3.3 Programming and Data Encoding in the $\lambda$-calculus

We now consider the computational power of the $\lambda$-calculus through its expressivity as a programming model. Despite its minimality, we will see that the $\lambda$-calculus is expressive enough to encode boolean values, ordered pairs, natural numbers and lists - core data structures in any programming language. One common theme of these data structure encodings is that they carry with them the control structure associated with them i.e., we encode the data together with the operations that operate on this data.

Although not computationally efficient, these encodings serve many purposes. For a start, such encodings are of mathematical interest and frequently re-occur in theoretical studies of the model. Secondly, they also help us elevate the calculus to a level that resembles more that of a programming language. This second point is in line with a powerful concept in programming languages due to Peter Landin, whereby complex programming languages are understood by formulating them as a tiny core calculus capturing the essential mechanisms, together with a collection of derived forms for additional constructs whose behaviour is understood with respect to the core language. The third purpose, perhaps specific to our course, is that these encodings will serve as the main building blocks when arguing (informally) that the $\lambda$-calculus has the same expressive power as that of the Turing Machine model of computation (cf. Sec. 3.4).

3.3.1 Booleans and Conditionals

The first encoding we consider is that of the booleans. The $\lambda$-terms encoding the boolean values $\text{true}$ and $\text{false}$ will be typically used in conjunction with a conditional $\lambda$-term $\text{if}$ which branches according to its boolean value. Slide 102 specifies this expected behaviour, whereby an expression of the form $\text{if true }MN$ is expected to behave as the branch to be taken, $M$ (and dually for $\text{if false }MN$ and $N$). Note that here we use the notion of $\beta$-equivalence outlined earlier in Def. 72 to formally describe this relationship.

<table>
<thead>
<tr>
<th>Encoding Booleans and Conditionals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required Properties of the Encoding</strong></td>
</tr>
<tr>
<td>$\text{if true }MN \equiv_\beta M$</td>
</tr>
<tr>
<td>$\text{if false }MN \equiv_\beta N$</td>
</tr>
</tbody>
</table>

**Definition 81 (Boolean Encoding).**

\[
\begin{align*}
\text{true} &= \lambda l.\lambda r.l \\
\text{false} &= \lambda l.\lambda r.r \\
\text{if} &= \lambda c.\lambda l.\lambda r.(clr)
\end{align*}
\]

The encoding usually given for these terms is given on Slide 102. We first note that since both $\text{true}$ and $\text{false}$ are normalised terms and are distinct (even up-to $\alpha$-equivalence) then we have that $\text{true} \not\equiv_\beta \text{false}$. As it happens, the conditional $\text{if}$ is not even necessary as the boolean values are their own conditional operators. In fact, $\text{true}$ is a term that takes two branches as parameters ($l$ and $r$) and discards the second terms $r$; the $\text{false}$ is dual and discards the first argument $l$. In fact for any pair of branches $M$ and $N$ (irrespective of whether $M$ and $N$ have normal forms) we have

\[
\text{true }MN \Rightarrow M \quad \text{and} \quad \text{false }MN \Rightarrow N
\]

Thus if \( LMN \) on Slide 102 is essentially the identity function on \( L \) and the requirements stated on Slide 102 hold since:
\[
\text{if true } M N \rightarrow M \quad \text{and} \quad \text{if false } M N \rightarrow N
\]
All the usual operations on boolean values can be defined as in the case of the conditional operator. Below we list conjunction, disjunction and negation.
\[
\text{and} = \lambda b c. (\text{if } b c \text{ false}) \\
\text{or} = \lambda b c. (\text{if } b \text{ true } c) \\
\text{not} = \lambda b. \text{if } b \text{ false true}
\]

### Ordered Pairs

#### Encoding Ordered Pairs and Projections

**Definition 82** (Pairs and Projection Encoding).

\[
\begin{align*}
\text{pair} &= \lambda l.\lambda r.\lambda p.(p l r) \\
\text{fst} &= \lambda p.(p \text{ true}) \\
\text{snd} &= \lambda p.(p \text{ false})
\end{align*}
\]

**Properties of the Encoding**

\[
\begin{align*}
\text{fst } (\text{pair } M N) &\equiv_\beta M \\
\text{snd } (\text{pair } M N) &\equiv_\beta N
\end{align*}
\]

Slide 103

We can also encode constructors for ordered pairs and the corresponding projection functions \( \text{fst} \) and \( \text{snd} \) in the \( \lambda \)-calculus. Their definition is given on Slide 103 and reuse the definitions of \text{true} and \text{false} from Slide 102. Intuitively, the constructor \text{pair} first inputs the left and right component of the pair, say \( M \) and \( N \), and returns the function \( \lambda p.p M N \) which abstracts over a selection operation, \( p \), choosing either the first or second component of the pair; the abstraction effectively acts as a package around \( M \) and \( N \). For the selection function we can use the earlier definitions \text{true} and \text{false} to unpack the left and right components of the pair. Both \( \text{fst} \) and \( \text{snd} \) take a pair of the form \( \lambda p.p M N \) and parameterised \( p \) with \text{true} and \text{false} respectively. For arbitrary \( M \) and \( N \) we can thus trace the following reduction:
\[
\text{fst } (\text{pair } M N) \rightarrow_\beta (\text{pair } M N) \text{true} \\
\rightarrow_\beta (\lambda p.p M N) \text{true} \\
\rightarrow_\beta \text{true } M N \\
\rightarrow_\beta M
\]
Similarly, we can deduce \( \text{snd } (\text{pair } M N) \Rightarrow N \). Once again we note that the components \( M \) and \( N \) can be extracted irrespective of the fact that either of them may not have a normal form.

We can define \( n \)-ordered tuples in analogous fashion by either defining them directly in terms of an \( n \)-packaging abstractions and \( n \)-projection functions or otherwise as nested pairs; the latter yield a simpler encoding.
3.3.3 The Natural Numbers

There are a number of encodings for the natural numbers. Slide 104 presents the original encoding due to Alonzo Church; these are often referred to Church’s numerals and they follow the theme of earlier encodings, by putting the control structure in with the data structure.

Encoding The Natural Numbers

Definition 83 (Church Numerals).

\[
\begin{align*}
0 &= \lambda f \ x. x \\
1 &= \lambda f \ x. f \ x \\
2 &= \lambda f \ x. f(f \ x) \\
&\vdots \\
n &= \lambda f \ x. f(\ldots(f \ x)) \\
\end{align*}
\]

The main idea behind the church encoding is rather straightforward: The church numeral \( n \) is the function that applies an arbitrary function \( f \) \( n \) times to some argument \( x \). Stated otherwise, \( n \) maps \( f \) to \( f^n \) making each numeral an iteration operation.

Encoding The Natural Numbers: Properties

Definition 84 (Church Numeral Operations).

\[
\begin{align*}
suc &= \lambda n f x. f(n \ f \ x) \\
isZero &= \lambda n. (n (\lambda x. \text{false}) \text{true}) \\
\end{align*}
\]

Properties of the Encoding

\[
\begin{align*}
suc \ n &\equiv_\beta (n + 1) \\
isZero \ 0 &\equiv_\beta \text{true} \\
isZero \ (n + 1) &\equiv_\beta \text{false} \\
\end{align*}
\]
numeral encodings are closed i.e., they do not admit free variables, and also that they are normal i.e., in their normal-form. Together with the properties of Slide 105, such systems are termed as normal numeral systems. Church numerals satisfy also these additional requirements making them a normal numeral system.

### Encoding The Natural Numbers: Properties

**Definition 85** (Numeral Derived Operations).

\[
\begin{align*}
\text{add} &= \lambda m n f x. m f (n f x) \\
\text{mul} &= \lambda m n f x. m (n f) x \\
\text{exp} &= \lambda m n f x. n m f x
\end{align*}
\]

**Properties of Derived Operations**

To show \(\text{add} m n \equiv_\beta (m + n)\) \[\text{add} m n \equiv \lambda f x. m f (n f x)\]

\[\equiv \lambda f x. f^{m+n} x = m + n\]

Slide 106

Using the encodings of Def. 84 we can define operations for addition, multiplication and exponentiation in direct fashion as shown in Def. 85 on Slide 106. These operations satisfy the expected properties. Such satisfactions can be proved as shown on Slide 106. Below we show also the proof derivation for multiplication. Note that these derivations work for all church numerals \(m\) and \(n\), but necessarily for arbitrary terms \(M\) and \(N\).

To show \(\text{mul} m n \equiv_\beta (m \times n)\) we derive

\[
\begin{align*}
\text{mul} m n &= \lambda f x. m (n f) x \\
&\equiv \lambda f x. (n f)^m x \\
&\equiv \lambda f x. (f^n)^m x \\
&\equiv \lambda f x. f^{m \times n} x = m \times n
\end{align*}
\]

Defining the predecessor function for Church numerals, on which subtraction is then defined, is less trivial since we cannot rely directly on the iterative structure of these encodings. The main difficulty lies in reducing an \(n + 1\) iterator into an \(n\) iterator. More specifically, given \(f\) and \(x\) we must find a function \(g\) such that \(g^{n+1} f x\) returns \(f^n x\).

The mechanism used here is that of a pair of values that acts like a one-element delay line; for any \(x\) the pair would hold \((f(x), x)\). The function \(\text{prePr}\) defined below takes a function \(f\) and a pair of the form \((x, z)\) and returns another pair \((f(x), x)\) (discarding the second element \(z\)). This means that for any pair \((f(x), x)\), \(\text{prePr}\) would return \((f(f(x)), f(x))\). Importantly though, if we pass the pair \((x, x)\) to \(\text{prePr}\), we would obtain \((f(x), x)\), thereby skipping one iteration of \(f\) on the second argument of the pair. Joining these two facts together means that if we apply \((\text{prePr} f)\) \(n + 1\) times on a pair \((x, x)\) we would obtain the pair \((f^{n+1}(x), f^n(x))\). This is precisely the mechanism we want for the predecessor function as we can then simply take the pair \((f^{n+1}(x), f^n(x))\), discard the first element and return \(f^n x\).

\[
\text{prePr} = \lambda f p. \text{pair}(f(fst p))(fst p)
\]

\[
\text{pre} = \lambda n f x. \text{snd}(n(\text{prePr} f)(\text{pair} x x))
\]
Encoding The Natural Numbers: Predecessor

Problem
Operations so far relied on the iterative structure of numeral encodings. Harder to do for the predecessor.

Properties of Derived Operations
\[
\text{pre} (n + 1) \Rightarrow n \\
\text{pre} 0 \Rightarrow 0
\]

Definition 86 (Predecessor).
\[
\text{prePr} = \lambda f . \text{pair} (f (\text{fst} p)) (\text{fst} p) \\
\text{pre} = \lambda n . \text{snd} (n (\text{prePr} f) (\text{pair} x x))
\]

Slide 107

The reader is invited to verify the following reductions:
\[
\text{pre} (n + 1) \Rightarrow n \\
\text{pre} 0 \Rightarrow 0
\]

Definition subtraction is now straightforward using \text{pre}: subtracting \(n\) from \(m\) reduces to computing the \(n^{th}\) predecessor of \(m\). Moreover, with \text{sub} (and thus \text{pre}) we can define equality over the naturals as show below.

\[
\text{sub} = \lambda m . n . \text{pre} m \\
\text{eq} = \lambda m . \text{isZero} (\text{sub} m n)
\]

3.3.4 Lists

The final data structure encoding we will consider is that of lists. For this encoding we will represent lists in a similar fashion to how they are represented in traditional functional languages i.e., using the two constructors \text{nil} and \text{cons}. More precisely, we represent a list of the form \([x_1, x_2, \ldots, x_n]\) as the cascaded pairings \text{cons} \(x_1\) (\text{cons} \(x_2\) (\ldots(\text{cons} \(x_n\) \text{nil}))). To keep the operations as simple as possible, we shall employ two levels of pairing in our encoding:

- each "cons cell" \text{cons} \(x y\) will be represented as the nested pairs \(\langle \text{false}, (x, y) \rangle\) where \text{false} is a distinguished tag field;
- the "nil cell" \text{nil} is represented by the pair \(\langle \text{true}, \text{Id} \rangle\) where \text{Id} is just a dummy value.

The full encoding of lists and the associated operations in the \(\lambda\)-calculus are given on Slide 108. The reader should be able to verify properties such as

\[
\text{isNil nil} \Rightarrow \text{true} \\
\text{isNil} (\text{cons} M N) \Rightarrow \text{false} \\
\text{tl} (\text{cons} M N) \Rightarrow N \\
\text{hd} (\text{cons} M N) \Rightarrow M
\]

irrespective of the fact that \(M\) and \(N\) may not have normal-forms.
Recursion and Fixed Point Combinators

Recall the earlier definition of \( \text{add} \) over church numerals:

\[
\text{add} = \lambda m n f x. m f (n f x)
\]

Although this definition works fine, there are a number of shortcomings. The first is that the definition is not very perspicuous. This is due, in part, to the second shortcoming of the definition i.e., there is no abstraction between the encoding of the church numerals and the definition of \( \text{add} \). More precisely, the definition of \( \text{add} \) relies on the iterative nature of church numerals to induce repeated behaviour and, should we need to change our encoding of the natural numbers, we would also need to change the implementation of \( \text{add} \).

Recursive Definitions (1st attempt)

\[
\begin{align*}
\text{add}' &= \lambda m n. \text{if} \ (\text{isZero} \ n) \ m \ (\text{suc}(\text{add}' \ m \ (\text{pre} \ n))) \\
\text{dec}' &= \lambda m n. \text{if} \ (\text{isZero} \ n) \ m \ (\text{pre}(\text{dec}' \ m \ (\text{pre} \ n))) \\
\text{mul}' &= \lambda m n. \text{if} \ (\text{isZero} \ n) \ m \ (\text{add} \ n \ (\text{mul}' \ m \ (\text{pre} \ n)))) \\
\text{eq}' &= \lambda m n. \text{if} \ (\text{and} \ (\text{isZero} \ m) \ (\text{isZero} \ n)) \text{true} \\
& \quad \text{if} \ (\text{or} \ (\text{isZero} \ m) \ (\text{isZero} \ n)) \text{false} \ (\text{eq}' \ (\text{pre} \ m) \ (\text{pre} \ n)))
\end{align*}
\]

And also for lists:

\[
\begin{align*}
\text{sum} &= \lambda l. \text{if} \ (\text{isNil} \ l) \ 0 \ (\text{add} \ (\text{hd} \ l) \ (\text{sum} \ (\text{tl} \ l))) \\
\text{app} &= \lambda k l. \text{if} \ (\text{isNil} \ k) \ l \ (\text{cons} \ (\text{hd} \ k) \ (\text{app} \ l \ (\text{tl} \ k)))
\end{align*}
\]

It would be desirable to encode repeated behaviour in a uniform manner, independent of the details of the function being defined and the representation of the data structures on which it operates. As an example of what we mean by this, consider the alternative definitions shown on Slide 109. Notice the pleasing regular
structure in each of these recursive definitions - once you understand the first definition, it is very easy to understand the rest; the same cannot be said of \texttt{add} and \texttt{mul} of Def. 85. But each one of these definitions suffers from the same problem: contrary to all the macro definitions we have seen so far, they all refer to themselves in their body definitions. Thus expanding these macro definitions will never terminate in a fully defined \lambda-\text{term}.

The work around this problem is to use what is termed as a \textit{fixed point combinator} - some term \texttt{Y} such that for any term \texttt{M} we have the equality \texttt{Y M} \equiv \texttt{M (Y M)}. Such a term is called a fixed point because fixed points for a function \texttt{F} are terms \textit{X} satisfying \texttt{F(X)} = \texttt{X}; here this value \textit{X} for the function \texttt{M} would be \texttt{Y M} i.e., read the earlier \beta-equivalence in reverse \texttt{M (Y M)} \equiv \texttt{Y M}. Combinators are \lambda-\text{terms} containing no free variables. This fixed point term \texttt{Y} thus allows us to code recursion in the macro definitions without the need to refer to the macro itself, whereby the law \texttt{Y M} \equiv_\beta \texttt{M (Y M)} would allow us to unfold the recursion as many times as necessary.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Recursive Definitions (2nd correct attempt)} \\
\hline
\texttt{add} = \texttt{Y} \lambda g. \lambda m. n. \text{if} (\texttt{isZero} n) \ m \ \text{(suc(g m (\text{pre} n))}) \\
\texttt{dec} = \texttt{Y} \lambda g. \lambda m. n. \text{if} (\texttt{isZero} n) \ m \ \text{(pre(g m (\text{pre} n))}) \\
\texttt{mul} = \texttt{Y} \lambda g. \lambda m. n. \text{if} (\texttt{isZero} n) \ n \ \text{(add n (g m (\text{pre} n)))} \\
\texttt{eq} = \texttt{Y} \lambda g. \lambda m. n. \text{if} (\text{and} (\texttt{isZero} m) (\texttt{isZero} n)) \ \text{true} \\
\hspace{2cm} (\text{if} (\text{or} (\texttt{isZero} m) (\texttt{isZero} n)) \ \text{false} \ (g \ (\text{pre} m) \ (\text{pre} n))) \\
\hline
\end{tabular}
\end{center}

Slide 110

Consider the now corrected recursive definitions on Slide 110. The macro body definitions never mention the macro name being defined; instead the definitions first abstract over some function \texttt{g} and replace the previous cyclic reference with a bound instance of this variable \texttt{g}; this function is then applied to the fixed point combinator \texttt{Y}. To see how the new encoding works we will consider the expansion for \texttt{add} (the remaining definitions work in the same way):

\texttt{add} = \texttt{Y} \lambda g. \lambda m. n. \text{if} (\texttt{isZero} n) \ m \ \text{(suc(g m (\text{pre} n))}) \\
\equiv_\beta \lambda g. \lambda m. n. \text{if} (\texttt{isZero} n) \ m \ \text{(suc(g m (\text{pre} n))}) \ (\texttt{Y} \lambda g. \lambda m. n. \text{if} (\texttt{isZero} n) \ m \ \text{(suc(g m (\text{pre} n))}) \\
= \lambda g. \lambda m. n. \text{if} (\texttt{isZero} n) \ m \ \text{(suc(g m (\text{pre} n)) \ add} \\
\longrightarrow_\beta \lambda m. n. \text{if} (\texttt{isZero} n) \ m \ \text{(suc(\text{add} \ m \ (\text{pre} n))})

The fixed point combinator we consider here \texttt{Y} was discovered by Haskell B. Curry and is defined as the \lambda-term of Def. 88 on Slide 111. It is easy to see that this definition satisfies the fixed point equation outlined earlier \texttt{Y M} \equiv_\beta \texttt{M (Y M)} (although \texttt{Y M} \not\equiv \texttt{M (Y M)}). We note that there is a whole family of fixed point combinators. For instance, Alan Turing defined another one:

\[ \Theta = (\lambda x. y. (x x y)) (\lambda x. y. (x x y)) \]
Fixed Point Combinators

Definition 88 (Y Combinator).

\[ Y = \lambda f. (\lambda x. f (x x)) \]

Proving the Fixed Point Equation

\[ Y M \rightarrow \beta (\lambda x. M (x x)) (\lambda x. M (x x)) \]
\[ \rightarrow \beta M ((\lambda x. M (x x)) (\lambda x. M (x x))) \]
\[ = M (Y M) \]

One can verify that \( \Theta M \Rightarrow M (\Theta M) \), which therefore implies that \( \Theta M \equiv_\beta M (\Theta M) \).

Notice that functions defined with the help of fixed point combinators, such as \( \text{add} \), are unlikely to have a normal form. In fact one can check that \( \text{add} \) under normal order reduction reduces forever. Nevertheless, we can still compute with such functions. For instance, the expression

\[ \text{add} \ n \ m \]

does have a normal form \( i.e., n + m \). Due to the possibility of infinite reductions, functions defined with the help of fixed point combinators usually assume a Call-by-Name reduction semantics. This is because Call-by-Name reductions do not evaluate under \( \lambda \)-abstractions. Thus, whereas the term \( \lambda x. \Omega \) does not have a normal form (and a normal-order reduction would reduce forever) the term is treated as a value under a Call-by-Name semantics and not evaluated further.

3.3.6 Exercises

1. Prove that \( \text{and} \ \text{false} \ \text{true} \equiv_\beta \text{false} \) and that \( \text{or} \ \text{false} \ \text{true} \equiv_\beta \text{false} \).
2. Give a direct encoding for triples and the associated projection functions \( \text{fst} \), \( \text{snd} \) and \( \text{thd} \).
3. Give an indirect encoding for triples and the associated projection functions in terms of tuples and their projection functions \( \text{fst} \) and \( \text{snd} \).
4. Prove that Church Numerals satisfy the properties on Slide 105.
5. Prove that \( \text{exp} \ n \ m \equiv_\beta m^n \).
6. Prove that \( \text{pre} \ 0 \equiv_\beta 0 \) and that \( \text{pre} \ (n + 1) \equiv_\beta n \).
7. Derive \( \text{sub} \ 2 \ 1 \Rightarrow 1 \).
8. Consider the alternative definition for addition defined in terms of the successor function.

\[ \text{add} = \lambda m \cdot n \cdot n \text{succ} \ m \]

Prove that \( \text{add} \ m \ n \equiv_\beta m + n \).
9. Use the encoding of lists to obtain an encoding for the natural numbers. Encode the operations \( \text{isZero} \), \( \text{suc} \) and \( \text{pre} \) for such an encoding.
10. Prove that \( \Theta M \equiv_\beta M (\Theta M) \).