\{x \mapsto N\}$ is the term obtained when we substitute $x$ by $N$ taking into account bound variables.

$\beta$-reduction:

$$(\lambda x.M)N \to_\beta M\{x \mapsto N\}$$

- Note that we use the word "reduce", but this does not mean that the term on the right is any simpler. Why?
- Notation: if $M \to_\beta M_1 \to_\beta M_2 \cdots M_n$ then we write $M \to_\beta^* M_n$

# Substitution

Substitution is a special kind of replacement: $M\{x \mapsto N\}$ means replace all *free* occurrences of $x$ in $M$ by the term $N$.

Question: Why only the free occurrences? What happens if we replace all occurrences?

A very useful property of substitution is the following, known as the Substitution Lemma:

If $x \notin FV(P)$:

$$(M\{x \mapsto N\})\{y \mapsto P\} = (M\{y \mapsto P\})\{x \mapsto N\{y \mapsto P\}\}$$

## Examples (conversion, reduction, substitution)

$\alpha$-conversion:

- $\lambda x.x =_\alpha \lambda y.y$
- $\lambda x.\lambda y.xy =_\alpha \lambda z_1.\lambda z_2.z_1 z_2$

and $\beta$-reduction:

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

## Normal forms

When do we stop reducing?

- Normal form (NF): Stop reducing when there are no redexes left to reduce.
- A *normal form* is a term that does not contain any redex.
- A term that can be reduced to a term in normal form is said to be *normalisable*.
  Example:
  $$(\lambda x.a(\lambda y.xy))\ b\ c \to_\beta a(\lambda y.by)c$$
  which is a normal form (application associates to the left).
- Weak Head Normal Form (WHNF). Stop reducing when there are no redexes left, but without reducing under an abstraction.

# Exercises

1. What is the difference between a term having a normal form, and being a normal form? Write down some example terms.
2. If a closed term is a weak head normal form, it has to be an abstraction $\lambda x.M$. Why?
3. Does the term $(\lambda x.xx)(\lambda x.xx)$ have a normal form?

# Reduction graphs

The $\beta$-reduction graph of a term $M$, written $G_\beta(M)$, is the set:

$$\{N \mid M \rightarrow_\beta^* N\}$$

directed by $\rightarrow_\beta$. If several redexes give rise to $M_0 \rightarrow_\beta M_1$, then that many directed arcs connect $M_0$ to $M_1$.

Example

$G_\beta(WWW)$ with $W \equiv \lambda xy.xyy$ is:

$$\begin{tikzcd}
WWW \arrow[d, shift left] & (\lambda y.yyy)W \arrow[l] \\
(\lambda y.Wyy)W \arrow[u, shift left] \arrow[r, shift left] & (\lambda y.(\lambda z.yzz)y)W \arrow[l, shift left] \arrow[u]
\end{tikzcd}$$

# Reduction graph examples

Exercise: Draw the reduction graph for $(II)(II)$, where $I = \lambda x.x$.

$$
\begin{tikzcd}
& (II)(II) \arrow[dl] \arrow[dr] & \\
I(II) \arrow[dr, "*"] & & III \arrow[dl] \\
& II \arrow[d] & \\
& I &
\end{tikzcd}
$$

Why is one arrow marked "$*$"?

# Reduction graph examples

$G_\beta((\lambda x.xx)(\lambda x.xx))$ is:

$$\begin{tikzcd} (\lambda x.xx)(\lambda x.xx) \arrow[loop below] \end{tikzcd}$$

Exercise: Draw $G_\beta((\lambda x.x(II))(II))$, where $I = \lambda x.x$

## Properties of Computations

- Confluence: If $M \to_\beta^* M_1$ and $M \to_\beta^* M_2$ then there exists a term $M_3$ such that $M_1 \to_\beta^* M_3$ and $M_2 \to_\beta^* M_3$
- Normalisation: there exists a sequence of reductions which terminates
- Strong Normalisation (or Termination): All reduction sequences terminate

Note:

- The $\lambda$-calculus is confluent but not normalising (or strongly normalising).
- Confluence implies unicity of normal forms: Each $\lambda$-term has at most one normal form.

Exercise: Find a term that is not strongly normalising (i.e., a term that does not terminate).

# Strategies for reduction

- There can be many different ways in which a term can be reduced to a normal form (resp. WHNF).
- The choice that we make can make a huge difference in how many reduction steps are needed.
- The leftmost-outermost strategy finds the normal form, if there is one. But it may be inefficient.

Exercise:

Indicate whether the following $\lambda$-terms have a normal form:

- $(\lambda x.(\lambda y.yx)z)v$
- $(\lambda x.xxy)(\lambda x.xxy)$

## Remark

Most functional programming languages adopt the following ideas:

1. reduce to WHNF (do not reduce under an abstraction). Exercise: Why?
2. evaluate arguments in a specific way

The difference between many functional languages lies in the choice taken for the second point.

# Evaluating Arguments

1. Call-by-Value (Applicative order of reduction):
   evaluate arguments first so that we substitute the reduced terms (avoid duplication of work).
2. Call-by-name (Normal order of reduction):
   evaluate arguments each time they are needed.
3. Lazy Evaluation: evaluate arguments at most once.

Question:

Which is the best/worst strategy? Give examples to support your claims.

# Arithmetic in the $\lambda$-calculus: Church Numerals

We can define the natural numbers as follows:

- $\overline{0} = \lambda x.\lambda y.y$
- $\overline{1} = \lambda x.\lambda y.x\ y$
- $\overline{2} = \lambda x.\lambda y.x(x\ y)$
- $\overline{3} = \lambda x.\lambda y.x(x(x\ y))$
- $\ldots$

Using this representation we can define arithmetic functions.

Example, $\overline{n} \mapsto \overline{n+1}$, is defined by the $\lambda$-term $S$:

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

To check it:

- $S\overline{n} = (\lambda x.\lambda y.\lambda z.y((x\ y)z))(\lambda x.\lambda y.x \ldots (x(x\ y))$
- $\rightarrow_\beta \lambda y.\lambda z.y((\lambda x.\lambda y.x \ldots (x(x\ y))\ y)z)$
- $\rightarrow_\beta^* \lambda y.\lambda z.y(y \ldots (y(y\ z)) = \overline{n+1}$

In general, to define an arithmetic function

$$f : Nat^k \mapsto Nat$$

we will use a $\lambda$-term $\lambda x_1 \ldots x_k.M$, which will be applied to $k$ numbers: $(\lambda x_1 \ldots x_k.M)\overline{n_1} \ldots \overline{n_k}$

For example, the following term defines addition:

$$ADD = \lambda x.\lambda y.\lambda a.\lambda b.(x\ a)(y\ a\ b)$$

Exercise:

Check that this term applied to two numbers computes their sum. Hint: reduce the term $(\lambda x.\lambda y.\lambda a.\lambda b.(x\ a)(y\ a\ b))\overline{n}\ \overline{m}$

Exercises:

1. Show that the $\lambda$-term MULT = $\lambda x.\lambda y.\lambda z.x(yz)$ applied to two Church numerals $m$ and $n$ computes $m \times n$.
2. What does the term $\lambda n.\lambda m.m\ (MULT\ \ n)\ \overline{1}$ compute?

## Booleans

We can represent Boolean values:

- $\mathit{False} = \lambda x.\lambda y.y$
- $\mathit{True} = \lambda x.\lambda y.x$

and Boolean functions. The function NOT is defined by

$$\mathit{NOT} = \lambda x.(x\ \mathit{False})\ \mathit{True}$$

We can check that this definition is correct:

- $\mathit{NOT}\ \mathit{False} = (\lambda x.(x\ \mathit{False})\ \mathit{True})\mathit{False}$
- $\rightarrow_\beta (\mathit{False}\ \mathit{False})\mathit{True} \rightarrow^*_\beta \mathit{True}$

and

- $\mathit{NOT}\ \mathit{True} = (\lambda x.(x\ \mathit{False})\ \mathit{True})\mathit{True}$
- $\rightarrow_\beta (\mathit{True}\ \mathit{False})\mathit{True} \rightarrow^*_\beta \mathit{False}$

## Conditionals

The following term implements an if-then-else:

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

Note that

$$IF\ B\ E_1\ E_2 \rightarrow_\beta^* E_1 \text{ if } B = True$$

and

$$IF\ B\ E_1\ E_2 \rightarrow_\beta^* E_2 \text{ if } B = False$$

Example:

The function *is-zero?* can be defined as: $\lambda n.n(True\ False)True$.

Then *is-zero?* $\overline{0} \rightarrow_\beta^* True$

and

*is-zero?* $\overline{n} \rightarrow_\beta^* False$ if $n > 0$.

We can use *IF* and *is-zero?* to define the *SIGN* function:

$$SIGN\ n = IF\ (is\text{-}zero?\ n)\ \ \overline{0}\ \ \overline{1}$$

Exercise: Write SIGN as a $\lambda$-term.

Check that the following definitions are correct:

$AND = \lambda x.\lambda y.(x\ y)x$
$OR = \lambda x.\lambda y.(x\ x)y$

# The cost of computing

- We have seen that different reduction strategies can change the efficiency of the computation (also termination)
- We can transform algorithms into more efficient versions. We look at one way here:

  *Continuation Passing Style*

  Note: tail recursive, or accumulating parameter style.
- Program transformation is a very rich topic. Many open research topics here...

# Continuations

- Continuations were originally introduced in the study of semantics of programming languages: to allow the formal definition of *control* structures.
  - Jumps in programs are considered a “bad” thing (difficult to reason about)
  - Many constructs allow controlled jumps (conditional, loops, case, etc)
  - Do not allow jumping into the middle of a block or function body

  Continuations allow some of these features to be captured in a “clean” way:
  - Exceptions to leave a block or function early
  - `callcc` allows a point in the program to be “marked”. `throw` returns to that point to continue the evaluation.
- They are an advanced control construct available in some functional languages (notably Standard ML and Scheme).

## Continuation Passing Style (CPS)

- The idea of CPS is that every function takes an extra argument: a continuation.
- A continuation is a function which consumes the result of a function, and produces the final answer.
- Thus, a continuation represents the remainder of the current computation.

The simplest way to understand CPS is to think about evaluating a simple functional application:

# Example CPS: Factorial

```
fact n = if n==0 then 1 else n*fact(n-1)
fact 4
```

Consider now the CPS form:

```
factcps n k = if n==0 then k 1
              else factcps (n-1) (\r -> k (n*r))
factcps 4 (\x -> x)
```

The second argument `k` is the continuation.

Exercise:

1. What is the relationship between:

   `k (fact n)` and `factcps n k`

2. What is one *main* difference between `fact` and `factcps`?

# Factorial: evaluation

```
fact 4 = if 4==0 then 1 else 4*fact(4-1)
       = 4*fact(3)
       = 4*(if 3==0 then 1 else 3*fact(3-1))
       = 4*(3*fact(2))
  ...  = 4*(3*(2*(1*1)))

factcps 4 (\x -> x)
  = factcps 3 (\r -> (\x -> x) (4*r))
  = factcps 3 (\r -> (4*r))
  = factcps 2 (\r -> (\r -> (4*r)) (3*r))
  = factcps 2 (\r -> (4*(3*r)))
  = factcps 1 (\r -> (4*(3*(2*r))))
  = factcps 0 (\r -> (4*(3*(2*(1*r)))))
  = (\r -> (4*(3*(2*(1*r)))))1
  = (4*(3*2*(1*1)))
```

## Tail Recursion

- It is generally well-understood in compiler technology that *tail* recursive programs can be implemented more efficiently (because they can be transformed into a simple loop).
- A well known example: Compare the following two functions:

```
rev [] = []
rev (h:t) = rev t ++ [h]

revacc [] acc = acc
revacc (h:t) acc = revacc t (h:acc)
```

- Nothing remains to be done *after* the recursive call (recall the definition of `++`).
- Formally, we can show that `rev l = revacc l []`

# Continued...

```
rev [] = []
rev (h:t) = rev t ++ [h]

revcps [] k = k []
revcps (h:t) k = revcps t (\r -> k(r++[h]))
```

Exercise: Verify that `rev l = revcps l (\x -> x)`

Note that all the continuations here can be represented by lists: `\x -> x++l` for some list `l`. Thus `revcps` can be simplified to `revacc`.

# CPS

- The previous examples are part of a rich theory of *program transformation*.
- Many advanced compilers perform this transformation automatically (when possible).
- In addition to eliminating recursion, these transformations add additional control in the form of strategies.
- On a negative note, programs become higher-order, and we might loose termination properties.

# Worked Example: factorial again

```
fact n = if n==0 then 1 else n*fact(n-1)
fact 4
```

Consider now the CPS form:

```
factcps n k = if n==0 then k 1
              else factcps (n-1) (\r -> k (n*r))
factcps 4 (\x -> x)
```

We can simplify the continuation:

```
factacc n acc = if n==0 then acc
                else factacc (n-1) (n*acc)
factacc 4 1
```

## Other uses of CPS

Many programming languages have features like:

- goto (in pascal like languages)
- longjmp/setjmp in C
- exceptions in Java, Haskell, SML, etc.

which allow for the change of control of a program (to exit the current block).

- Continuations are a way of expressing these issues
- Achieved by passing a stack as a value to functions: this stack allows the state of the computation to be reinstated at any point—we can move to any past state in a safe way.
- Such stacks are known as reified control stacks.

However, this is beyond the scope of this course...

## Summary of CPS

- All functions can be written in CPS style.
- Some continuations have nice representations as accumulating parameters.
- Tail recursive functions can be compiled into a loop: more efficient than a recursion.
- Many other program transformation techniques for functional programming

## Summary

- The $\lambda$-calculus is the foundation of functional programming (and also the foundation of many programming concepts).
- It is possible to program using only the $\lambda$-calculus, but easier if we allow data types (pattern matching, richer syntax, etc.)
- Test out examples in the notes, and do exercises.
- Try writing some of the $\lambda$-terms in Haskell
- Can you write a data type in Haskell for representing $\lambda$-terms?