3.3 Programming and Data Encoding in the λ-calculus

We now consider the computational power of the λ-calculus through its expressivity as a programming model. Despite its minimalism, we will see that the λ-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 λ-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 λ-terms encoding the boolean terms true and false will be typically used in conjunction with a conditional λ-term if which branches according to its boolean value. Slide 102 specifies this expected behaviour, whereby an expression of the form if true M N is expected to behave as the branch to be taken M (and dually for if false M N) Note that here we use the notion of β-equivalence outlined earlier in Def. 71 to formally describe this relationship.

<table>
<thead>
<tr>
<th>Encoding Booleans and Conditionals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Properties of the Encoding</td>
</tr>
<tr>
<td>if true M N ≡β M</td>
</tr>
<tr>
<td>if false M N ≡β N</td>
</tr>
<tr>
<td>Definition 81 (Boolean Encoding).</td>
</tr>
<tr>
<td>true = λ.l.λ.r.l</td>
</tr>
<tr>
<td>false = λ.l.λ.r.r</td>
</tr>
<tr>
<td>if = λ.c.λ.l.λ.r.(c l r)</td>
</tr>
</tbody>
</table>

Slide 102

The encoding usually given for these terms in given on Slide 102. We first note that since both true and false are normalised terms and are distinct (even up-to α-equivalence) then we have that true ≠β false. As it happens, the conditional if is not even necessary as the boolean values are their own conditional operators. In fact, true is a term that takes two branches as parameters (l and r) and discards the second terms r); the 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

true M N ⇒ M and false M N ⇒ N
Thus if $L M N$ on Slide 102 is essentially the identity function on $L$ and the requirements stated on Slide 102 hold since:

$$\text{if } \text{true } M N \implies M \quad \text{and} \quad \text{if } \text{false } M N \implies N$$

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.

$$\text{and} = \lambda b c. (\text{if } b \text{ c } \text{false})$$
$$\text{or} = \lambda b c. (\text{if } b \text{ true } c)$$
$$\text{not} = \lambda b. \text{if } b \text{ false } \text{true}$$

### 3.3.2 Ordered Pairs

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

$$\text{pair} = \lambda l. \lambda r. (p \ell r)$$
$$\text{fst} = \lambda p. (p \text{ true})$$
$$\text{snd} = \lambda p. (p \text{ false})$$

**Properties of the Encoding**

$$\text{fst} (\text{pair } M N) \equiv_{\beta} M$$
$$\text{snd} (\text{pair } M N) \equiv_{\beta} N$$

---

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:

$$\text{fst} (\text{pair } M N) \implies_{\beta} (\text{pair } M N) \text{true}$$
$$\implies (\lambda p. p M N) \text{true}$$
$$\implies_{\beta} \text{true } M N$$
$$\implies M$$

Similarly, we can deduce $\text{snd} (\text{pair } M N) \implies N$. 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).

\[
0 = \lambda f . x \\
1 = \lambda f . f x \\
2 = \lambda f . f (f x) \\
\vdots \\
n = \lambda f . f (\cdots (f x))
\]

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).

\[
suc = \lambda n . f . (f n) \\
isZero = \lambda n . (\lambda x \text{. false}) \text{ true}
\]

**Properties of the Encoding**

\[
suc n \equiv \beta (n + 1) \\
isZero 0 \equiv \beta \text{ true} \\
isZero (n + 1) \equiv \beta \text{ false}
\]

The least set of operations operating on the encoding of the Naturals are typically \( \text{suc} \), the successor function, and \( \text{isZero} \), a test for zero. These operations are defined in Def. 84 on Slide 105 and their expected properties are also stated on this slide. Thus typically, \( \text{isZero} \) returns \text{true} only when applied to \( 0 \) and \text{false} when applied to any other numeral encoding whereas \( \text{suc} \) returns the next numeral encoding. The reader is invited to check that the properties on Slide 105 are indeed satisfied by the church numerals and their associated operations.

Sometimes two additional conditions are also required of natural number encodings, namely that the
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).

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

Properties of Derived Operations

To show \(\text{add } m n \equiv_\beta (m + n)\) \(\iff \lambda f.x. m f (n f x)\)
\(\iff \lambda f.x. f^m (f^n x)\)
\(= \lambda f.x. f^{m+n} x = m + n\)

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

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

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 use 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 \(\text{prePr}\) define 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 \(f^n(x)\).

\[
\begin{align*}
\text{prePr} &= \lambda f.p.\text{pair}(f(p)\text{fst}p)\\
\text{pre} &= \lambda n x.\text{snd}(n(\text{prePr} f)(\text{pair} xx))
\end{align*}
\]
The reader is invited to verify the following reductions:

\[
\begin{align*}
\text{pre}(n + 1) & \Rightarrow n \\
\text{pre} 0 & \Rightarrow 0
\end{align*}
\]

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.

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

### 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 to levels of pairing in our encoding. Thus each “cons cell” \text{cons} \(x\) \(y\) will be represented as the nested pairs \text{false} \((x, y)\) where \text{false} is a distinguished tag field; dually, the “nil cell” \text{nil} is represented by the pair \((\text{true}, \text{Id})\) where \text{Id} is just a dummy value.

<table>
<thead>
<tr>
<th>Encoding The Natural Numbers: Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition 86</strong> (List Encodings),</td>
</tr>
</tbody>
</table>
| \begin{align*}
\text{nil} &= \text{pair} \, \text{true} \, \text{Id} \\
\text{cons} &= \lambda x. y. \text{pair} \, \text{false} \, (\text{pair} \, x \, y) \\
\text{isNil} &= \text{fst} \\
\text{hd} &= \lambda l. \text{fst} \, (\text{snd} \, l) \\
\text{tl} &= \lambda l. \text{snd} \, (\text{snd} \, l)
\end{align*} |

### Slide 107

The full encoding of lists and the associated operations in the \(\lambda\)-calculus are given on Slide 107. The reader should be able to verify properties such as

\[
\begin{align*}
\text{isNil} \, \text{nil} & \Rightarrow \text{true} \\
\text{isNil} \, (\text{cons} \, M \, N) & \Rightarrow \text{false} \\
\text{tl} \, (\text{cons} \, M \, N) & \Rightarrow N \\
\text{hd} \, (\text{cons} \, M \, N) & \Rightarrow M
\end{align*}
\]

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

### 3.3.5 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 add. More precisely, the definition of 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 add.

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 "Recursive Definitions (1st attempt)". 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 add and 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 λ-term.

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

Consider the now corrected recursive definitions on Slide 108. The macro body definitions never mention the macro name being defined; instead the definitions first abstract over some function g and replace the previous cyclic reference with a bound instance of this variable g; this function is then applied to the fixed point combinator Y. To see how the new encoding works we will consider the expansion for add (the
Recursive Definitions (2nd correct attempt)

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

And also for lists:

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

remaining definitions work in the same way):

\[
\begin{align*}
\text{add} &= \text{Y } \lambda g.\lambda m.n.\text{if (isZero n) m } (\text{suc} (g \ m \ (\text{pre} \ n))) \\
&\equiv_\beta \lambda g.\lambda m.n.\text{if (isZero n) m } (\text{suc} (g \ m \ (\text{pre} \ n))) (\text{Y } \lambda g.\lambda m.n.\text{if (isZero n) m } (\text{suc} (g \ m \ (\text{pre} \ n)))) \\
&= \lambda g.\lambda m.n.\text{if (isZero n) m } (\text{suc} (g \ m \ (\text{pre} \ n))) \ \text{add} \\
&\longrightarrow_\beta \lambda m.n.\text{if (isZero n) m } (\text{suc} \ (\text{add} \ m \ (\text{pre} \ n)))
\end{align*}
\]

The fixed point combinator we consider here \text{Y} was discovered by Haskell B. Curry and is defined as the \(\lambda\)-term of Def. 87 on Slide 110. It is easy to see that this definition satisfies the fixed point equation outlined earlier \(Y M \equiv_\beta M (Y M)\) (although \(Y M \not\equiv M (Y M)\)). We note that there is a whole family of fixed point combinators. For instance, Alan Turing defined another one:

\[
\Theta = (\lambda x.y.y(x \ x \ y)) \ (\lambda x.y.y(x \ x \ y))
\]

One can verify that \(\Theta M \Longrightarrow M (\Theta M)\), 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 reduced forever. Nevertheless, we can still computed with such functions. For instance, the expression

\[
\text{add} n m
\]

does have a normal form \textit{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}.
Slide 110

2. Give a direct encoding for triples and the associated projection functions \texttt{fist}, \texttt{snd} and \texttt{thd}.

3. Give an indirect encoding for triples and the associated projection functions in terms of tuples and their projection functions \texttt{fist} and \texttt{snd}.

4. Prove that Church Numerals satisfy the properties on Slide 105.

5. Prove that \texttt{exp m n} \equiv_\beta \texttt{m}^\texttt{n}.

6. Prove that \texttt{pre 0} \equiv_\beta 0 and that \texttt{pre (n + 1)} \equiv_\beta \texttt{n}.

7. Derive \texttt{sub 2 1} \rightarrow 1.

8. Consider the alternative definition for addition defined in terms of the successor function.

\[
\texttt{add} = \lambda m. n. \texttt{suc} m
\]

9. Use the encoding of lists to obtain an encoding for the natural numbers. Encode the operations \texttt{isZero}, \texttt{suc} and \texttt{pre} for such an encoding.

10. Prove that \texttt{ΘM} \equiv_\beta M(ΘM).

3.4 The \lambda-calculus and Turing Machines

We conclude this section by arguing, albeit informally, that the \lambda-calculus computational model is computationally as powerful as the Turing Machine computational model. We start off by showing that any Turing Computable function can also be computed by a \lambda-term.

**Theorem 88.** Any Turing Machine \texttt{M} can be simulated by a \lambda-term.

(*Proof Outline.*) The argument here mirrors that for universal Turing Machines. We recall that any Turing Machine was defined by the three sets \texttt{Q}, \texttt{Σ} and \texttt{δ}. Since all three sets are finite, we can assign a unique church numeral for every element of the sets \texttt{Q} and \texttt{Σ}. The transition rules in \texttt{δ} will then use these church numeral mappings to encode transitions as nested pairs.

The state of a Turing Machine is described by a Turing Machine configuration, i.e., a tuple consisting of the current state and the tape contents. A single step computation in a Turing Machine can be seen as a transition from one configurations to another.
Using the encodings of Sec. 3.3 we can easily encode a Turing Machine configuration as a pair holding the church numeral mapping to the current state and a term representing the current contents of the tape. Recall that we described the contents of the tape as the three components $xyi$; we can easily encode this in the $\lambda$-calculus as a nested pair where the first component of the pair is the church numeral corresponding to the current head symbol $a$ and the second component is another pair, containing two lists of church numerals encoding the left and right tape contents $x$ and $y$.

The entire Turing Machine computation can then be simulated by a recursive $\lambda$-term that repeatedly takes configuration encodings as input and returns configuration encodings as output. The simulation of the next step to be taken by the Turing Machine would rely on state and symbol matching with the left hand side of some transition rule; since both these elements are encoded as church numerals, this matching amounts to equality checking amongst numerals, i.e., $eq$ defined earlier on Slide 109. Modifying a configurations encoding amounts to a series of operations on pairs and list defined earlier.

The converse can also be shown to be true. More precisely, any computation in the $\lambda$-calculus can be simulated by a Turing Machine.

**Theorem 89.** The reduction of any $\lambda$-term $M$ can be simulated by some Turing Machine $M$.

*(Proof Outline).* Firstly, observe that any $\lambda$-term can be encoded on a Turing Machine tape, just as we did in the case of Universal Turing Machines. For instance, the set of variables is enumerable and therefore we can assign an ordering to these variables and denote each variable as a a number, encoded as a sequence of 0s and 1s. Also, application and abstraction can be described using special tape symbols such as $\@$ and $\$$.

At the heart of any $\lambda$-term reduction is the substitution operation. This can be expressed in terms of the following simple Turing Machine operations for which we have already built Turing Machines for:

- Matching of variables (to be substituted)
- String shifting (to make space for the $\lambda$-term to be substituted for)
- String overwriting with the $\lambda$-term to be substituted for (to perform the substitution for the variable in the term)

The only thing left to specify is the actual computation. To simplify this, we will assume that every bound variable and free variable in the term is unique (we can easily adjust for this using a simple pass over the term and making $\alpha$-conversion substitutions.) Computation boils down to finding the next redex to reduce in the term and performing the actual substitution appropriately. To keep things simple, we here only illustrate this for the Call-by-Name reduction strategy - other strategies are similar albeit slightly more complex. Our Turing Machine simulation uses two tapes, the input/output tape and the stack tape (which, as the name suggests, is used as a stack i.e., LIFO of terms). The algorithm starts with the $\lambda$-term encoding placed on the input/output tape and an empty stack tape and proceeds by case analysis of the subterm currently being inspected:

1. Determine the form of the term:

   **Application:** Open the application term by removing the symbol. Place the right term on the stack tape, leave the left term on the input/output tape to be evaluated further. Goto step 1.

   **Abstraction:**
   - If the Stack tape is empty do nothing.
   - Else pop the first term, say $M$. Open up the abstraction, taking note of the bound variable, say $x$, and leaving only the abstraction body on the input/output tape. Perform the substitution of every occurrence of $x$ in the expression on the input/output tape by the popped term $M$. Goto Step1.

   **Variable:** do nothing.
2. pop the remaining terms left on the stack and reconstruct the applications opened earlier using the now reduced left term on the input/output tape.

4 Conclusion

This concludes our brief and somewhat informal overview of two main computational models, namely Turing Machines and $\lambda$-calculus. Through these topics the student should have a better understanding of the limits of computation, which also completes the discussion initiated in the Formal languages course.

The choice of the computational models explored was motivated by the programming languages in use nowadays and their direct link with these two models. The reader should not find it hard to establish a direct link between Turing Machines and the Von Neumann architecture used in nearly every electronic computer nowadays. The link is also true for languages that are tightly bound to this underlying architecture organization such as assembly languages and, at as slightly higher level of abstraction, the ubiquitous C programming language. The link between functional programming languages and the $\lambda$-calculus is perhaps even more direct and we have already mentioned examples such as Standard ML, OCaml and Haskell. Modern programming languages now also incorporate a mixture of the two programming styles such as imperative languages with an explicit notion of state through some form of memory organisation, but also allowing for the organisation of code through function definitions and function calls. Pascal is one obvious example of such a language. Students who are interested further in these topic are encouraged to take courses such as Principles of Programming Languages, where a more detailed study of these phenomena is explored.