COMP0020 Functional Programming

Lecture 2

The Lambda Calculus :

A Simple Introduction

Contents

- Low-level target language and computational model for functional languages
- Very simple syntax
- Simple rules for evaluation
- Order of applying the rules
- Terminology : "bound" and "free"

- High-level functional languages may be translated by a compiler into the lambda calculus (though there are other implementation routes) ; the λ-calculus might then then translated to an even simpler run-time representation.
- The λ-calculus is very simple — few operators and few rules

- The λ-calculus views functions as rules for generating an answer given a certain input.
- Although the λ-calculus was initially conceived as being sequential, there are many non-sequential implementations (e.g. much work was done in the 1980s to use functional languages - based on the λ-calculus - for parallel processing).

- A program is an expression (like an arithmetic expression) rather than a sequence of instructions

- All a program does is to return a value
  - There are no "side effects" — the only purpose of the program is to return a value, and the only purpose of each part of the program is to return a value

  - In a programming language based on the λ-calculus, the value returned by the program might be an instruction to the operating system (e.g. to write to a file, or to print to the screen)

Untyped (or “Type-Free”) Lambda Calculus Syntax

$$
\begin{array}{lcl}
\text{program} & ::= & \text{expression} \\
\\
\text{expression} & ::= & x \\
 & | & \text{expression} \quad \text{expression} \\
 & | & \lambda x \;.\; \text{expression}
\end{array}
$$

- **Variable :** the name x indicates a variable name — it can be any name
- **Application :** when one expression follows another, the former is normally taken to be a function and the latter is taken to be the argument, thus $expression_1$ $expression_2$ indicates the function $expression_1$ applied to the argument $expression_2$.
- **Abstraction :** the lambda abstraction $\lambda x.expression$ indicates a function of one argument x and with function body *expression*. The name x can be used inside *expression* and represents the value to which the function is applied. We will assume that it is permissible for x to **not** appear inside *expression* (there are different versions of the λ-calculus : some permit this, and some do not).

The type-free λ-calculus can compute anything that is computable. However, the minimal syntax is cumbersome. For example, the numbers 0 and 1 are represented as functions : $\lambda f.\ \lambda x\ .\ x$ $\lambda f\ .\ \lambda x\ .\ f\ x$

The syntax is therefore often extended with :

- Constant values such as 3
- Operators such as $+, \times$
  - Initially, all operators are *prefix* (the operator appears to the left of its argument(s))
- Extra brackets for grouping (such as (x))
- Types (such as char, bool)
  - But we will not cover the typed lambda calculus[1]
- Lambda abstractions with more than one argument : these can already be accommodated as nested abstractions (e.g. $\lambda x.\lambda y.expression$ or $\lambda x.(\lambda y.expression)$) but the syntax can be extended to permit the equivalent $\lambda x_1 x_2.expression$ or in general $\lambda x_1 \ldots x_n.expression$

1. Note that whilst the untyped λ-calculus is Turing-equivalent, the typed λ-calculus typically is not (it depends on the properties of the type system)

Untyped Lambda Calculus — extended syntax

program	::	expression
expression	::	x
	|	constant
	|	operator
	|	expression expression
	|	λx . expression
	|	(*expression*)

Lambda calculus functions

- Functions do NOT have names !
  - Functions can be defined but must be used immediately
- Function arguments DO have names
  - that can only be used inside the function body (zero or more times)
- Functions can be arguments to other functions (they are *higher order*)
  - that way they can have names when they are passed as arguments to other functions
  - and can be used zero or more times inside the other function
  - and it is also possible for a function to return a function as its result

Defining and applying a Lambda calculus function

- To define the (anonymous) function taking one argument (the argument is called x) which adds 1 to x and returns the sum as its result :

$$\lambda x.((+x)1)$$

- Often simplified to one of the following :

$\lambda x.(+x\ 1)$ [*because function application associates to the left*]

$\lambda x.(x+1)$ [*but we must extend the syntax to permit infix operators*]

- To apply the previously defined function to the constant number 3 :

$$(\lambda x.(x+1))\ 3$$

Untyped Lambda Calculus — extended syntax with infix operators

program :: expression

expression :: x
| constant
| operator
| expression expression
| expression operator expression
| λx . expression
| (*expression*)

Rules for evaluation

- α-reduction

$$\lambda x.E \;\rightarrow\; \lambda y.E[y/x]$$

- β-reduction

$$(\lambda x.E)\ z \;\rightarrow\; E[z/x]$$

- η-reduction

$$\lambda x.(E\ x) \;\rightarrow\; E \qquad (\textit{if } x \textit{ is not free in } E)$$

- δ-rules — there is a separate δ-rule for each operator (such as $+, \times$) ; e.g. the δ-rule for $+$ says that $3 + 4$ evaluates to 7
- NB : $E[y/x]$ means "for each *free* occurrence of x in E replace that x with y". This becomes important if E contains another function definition that re-uses the name x for its argument. The embedded function definition *binds* the name x to a new value, thus the enclosing expression E sees all occurrences of x inside the embedded function definition as being *bound* (i.e. not *free*).

Terminology

- Binding — a BINDING links a name to a value. This happens whenever a function is applied.
- Bound and Not Bound (a.k.a. "Free"). In the following expression :

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

we say that x is BOUND and y is NOT BOUND (alternatively, we say that y is FREE). This is because we know what value x refers to - it is the argument to *this* function, but the value of y is unknown (presumably this expression occurs inside the function body of a function that binds y, but we don't know how deeply nested we might be inside function definitions inside other function definitions)

Order of evaluation ("reduction order")

- Normal Order Reduction
  - "leftmost-outermost" first
  - guaranteed to terminate (if termination is possible)

- Other possible reduction orders
  - applicative order reduction
  - parallel reduction

All evaluation strategies are guaranteed to give the same result for an expression (caveat termination). That unique result is called the **Normal Form** of the expression.

Lambda Calculus Examples

Example 1 :

$(5 + 3) \quad \rightarrow$ *by δ rule for* $+$
8

Example 2 :

$(\lambda x.(x + 3))\ 5 \quad \rightarrow$ *by β reduction*
$(5 + 3) \quad \rightarrow$ *by δ rule for* $+$
8

Example 3 :

$(\lambda y.((\lambda x.(x + y))\ 5))\ 3 \quad \rightarrow$ *by β reduction*
$(\lambda x.(x + 3))\ 5 \quad \rightarrow$ *by β reduction*
$(5 + 3) \quad \rightarrow$ *by δ rule for* $+$
8

Lambda Calculus Examples

Example 4 :

$$
\begin{array}{ll}
(\lambda x.((\lambda x.(x+3))\ x))\ 5 & \rightarrow \mathit{by}\,\alpha\ \mathit{reduction} \\
(\lambda y.((\lambda x.(x+3))\ y))\ 5 & \rightarrow \mathit{by}\,\beta\ \mathit{reduction} \\
(\lambda x.(x+3))\ 5 & \rightarrow \mathit{by}\,\beta\ \mathit{reduction} \\
(5+3) & \rightarrow \mathit{by}\ \delta\ \mathit{rule\ for}\ + \\
8 &
\end{array}
$$

Example 5 :

$$
\begin{array}{ll}
(\lambda x.(x\ 5))\quad (\lambda x.(x+3)) & \rightarrow \mathit{by}\,\beta\ \mathit{reduction} \\
((\lambda x.(x+3))\ 5) & \rightarrow \mathit{by}\,\beta\ \mathit{reduction} \\
(5+3) & \rightarrow \mathit{by}\ \delta\ \mathit{rule\ for}\ + \\
8 &
\end{array}
$$

Lambda Calculus Examples

Example 6 :

$$
\begin{array}{ll}
\lambda x.((x\ 5) + (x\ 4))\ \ (\lambda x.(x+3)) & \rightarrow \textit{by}\ \beta\ \textit{reduction} \\
((\lambda x.(x+3))\ 5) + ((\lambda x.(x+3))\ 4) & \rightarrow \textit{by}\ \beta\ \textit{reduction} \\
(5+3) + ((\lambda x.(x+3))\ 4) & \rightarrow \textit{by}\ \beta\ \textit{reduction} \\
(5+3) + (4+3) & \rightarrow \textit{by}\ \delta\ \textit{rule for} + \\
8 + (4+3) & \rightarrow \textit{by}\ \delta\ \textit{rule for} + \\
8 + 7 & \rightarrow \textit{by}\ \delta\ \textit{rule for} + \\
15 &
\end{array}
$$

Summary

- Low-level target language and computational model for functional languages
- Very simple syntax
- Only four rules for evaluation
- Apply the rules in any order (caveat termination)
- "Normal Order" guaranteed to terminate (if termination is possible)
- Terminology : "bound" and "free"

END OF LECTURE