Handout 3: Pure and Applied Lambda Calculi

The lambda calculus defined by Church in 1930’s and given by the syntax

\[ M ::= x \mid \lambda x. M' \mid M_1 M_2 \]

is termed the “Pure Lambda Calculus.” The pure lambda calculus deals with nothing but functions. A language obtained by adding “constants” for primitive values (like booleans, integers and other data types) and operations on them is called an “Applied Lambda Calculus”. Church proved that the pure lambda calculus is Turing-complete in itself. So, integers and other data types can be encoded within the lambda calculus. However, from a programming language point of view, there is value in treating primitive data types separately because we have hardware support for them.

Pure Lambda Calculus

1. Church encodings. Church’s encodings of data types in lambda calculus proceed in an “object-oriented” style. For each data type, we think of one or more key methods that are needed to make use of the values of that data type. We can then encode the value data as simply collections of such methods.

2. Boolean values. The key method for boolean values is the “if” method, using which we can select one of two conditional branches:

\[ \text{if} : \text{bool} \to A \to A \to A \]

The data values true and false should then satisfy:

\[ \text{if true } x y = x \]
\[ \text{if false } x y = y \]

The trick now is to define true and false as the corresponding “if” methods. The “if” method of true should select the then-branch and the “if” method of false should select the else-branch.

\[ \text{true} = \lambda x. \lambda y. x \]
\[ \text{false} = \lambda x. \lambda y. y \]

(Notice that true is nothing but the \text{fst} function and false nothing but the \text{snd} function.) We can now define the if operation very simply:

\[ \text{if} = \lambda p. \lambda x. \lambda y. p \ x \ y \]

Let us show how we can define various operations on booleans using these definitions:

\[ \text{not} = \lambda p. \text{if } p \text{ false } \text{true} \]
\[ \text{and} = \lambda p. \lambda q. \text{if } p \ q \text{ false} \]
\[ \text{or} = \lambda p. \lambda q. \text{if } p \text{ true } q \]

You should verify that these operations behave as expected.

3. Natural numbers The positive integers from 0 upwards are referred to as “natural” numbers. To encode natural numbers we need to figure out what their key methods are. Any number of operations present themselves: addition, multiplication, comparisons etc. However, the most basic method turns out to be the iteration operation. A natural number \( n \) allows us to iterate some function \( n \) times. Iteration in this context means applying the function repeatedly as in: \( f (f (\cdots (f x) \cdots)) \)

\[ \text{iter} : \text{nat} \to (A \to A) \to (A \to A) \]
So, \textit{iter} \( n \) takes a function \( f \) as an argument and produces a function that represents the \( n \)-fold iteration of \( f \). The trick now is to represent a natural number \( n \) as its iteration method. We write \( \nu \) for this representation, which is called “Church numerals.” Here are some examples:

\[
\begin{align*}
0 &= \lambda f. \lambda x. x \\
1 &= \lambda f. \lambda x. f x \\
2 &= \lambda f. \lambda x. f (f x) \\
3 &= \lambda f. \lambda x. f (f (f x)) \\
&\cdots \\
\nu &= \lambda f. \lambda x. f^n(x)
\end{align*}
\]

We have already seen \( 1, 2, 3 \) in Handout 2, as \textit{apply}, \textit{twice} and \textit{thrice} respectively. We can now define \textit{iter} \( x \) as simply \( x \) because if \( x \) is a Church numeral, it is an iteration method. Or, more simply, we won’t bother to write \textit{iter} \( x \) at all; we just write \( x \).

We define an operation called \textit{suc} (for “successor”) which, when applied to a Church numeral \( \nu \), returns the next Church numeral \( \nu + 1 \).

\[
suc = \lambda i. \lambda f. \lambda x. f (i f x)
\]

Note that \( \text{suc} \: \nu \: f = \lambda x. f (\nu f (x)) = \lambda x. f^{\nu+1}(x) = \lambda x. \nu + 1 \: f \: x \). We can also use the function composition notation mentioned in Handout 2 to note that \( \text{suc} \: i = \lambda f. f \circ (i f) \).

We can define an addition operation in the same way:

\[
\text{add} = \lambda i. \lambda j. \lambda f. \lambda x. (i f) \circ (j f x)
\]

In other words, \( \text{add} \: i \: j \: f = \lambda x. f \circ (i f x) \circ (j f x) \). You can verify that \( \text{add} \: \nu \: m = \nu + m \). Multiplication is defined by:

\[
\text{mult} = \lambda i. \lambda j. \lambda f. i \circ (j f)
\]

Verify that \( \text{mult} \: \nu \: m \: f = \nu \: (m \: f) = \nu \: (f^m) = f^m \circ \nu = \nu \: m \: f \). We leave out subtraction and division operations because they are somewhat involved. The “zero test” which determines whether a given Church numeral is zero or not can be defined as follows:

\[
\text{zero?} = \lambda i. \lambda x. \text{false}
\]

If \( i \) is the Church numeral \( 0 \) then \text{true} is returned. If it is \( \nu \) for some \( n > 0 \) then it applies \( \lambda x. \text{false} \) \( n \) times, producing the result \text{false}.

4. Pairs. We would like to define three operations called \textit{cons}, \textit{car} and \textit{cdr} for building pairs and selecting from pairs such that

\[
\text{car} \: (\text{cons} \: x \: y) = x \\
\text{cdr} \: (\text{cons} \: x \: y) = y
\]

(The names \textit{cons}, \textit{car} and \textit{cdr} come from the Lisp programming language designed in 1950’s; “cons” is short for “construct”; “car” and “cdr” are hangovers from the machine instructions of IBM 704 computer which were used to select components of pairs).

Think of a \textit{cons}-pair as an object. It should have two instance variables \( x \) and \( y \) and it should provide two methods for \textit{car} and \textit{cdr} which return the values of the instance variables. This is the first time that we are encountering objects with multiple methods. (Booleans only need an “if” method and natural numbers need only an “iterate” method. But pairs definitely need two methods.) To implement multiple methods, we need to use some kind of tokens to represent them, say \( t_1 \) and \( t_2 \). By passing one of these tokens to a pair, we can request the pair to return one of the components:

\[
\begin{align*}
\text{car} &= \lambda p. \text{fst} \\
\text{car} &= \lambda p. \text{snd}
\end{align*}
\]

We can use whatever is convenient for the two tokens, e.g., booleans true and false, integers 0 and 1, or strings “car” and “cdr”. Since we are using lambda calculus, which is a calculus of functions, it turns out to be efficient to use two functions \textit{fst} and \textit{snd} as the tokens. Now, we complete the definitions as follows:

\[
\begin{align*}
\text{car} &= \lambda p. \text{fst} \\
\text{car} &= \lambda p. \text{snd} \\
\text{cons} &= \lambda x. \lambda y. \lambda t. t \: x \: y
\end{align*}
\]

You can verify that the required equations are satisfied for the operations are satisfied.

It is possible to build linked lists, trees and other data structures using pairs. We omit the details here.
Applied Lambda Calculus

5. An applied lambda calculus is given by the syntax:

\[ M ::= c \mid x \mid \lambda x. M' \mid M_1 M_2 \]

where we have added the alternative \( c \) representing constants. A particular applied lambda calculus is obtained by choosing a particular set of constants. These constants may represent data values or functions (or functions on functions etc). The semantics of these constants is defined by giving a set of conversion rules, which are called “\( \delta \) conversion rules”.

6. Scheme-like calculus. We define an applied lambda calculus which is similar to a basic version of the Scheme programming language. (The Scheme programming language was defined and implemented by Sussman and Steel at MIT, as an extension of Lisp that represents a full lambda calculus incorporating higher-order functions.) The constants we use are as follows:

- boolean values: true and false, and the operation “if”.
- integers: \( \ldots, -2, -1, 0, 1, 2, \ldots \)
- arithmetic operations: “+”, “−”, “∗”, “/”, “mod”.
- comparison operations: “=”, “\( \neq \)”, “<”, “\( \leq \)”, “>”, “\( \geq \)”.
- operations for pairs: “cons”, “car”, “cdr”.
- a special value denoted “nil”, and a boolean function “null”.

The \( \delta \) conversion rules for these operations are:

\[
\begin{align*}
\text{if true } x \ y &= x \\
\text{if false } x \ y &= y \\
\text{car (cons } x \ y \text{) } &= x \\
\text{cdr (cons } x \ y \text{) } &= y \\
\text{null nil } &= \text{true} \\
\text{null (cons } x \ y \text{) } &= \text{false}
\end{align*}
\]

and an infinite collection of rules for the semantics of arithmetic/comparison operations on integers such as \( +01 = 1 \), \( <01 = \text{true} \) etc.

7. Reduction. The reduction relation \( \rightarrow \) for an applied lambda calculus is defined to include both \( \beta \)-conversion and \( \delta \)-conversion rules. The addition of the \( \delta \) conversion rules changes the properties of the reduction relation in subtle ways. Firstly, the confluence and Church-Rosser properties proved for beta-reduction do not necessarily hold for the new reduction relation. These properties need to be re-verified. For the Scheme-like language described above, the properties do continue to hold. Secondly, while the pure lambda calculus has only functions and so it is always sensible to apply one term to another, in the applied lambda calculus, we have terms such as “0”, “true” and “nil” which are apparently not functions. What is to be done if they are used in the function position of an application term? On a similar note, what is to be done if illegal values are used as arguments to primitive functions, e.g., if the first argument for “if” is something other than true/false? For the time being, we will simply ignore the problem. So, all such “erroneous” function applications remain in the terms throughout their reduction and show up in the normal forms. We will discuss type systems later which can eliminate most such erroneous terms.

8. Linked lists. We can use cons-pairs to represent linked lists as follows. A list with elements \( x_1, \ldots, x_n \) is represented as the structure \( \text{(cons } x_1 \ (\text{cons } x_2 \ (\ldots \ (\text{cons } x_n \text{nil}) \ldots)) \)\). Evidently, an empty list would be represented by just \text{nil}. Now, the pair operations have meaningful effects on linked lists: “null” checks to see if a list is the empty list, “car” accesses the first element of the linked list, “cdr” accesses the sublist with all but the first element of the argument list. Using this interpretation, we can define a function for appending two lists as follows:

\[
Y \lambda f. \lambda x. \lambda y. \text{if (null } x \text{) } y \left( \text{cons (car } x \text{) } (f \ (\text{cdr } x \text{) } y) \right)
\]
Syntactic sugar

9. The syntax of the lambda calculus is rather terse, because it was designed as a mathematical system, and it also lacks various features that we find important in programming, such as the ability to define and name functions. One might consider extending the lambda calculus in various ways to address these problems. However, such efforts would complicate the structure of the language which would add to our burden in defining semantics etc. Fortunately, there is another way. We can define new notations for term forms that are already available within our calculus and describe how the new notations can be eliminated in favour of the base calculus. In such a situation, the new notations do not add to the basic content of our calculus, but merely allow us to express it in a more convenient way. Christopher Strachey introduced the term “syntactic sugar” to describe such notations. In this section, we introduce several forms of such syntactic sugar which will allow us to work with lambda calculi more conveniently.

10. Notation for functions. In lambda calculus all functions are unary and written using prefix notation. We will treat infix and mixfix notations as syntactic sugar for the basic notation:

- All binary arithmetic operations (“+”, “-”, “*”, “/”) can be used in infix notation and they mean the corresponding prefix equivalents. For example, \( x + 1 \) is sugared notation for the term \( + x 1 \) of the base calculus.
- All comparison operations (“=”, “\neq”, “<”, “\leq”, “>”, “\geq”) can be used in infix notation.
- The applications of “if” can be written in the mixfix notation \( \text{if } p \text{ then } x \text{ else } y \). This is sugared notation for \( \text{if } p x y \).

11. Local declarations. The sugared notation \( \text{let } x = M \text{ in } N \) denotes the basic calculus expression \( (\lambda x. N) M \). This notation allows us to introduce a local name \( x \) with the definition \( M \) and to use it in the context \( N \). Note that \( x \) is a bound variable in the term \( \text{let } x = M \text{ in } N \) and that its scope is the term \( N \). If there are occurrences of \( x \) in the term \( M \), there are free occurrences.

12. Local recursive declarations. The sugared notation \( \text{letrec } x = M \text{ in } N \) denotes the basic calculus expression \( (\lambda x. N) (Y \lambda x. M) \), where \( Y \) is the recursion combinator. In this notation, \( x = M \) represents a recursive definition of \( x \). (So there are no free occurrences of \( x \) in \( M \).) This recursively defined \( x \) is then used in the context \( N \).

13. Example. As an example of the syntactic sugar introduced so far, consider the following term that defines and uses the factorial function:

\[
\text{letrec } f = \lambda n. \text{if } n = 1 \text{ then } 1 \text{ else } n \ast (f (n - 1)) \text{ in } f 5
\]

The basic “desugared” version of the term is:

\[
(\lambda f. f 5) (Y \lambda f. \lambda n. \text{if } (= n 1) 1 \text{ else } (\ast n (f (- n 1))))
\]

14. Function declarations. We will use the \( \text{let} \) as well as \( \text{letrec} \) declarations for functions with named formal parameters. The notation

\[
\text{let } f x_1 \ldots x_n = M \text{ in } N
\]

means

\[
\text{let } f = \lambda x_1. \ldots \lambda x_n. M \text{ in } N
\]

which in turn means

\[
(\lambda f. N) (\lambda x_1. \ldots \lambda x_n. M)
\]

Similarly,

\[
\text{letrec } f x_1 \ldots x_n = M \text{ in } N
\]

means

\[
\text{letrec } f = \lambda x_1. \ldots \lambda x_n. M \text{ in } N
\]

15. Lists. The Lisp/Scheme notation \( \text{list } x_1 \ x_2 \ldots \ x_n \) is used as syntactic sugar for the expression \( \text{cons } x_1 (\text{cons } x_2 (\ldots (\text{cons } x_n \text{nil}) \ldots)) \).