Lecture Notes on the $\lambda$-Calculus

PETAR MAKSIMOVIĆ, Imperial College London, UK

1 INTUITION AND SYNTAX

The third model of computation introduced in this course, after register machines and Turing machines, is the $\lambda$-calculus. It was first created by Alonzo Church in the 1930s, as a formalism for the study of computable functions. Despite its simplicity, $\lambda$-calculus is as expressive as some modern programming languages, and one can think of it as the first functional programming language.

Programs in the $\lambda$-calculus are called $\lambda$-terms. We denote them by $M, N \in T$, and they come in the following three flavours

$$T \ni M ::= x \quad \text{(Variable)}$$
$$\quad \quad | \quad \lambda x. M \quad \text{(Abstraction)}$$
$$\quad \quad | \quad (M N) \quad \text{(Application)}$$

the intuition behind which is as follows:

1. Variables are standard; we will commonly denote them by $x, y, z \in X$.
2. Abstractions represent function definitions. An abstraction is of the form $\lambda x. M$ and has two components: the formal parameter of the function, $x$, also called the binder; and the function body, $M$. Note that all functions in the $\lambda$-calculus are anonymous, meaning that they have no associated name. Anonymous functions exist in various programming languages: for example, one could write an anonymous function for computing the square of a given number in Haskell as $(\backslash x \rightarrow x \times x)$, and in JavaScript as $(x \Rightarrow x \times x)$. For readability, we will often contract multiple abstractions into one; for example,

$$\lambda x. \lambda y. \lambda z. M \quad \text{will contract to} \quad \lambda x y z. M.$$ 

3. Applications represent function calls. An application is of the form $(M N)$, and has two components: the function that is being called, $M$; and the argument that is passed into the function, $N$. It will be clear how application works later on, when we get to the semantics and, in particular, $\beta$-reduction. The parentheses may be omitted, in which case application should be treated as left-associative, meaning that:

$$M_1 M_2 M_3 \quad \text{corresponds to} \quad (M_1 M_2) M_3, \quad \text{not to} \quad M_1 (M_2 M_3).$$

1.1 Free and Bound Variables

1.2 $\alpha$-Equivalence

As functions are at the core of the $\lambda$-calculus, a legitimate question arises: when should two functions be considered equal? Syntactic equality would be one criterion, but the $\lambda$-calculus goes one step further and takes into consideration the behaviour of the functions. In particular, abstractions such as the following are thought of as equal:

$$\lambda x. x \quad \lambda y. y \quad \lambda z. z \quad \lambda t. t$$

This is intuitive for programmers, as such a renaming of variables clearly has no impact on the output of the function; one might write the squaring function in Haskell as $(\backslash x \rightarrow x \times x)$, another might write it as $(\backslash y \rightarrow y \times y)$, but those two functions would perform the same computation.

This extended notion of equality is called $\alpha$-equivalence. We will not formally define $\alpha$-equivalence, but will give an intuitive strategy. Two $\lambda$-terms $M$ and $N$ are $\alpha$-equivalent, denoted by $M =_\alpha N$, if and only if:
(1) their structure matches;
(2) they have exactly the same free variables in exactly the same places; and
(3) the bound variables of one term can be renamed to match those of the other, and vice versa.

We illustrate how this strategy works on a few examples:

\[
\lambda x.x \neq_{\alpha} \lambda y.x_y \quad \text{(structure does not match)}
\]

\[
\lambda x. (\lambda z.x z) y \neq_{\alpha} \lambda y. (\lambda t.y) t \quad \text{(free variables do not match)}
\]

\[
\lambda x. (\lambda z.x z) y \equiv_{\alpha} \lambda w. (\lambda t.w) y \quad \text{(can rename } x \text{ to } w \text{ and } z \text{ to } t)\]

1.3 Substitution

Computation in the \(\lambda\)-calculus is done through substitution, making the mastery of this concept essential. Substitution of a variable \(y\) for a \(\lambda\)-term \(N\) inside a \(\lambda\)-term \(M\), denoted by \(M[N/x]\), is defined inductively on the structure of \(M\), as follows:

\[
x[N/y] := \begin{cases} 
N, & \text{if } x = y \\
x, & \text{otherwise}
\end{cases}
\]

\[
(\lambda x.M)[N/y] := \begin{cases} 
\lambda x.M, & \text{if } x = y \\
\lambda x.M[N/y], & \text{if } x \neq y \text{ and } x \notin FV(N) \\
\lambda z.M[z/x][N/y], & \text{if } x \neq y \text{ and } x \in FV(N), \text{ for some fresh } z
\end{cases}
\]

\[
(M_1 M_2)[N/y] := (M_1[N/y] M_2[N/y])
\]

The variable and the application cases are straightforward, but the three abstraction cases require a bit of clarification:

(1) the function parameter cannot be substituted for;
(2) the substitution proceeds inductively if the function parameter is not free in the expression that is to be substituted;
(3) substitution cannot result in a free variable being captured by the function parameter; in this case, the function parameter first needs to be renamed to an arbitrary fresh variable (resulting in an \(\alpha\)-equivalent abstraction) before the substitution can be performed.

Here are some examples of how substitution works in practice:

\[
x[y/x] = y
\]

\[
z[y/x] = z
\]

\[
((x y)(y z))[y/x] = (y y)(y z)
\]

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

\[
(\lambda z.x z)[y/x] = \lambda z.y z
\]

\[
(\lambda y.x y)[y/x] = \lambda z.y z
\]

2 SEMANTICS: \(\beta\)-REDUCTION AND ITS PROPERTIES

The core computation in the \(\lambda\)-calculus is described using a single reduction rule:

\[
(\lambda x.M) N \rightarrow_{\beta} M[N/x]
\]

We call this reduction \(\beta\)-reduction. Given our interpretation of the abstraction and application, the \(\beta\)-reduction performs the actual function call: that is, it substitutes the argument passed to the function \((N)\) into the function body \((M)\).

We call terms of the form \((\lambda x.M) N\) redexes, meaning that they can be reduced using \(\beta\)-reduction. Given that \(\lambda\)-terms can contain multiple redexes, we need several congruence rules that will allow us to perform \(\beta\)-reduction in any part of a given \(\lambda\)-term:
### Lecture Notes on the \(\lambda\)-Calculus 3

\[
\begin{align*}
\frac{M \rightarrow_\beta M'}{\lambda x. M \rightarrow_\beta \lambda x. M'} & \quad \frac{M \rightarrow_\beta M'}{M N \rightarrow_\beta M' N} & \quad \frac{N \rightarrow_\beta N'}{M N \rightarrow_\beta M N'} \\
\end{align*}
\]

\[
\frac{M =_\alpha M'}{M' \rightarrow_\beta N' & \quad \frac{N' =_\alpha N}{M \rightarrow_\beta N}}
\]

These rules are standard and similar in style to those you’ve previously seen in the small-step semantics of the While language. Note that the last rule allows for arbitrary \(\alpha\)-reduction to take place before and after the reduction.

**Exam:** We will not ask for derivation trees using these congruence rules.

We also need to define, analogously to the While small-step semantics, a multi-step \(\beta\)-reduction, denoted by \(\rightarrow^*\), and accounting for \(\alpha\)-equivalence, reflexivity, and transitivity:

\[
\begin{align*}
\text{\(\alpha\), Reflexivity} & \quad \frac{M =_\alpha M'}{M \rightarrow^* M'} & \quad \frac{M \rightarrow_\beta M''}{M' \rightarrow^* M''} & \quad \frac{M'' \rightarrow^* M'}{M \rightarrow^* M'} \\
\end{align*}
\]

**Example.** Consider the \(\lambda\)-term:

\[
(\lambda x. x) ((\lambda x. y) z).
\]

This term as a whole is a redex, but it also contains a smaller redex, \((\lambda x. y) z\). When performing \(\beta\)-reduction, unless instructed otherwise, we are free to choose which redex to reduce first.

On the other hand, had we chosen the smaller redex first, we would have obtained the following reduction sequence:

\[
(\lambda x. x) ((\lambda x. y) z) \rightarrow_\beta (\lambda x. x) y \\
\rightarrow_\beta y y
\]

(small redex: \((\lambda x. x) y\))

Observe that all of the reduction sequences resulted in the same final \(\lambda\)-term, \(y y\). This is not a coincidence, but rather an illustration of one important property of \(\beta\)-reduction.

**Theorem 2.1 (Confluence).** The \(\lambda\)-calculus is confluent:

\[
\forall M, M_1, M_2. M \rightarrow^*_\beta M_1 \Rightarrow M \rightarrow^*_\beta M_2 \Rightarrow (\exists M'. M_1 \rightarrow^*_\beta M' \land M_2 \rightarrow^*_\beta M')
\]

What the confluence theorem tells us it that, regardless how we choose to reduce a given \(\lambda\)-term, we can always direct that reduction further to end up at the same \(\lambda\)-term.

### 2.1 Normal Forms and Reduction Strategies

We say that a \(\lambda\)-term is in normal form if it contains no redexes, and that it has a normal form (or that it is normalising) if it reduces to a term in normal form. For example, the term \((\lambda x. x)(\lambda x. x)\) normalises as follows:

\[
(\lambda x. x)(\lambda x. x) \rightarrow_\beta (\lambda x. x) (\lambda x. x) \rightarrow_\beta \lambda x. x
\]

It can be proven that if a term is normalising, then its normal form is unique up to \(\alpha\)-equivalence. However, the \(\lambda\)-calculus as a whole is not normalising, meaning that there exist \(\lambda\)-terms whose reduction will never terminate. One example of this is the term

\[
D \equiv (\lambda x. x x) (\lambda x. x x)
\]

which, as you can check, will keep reducing to itself.
Interestingly, even if a term is normalising, it is not guaranteed that an arbitrary order of reduction will reach it. For example, if we consider the term

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

and choose the entire term as the redex, we immediately arrive at its normal form, $y$. On the other hand, if we keep choosing the redex $D$, the reduction will never terminate, as shown above. Various reduction strategies have been studied extensively in the literature; we will be considering the following three:

1. **Normal-order**: evaluate the leftmost redex that is not contained in another redex;

2. **Call-by-name**: do not reduce function arguments before the function call, do not reduce under $\lambda$ (i.e. in uninstantiated function bodies);

3. **Call-by-value**: reduce function arguments fully before the function call, do not reduce under $\lambda$.

These strategies all have their pros and cons. For example, normal-order will always reduce a term to its normal form (if the term has a normal form), but will also perform reductions in uninstantiated function bodies, which is not normally done in programming and which the other two strategies forbid. Call-by-name finds more normal forms (i.e. terminates more often) than call-by-value, but since it passes the arguments into the function body without reducing them, the same argument may be reduced multiple times. Finally, call-by-value requires arguments to be fully reduced before the function call, but that means that it will terminate the least often of the three.

Let’s see how these strategies can be used in practice, on the following $\lambda$-term:

$$(\lambda xy. x \ y \ x) \ t \ u \ ((\lambda x y z. ((\lambda x. x \ x) \ y)) \ v \ ((\lambda x. x \ y) \ w))$$

**Normal-order.** The normal-order reduction of the example term proceeds as follows, with the redex reduced in each step highlighted in yellow:

1. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$
2. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$
3. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$
4. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$
5. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$
6. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$
7. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$
8. $$\beta (\lambda x y z. ((\lambda x. x \ x) \ y)) (\lambda x. x \ y) \ w))$$

Let’s analyse the steps one-by-one:

1. The term being analysed has four redexes: $(\lambda xy. x \ y \ x) \ t$, $(\lambda x y z. ((\lambda x. x \ x) \ y)) \ v$, $(\lambda x. x \ x) \ y$, and $(\lambda x. x \ y) \ w$. The third redex is contained within the second, so it cannot be chosen, and of the remaining three, the first is the leftmost.

2. This step is analogous to the previous, with the first redex being the newly created redex, $(\lambda y. t \ y \ t) \ u$.

3. There are three redexes to consider: $(\lambda xy z. ((\lambda x. x \ x) \ y)) \ v$, $(\lambda x. x \ x) \ y$, and $(\lambda x. x \ y) \ w$. We cannot choose the second one, because it is contained within the first, and of the remaining two, the first is the leftmost.

4. Again, we have three redexes to choose from: the newly created $(\lambda y z. v ((\lambda x. x \ x) \ y)) ((\lambda x. x \ y) \ w)$, $(\lambda x. x \ x) \ y$, and $(\lambda x. x \ y) \ w$. We cannot choose the last two, since they are both contained in the first, so we have to choose the first.
(5) At this point, we have two redexes to consider: \((\lambda x. x) ((\lambda x. y) w) \) and \((\lambda x. x) w\). We must choose the first, as it contains the second. Note that this is the first step in which we are doing a reduction under a \(\lambda\): this redex is inside a function body that has not been fully instantiated yet (its \(z\) parameter is still bound by the enclosing abstraction). This type of reduction is forbidden by call-by-name and call-by-value.

(6) Now, we have two redexes of the form \((\lambda x. x) y\); we choose the leftmost one.

(7) There is only one redex to choose, \((\lambda x. x) y\); we reduce it and we are done.

Note that we had to reduce one of the function arguments, \((\lambda x. x) y\), twice, because we passed it unreduced into the function body.

**Call-by-name.** One of the call-by-name reductions of the example term proceeds as follows, with the redex reduced in each step highlighted in yellow:

\[
1) \ (\lambda x. x) y \ x \ t \ u \ ((\lambda x. x) y) v \ ((\lambda x. x) w) \\
2) \ \rightarrow_\beta \ (\lambda y. t y t) u \ ((\lambda x. x) y) v \ ((\lambda x. x) w) \\
3) \ \rightarrow_\beta \ (t u t) ((\lambda x. x) y) v \ ((\lambda x. x) w) \\
4) \ \rightarrow_\beta \ (t u t) ((\lambda x. x) y) ((\lambda x. x) w) \\
5) \ \rightarrow_\beta \ (t u t) \ ((\lambda x. x) ((\lambda x. x) w)) \\
6) \ (t u t) \ ((\lambda x. x) ((\lambda x. x) w)) \\
\]

The steps are the same as for normal-order, but the reasoning is different. In particular:

- We cannot choose the redex \((\lambda x. x) y\) in any of the steps because it is under a \(\lambda\) (the \(\lambda x y z\)), meaning that all three arguments, \(x\), \(y\), and \(z\) have to be instantiated first before we are allowed to choose it.
- We cannot choose the redex \((\lambda x. x) y\) in any of the steps because it is morally an argument of the function \(\lambda x y z. x ((\lambda x. x) y)\), and call-by-value does not allow us to reduce function arguments before the function call.
- The reduction stops because all the only available redex \(((\lambda x. x) ((\lambda x. x) y) w)\) is under a lambda \((\lambda z)\).

Note that, since we have specified only what call-by-value should not do, there is some freedom left to choose the available redexes. In particular, we could have chosen the redex \((\lambda x y z. x ((\lambda x. x) y) v)\) first, the redex \((\lambda x y z. x x) t\) second, etc. Any of these choices would be acceptable in the exam.

**Call-by-value.** One of the call-by-value reductions of the example term proceeds as follows, with the redex reduced in each step highlighted in yellow:

\[
1) \ (\lambda x. x) y \ x \ t \ u \ ((\lambda x. x) y) v \ ((\lambda x. x) w) \\
2) \ \rightarrow_\beta \ (\lambda y. t y t) u \ ((\lambda x. x) y) v \ ((\lambda x. x) w) \\
3) \ \rightarrow_\beta \ (t u t) ((\lambda x. x) y) v \ ((\lambda x. x) w) \\
4) \ \rightarrow_\beta \ (t u t) ((\lambda y z. v ((\lambda x. x) y)) ((\lambda x. x) w)) \\
5) \ \rightarrow_\beta \ (t u t) ((\lambda y z. v ((\lambda x. x) y)) (w y)) \\
6) \ (t u t) ((\lambda z. v ((\lambda x. x) y)) (w y)) \\
\]

The only difference between call-by-name and call-by-value is that we have to reduce the function argument \((\lambda x. x) y\) before passing it into the function body—note that it has been reduced only once this time. The reduction stops for the same reason as call-by-name, as the only redex available is under a \(\lambda\).
3 EXPRESSIVITY AND ENCODINGS

The $\lambda$ calculus is quite expressive: it can be used to encode standard data structures and their associated operations, as well as recursion. In fact, the Church-Turing thesis states that register machines, Turing machines, and the $\lambda$-calculus have the same expressivity, and all capture the intuitive notion of computability. For the $\lambda$-calculus, this notion is called $\lambda$-definability, and is is formulated as follows.

Definition 3.1 ($\lambda$-definability). A partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is $\lambda$-definable if and only if there exists a closed term $M$ with the following properties:

$$f(x_1, \ldots, x_n) = y \iff M \overrightarrow{x_1 \ldots x_n} = \beta y$$

and

$$f(x_1, \ldots, x_n) \uparrow \iff M \overrightarrow{x_1 \ldots x_n} \text{ has no normal form}$$

Exam: Recursion will not be examinable this year.

Here, we will show how to encode various data structures in the $\lambda$-calculus, in particular focussing on pairs, lists, and natural numbers. In the slides, we also cover booleans and binary trees.

Exam: For a given data structure, you are expected to be able to:

- Understand its definition
- Encode its values
- Encode its constructors
- Create its recognisers
- Encode basic operations on the data structure

Data Structure Definitions. We will define data structures using the fully formal notation in the style of the proof assistant Coq. Below, from left to right, we give the definitions of pairs, lists, and natural numbers.

\[
\begin{align*}
\text{Inductive } \text{Pair} : \text{Set} &= \text{Inductive } \text{List} : \text{Set} = \text{Inductive } \text{Nat} : \text{Set} = \\
| & \text{pair} : \text{elt} \rightarrow \text{elt} \rightarrow \text{Pair} | \text{nil} : \text{List} | \text{zero} : \text{Nat} \\
| & \text{cons} : \text{elt} \rightarrow \text{List} \rightarrow \text{List} | \text{succ} : \text{Nat} \rightarrow \text{Nat}
\end{align*}
\]

Data structure definitions start with the keyword Inductive, followed by the type of the data structure (here it's Pair/List/Nat), and the sort in which the data structure lives (normally Set). You can safely ignore the meaning of Set for now. What comes next is a series of constructors, which describe the different ways in which the values of the data structure can be created. Every constructor is associated with a name and a type. Let's take a closer look at the constructors of pairs, lists, and natural numbers:

[Pairs] In order to construct a pair $(a, b)$, we only need its two constituents; this means that pairs have a single constructor:

- $\text{pair} : \text{elt} \rightarrow \text{elt} \rightarrow \text{Pair}$, which takes two elements, $a$ and $b$, and returns the pair $\text{pair } a \ b$, which represents the mathematical pair $(a, b)$.

[Lists] A list can either be an empty list or a concatenation of an element with another list; this means that we will have two constructors:

- $\text{nil} : \text{List}$, which in itself is a list, representing the empty list; and
- $\text{cons} : \text{elt} \rightarrow \text{List} \rightarrow \text{List}$, which takes an element $e$, a list $l$ and returns another list, $\text{cons } e \ l$, which represents the list obtained by prepending $e$ to $l$. 
Using this formal definition, for example, the empty list corresponds to \texttt{nil}, a single-element list \([a]\) corresponds to \texttt{cons} \(a\ \texttt{nil}\), and a three-element list \([a, b, c]\) corresponds to \texttt{cons} \(a\ (\texttt{cons} \ b\ (\texttt{cons} \ c\ \texttt{nil}))\).

\textbf{[Nats]} A natural number can either be a zero or a successor of another natural number; this means that, similarly to lists, we will also have two constructors:

- \texttt{zero} : \texttt{Nat}, which in itself is a natural number, representing 0; and
- \texttt{succ} : \texttt{Nat} \to \texttt{Nat}, which takes a natural number \(n\) and returns another natural number, \texttt{succ} \(n\), which represents the natural number \(n + 1\).

Using this formal definition, for example, 0 corresponds to \texttt{zero}, 3 corresponds to \texttt{succ} \texttt{zero}, and 3 corresponds to \texttt{succ} \texttt{(succ} \texttt{zero)}).

Note that, in the encodings of pairs and lists, we use a generic type \texttt{elt} to denote the type of their elements. If we wanted, for example, to have pairs of natural numbers or lists that contain natural numbers, we would, in the definition, replace the \texttt{elt} type with \texttt{Nat}.

### 3.1 Encoding Values

The recipe for encoding values of a given data structure is as follows:

1. All values are encoded as abstractions, with one binder per data-structure constructor.
2. The abstraction body corresponds to the formal definition of the value.
3. Any non-inductive parameters have to be encoded themselves.

Let’s see how this works in practice by encoding, for example, the pair \((a, b)\). According to step (1), this encoding will be an abstraction. Moreover, since pairs only have one constructor \((\texttt{pair})\), the abstraction will have only one binder, corresponding to that constructor; to avoid clutter, we will shorten it to \(p\). Therefore, the provisional encoding is of the form:

\[(a, b) = \lambda p. ?\]

The formal definition of this pair, as seen above, is \texttt{pair} \(a\ \texttt{b}\). The abstraction body, according to step (2), will then be first constructed as \(p\ a\ b\). Therefore, after the second step, the provisional encoding is:

\[(a, b) = \lambda p. p\ a\ b\]

Finally, since the \texttt{pair} constructor has no inductive parameters, according to step (3), both \(a\) and \(b\) have to be encoded themselves, meaning that the encoding of the pair \((a, b)\) is

\[(a, b) = \lambda p. p\ a\ b\]

\underline{Exam: In the exam, you only need to produce the final encoding. You do not have to explain the intermediate steps.}

As a second example, we will encode the list \([a, b]\). In contrast to pairs, we will need two binders since lists have two constructors, \texttt{nil} and \texttt{cons} (which we will shorten to \(n\) and \(c\)). Therefore, the first provisional encoding is:

\[[a, b] = \lambda cn. ?\]

The order of constructors could have been the other way around, or any combination thereof if we had more than two constructors; what is important is that once we’ve chosen the order, we need to maintain it throughout.

The formal definition of this list is \texttt{cons} \(a\ (\texttt{cons} \ b\ \texttt{nil})\). The abstraction body, according to step (2), will then be first constructed as \(c\ a\ (c\ b\ n)\). Therefore, after the second step, the provisional encoding is:

\[[a, b] = \lambda cn. c\ a\ (c\ b\ n)\]
Finally, as the first parameter of the \texttt{cons : elt -> List -> List} constructor is not inductive (but the second one is, as its type is \texttt{List}), we have to encode \texttt{a} and \texttt{b} according to step (3), meaning that the final encoding of the list \([a, b]\) is:

\[
[a, b] = \lambda cn. a (c b n)
\]

As an exercise, you can check for yourself that, according to the given instructions, the encoding of a given natural number \(n\) are of the form:

\[
n = \lambda sz. s \ldots (s z) \ldots
\]

\subsection{3.2 Encoding Constructors}

The encoding of constructors is more involved than that of values. The recipe is as follows:

1. All constructors are encoded as abstractions, with one binder per data-structure constructor.
2. All of the parameters of the constructor are prepended as additional binders.
3. The abstraction body follows the constructor type, with all inductive parameters unpacked.

As a first example, let us encode the \texttt{pair : elt -> elt -> Pair} constructor of pairs. According to step (1), the encoding starts off the same as for values:

\[
\text{pair} = \lambda p.? 
\]

According to step (2), the parameters of the constructor (there are two of them, they can be named arbitrarily, we will name them \(a\) and \(b\)) should be prepended to the current encoding:

\[
\text{pair} = \lambda abp.? 
\]

Finally, according to step (3), the abstraction body needs to follow the structure of the constructor:

\[
\text{pair} = \lambda abp. a b
\]

Since there are no inductive parameters, this is also the final encoding of the pair constructor.

\textbf{Exam:} In the exam, you only need to produce the final encoding. You do not have to explain the intermediate steps.

Let us check that this encoded constructor actually works: that is, let us apply any two parameters to it and \(\beta\)-reduce the resulting term to its normal form:

\[
(\lambda abp. a b) 1 2 \rightarrow_\beta (\lambda bp. 1 b) 2

\rightarrow_\beta \lambda p. p 1 2
\]

As expected, the obtained \(\lambda\)-term is the encoding of the appropriate pair, \((1, 2)\), which means that the \texttt{pair} constructor has been encoded correctly.

As a second example, let us encode the two list constructors, \texttt{nil} and \texttt{cons}. As the \texttt{nil} constructor has no parameters, meaning that it is, in itself, a value, we can immediately give its encoding as:

\[
\text{nil} = \lambda cn.n 
\]

The \texttt{cons : elt -> List -> List} constructor will make use of all aspects given in the recipe. According to step (1), the encoding starts off the same as for values:

\[
\text{cons} = \lambda cn.? 
\]

According to step (2), the parameters of the constructor (there are two of them, they can be named arbitrarily, we will name them \(e\) and \(l\)) should be prepended to the current encoding:

\[
\text{cons} = \lambda elcn.? 
\]
According to step (3), the abstraction body needs to follow the structure of the constructor:
\[ \text{cons} = \lambda e l c n. e \]
Finally, since the second parameter, \( l \), of the \text{cons} constructor is inductive, it needs to be unpacked in the encoding, as per step (3). Unpacking a \( \lambda \)-term means removing its binders. In this case, we know that \( l \), since it is an encoding of a list, is of the form \( \lambda c n. ? \), and we will remove these two binders by instantiating them with the corresponding \( c \) and \( n \) of the encoding, resulting in the final encoding of the \text{cons} constructor:
\[ \text{cons} = \lambda e l c n. e (l c n) \]

Now, let us make sure that the encoded \text{cons} constructor works. We will do this by applying to it the encodings of 1 and the list \([1, 2]\), and reducing the obtained term to its normal form:
\[
\begin{align*}
    \left( \lambda e l c n. e (l c n) \right) \lambda c n. 1 \rightarrow & \beta \left( \lambda e l c n. e (l c n) \right) \left( \lambda c n. 1 \right) \\
    & \rightarrow \beta \lambda c n. 1 \left( \left( \lambda c n. 1 \right) c n \right) \\
    & \rightarrow \beta \lambda c n. 1 \left( \left( \lambda n. c 2 n \right) n \right) \\
    & \rightarrow \beta \lambda c n. 1 \left( c 2 n \right)
\end{align*}
\]

obtaining the encoding of the list \([1, 2]\), as hoped for.

As an exercise, you can check for yourself that, according to the given instructions, the encodings of the \text{zero} and \text{succ} constructors for natural numbers are of the following form:
\[
\begin{align*}
    \text{zero} &= \lambda s z. z \\
    \text{succ} &= \lambda n s z. (n s z)
\end{align*}
\]

### 3.3 Creating Recognisers

Recognisers provide a mechanism for recognising the lead constructor of a given value of a given data structure. They can then be used to encode operations on the data structure, as we will see in the next subsection.

We illustrate how to create recognisers for list constructors, starting from the recogniser for \text{nil}. The recogniser for \text{nil} effectively needs to reduce to the encoding of \text{true} if the list passed to it is an empty list, and to the encoding of \text{false} otherwise.

The methodology is to start from an arbitrary list and then substitute its constructors for functions that will do the required task. In particular, for the \text{nil} recogniser, we start from:
\[
R_{\text{nil}} = \lambda l. ?
\]
We know that the list \( l \) is either of the form \( \lambda c n. n \) or \( \lambda c n. e l' \). Note that, in order to be able to reduce to either \text{true} or \text{false} from these terms, we would need to apply two terms to them, with the first corresponding to the case of the \text{cons} constructor (\( e \)) and the second corresponding to the case of the \text{nil} constructor (\( n \)):
\[
R_{\text{nil}} = \lambda l. ?_c ?_n
\]
For the \text{nil}-related term, we can simply put \( ?_2 = \text{true} \), obtaining:
\[
R_{\text{nil} \ n} \equiv \left( \lambda l. ?_c \text{true} \right) \left( \lambda c n. n \right)
\]
\[
\rightarrow \beta \left( \lambda c n. n \right) ?_c \text{true}
\]
\[
\rightarrow \beta \left( \lambda n. n \right) \text{true}
\]
\[
\rightarrow \beta \text{true}
\]
We could try, similarly, to put \(?_c = false\) for the cons-related term. This, however, would result in the following:

\[
R_{n11} (\text{cons } e' l') ≡ (\lambda l.l (\text{false}) (\lambda c \text{n}. e (l' c \text{n})))
\]

\[
→ _\beta (\lambda c \text{n}. e (l' c \text{n})) \text{false true}
\]

\[
→ _\beta (\lambda n (\text{false}) e (l' \text{false } n)) \text{true}
\]

\[
→ _\beta (\text{false} e (l' \text{false } true))
\]

which does not necessarily further reduce to false, and the problem is caused by the surviving e and l' false true. The solution is to consume these two terms by adapting ?c as follows:

\[
?_c = \lambda e.l.\text{false}
\]

resulting in the following reduction:

\[
R_{n11} (\text{cons } e' l') ≡ (\lambda l.l (\lambda e.l.\text{false}) (\text{true}) (\lambda c \text{n}. e (l' c \text{n})))
\]

\[
→ _\beta (\lambda c \text{n}. e (l' c \text{n})) (\lambda e.l.\text{false}) \text{true}
\]

\[
→ _\beta (\lambda n (\lambda e.l.\text{false}) e (l' (\lambda e.l.\text{false}) n)) \text{true}
\]

\[
→ _\beta (\lambda e.l.\text{false}) e (l' (\lambda e.l.\text{false}) \text{true})
\]

\[
→ _\beta (\lambda e.l.\text{false}) \text{true}
\]

Finally, we have that the n1 recogniser is:

\[
R_{n11} = \lambda l.l (\lambda e.l.\text{false}) \text{true}
\]

and the cons recogniser, analogously, is:

\[
R_{\text{cons}} = \lambda l.l (\lambda e.l.\text{true}) \text{false}
\]

You can check, as an exercise, that \(R_{\text{cons}}\) behaves as intended.

**Exam:** In the exam, you only need to produce the recogniser directly. You do not have to explain the intermediate steps.

In general, if we had a data structure \(d\) with constructors \(c_i|_{i=1}^n\), and each constructor takes \(k_i|_{i=1}^n\) parameters, and we wanted to create the recogniser for the \(m\)-th constructor, given how we have chosen to do encodings, that term will be of the form:

\[
\lambda d.\ D (\lambda x_1, \ldots x_{k_1}, \text{false}) \ldots (\lambda x_1, \ldots, x_{k_m}, \text{true}) \ldots (\lambda x_1, \ldots x_{k_n}, \text{false})
\]

The key points that you need to remember are:

1. Always start from \(\lambda d.\ D ?\)
2. Remember that the encodings of any value of this data structure will be of the form \(\lambda c_1 \ldots c_n.\ D\), and that the constructors are ordered. This means that the \(d\) from point (1) will need to have \(n\) terms applied to it, with the \(i\)-th term corresponding to the \(i\)-th constructor.
3. For each constructor, you have to consume all the parameters it takes (that is done via the \(\lambda x_1, \ldots, x_{k_i}\)) and then you need to put true for the one you want to recognise and false for the others.
3.4 Encoding Basic Operations

We demonstrate how to use the provided encodings to express the \( \text{fst} \) and \( \text{snd} \) operator on pairs, the \( \text{hd} \) and \( \text{tl} \) operators on lists, and the addition and multiplication operators of natural numbers. As for the recognisers, the methodology is to insert the appropriate terms in place of the constructors and the challenge is to understand what these terms are.

\( \text{fst} \) on pairs. The \( \text{fst} \) operator is unary, with \( \text{fst} \; p \) returning the first component of the pair. As we have seen previously, a pair \((a, b)\) is encoded in the \( \lambda \)-calculus as the term \( \lambda p. p \, a \, b \). The aim is to find a \( \lambda \)-term \( t \) such that, if we were to substitute \( t \) for \( p \) in the body of the encoding, we would be able to reduce the obtained term to \( a \). This term would have to accept two parameters and return the first, and we have already seen this term as the encoding of \( \text{true} \) in the slides: \( t = \lambda ab.a \). Therefore, we have:

\[
\text{fst} = \lambda p. (\lambda ab.a)
\]

To convince ourselves of the correctness of \( \text{fst} \), let us apply it to the encoding of the pair \((1, 2)\):

\[
\text{fst} (\lambda p. p \, 1 \, 2) \equiv (\lambda p. p (\lambda ab.a)) (\lambda p. p \, 1 \, 2) \\
\rightarrow \beta (\lambda p. p \, 1 \, 2) (\lambda ab.a) \\
\rightarrow \beta (\lambda ab.a) \, 1 \, 2 \\
\rightarrow \beta (\lambda b.1) \, 2 \\
\rightarrow \beta 1
\]

\( \text{snd} \) on pairs. Analogously to \( \text{fst} \), the \( \text{snd} \) operator, which takes a pair and returns its second component, is shown to correspond to the following term:

\[
\text{snd} = \lambda p. (\lambda ab.b)
\]

\( \text{hd} \) on lists. The \( \text{hd} \) operator is unary, with \( \text{hd} \; l \) returning the first element of a given list. It is also not defined on all lists, but only on non-empty ones. For this encoding, we will start by adapting the previously seen \( \text{cons} \) recogniser:

\[
R_{\text{cons}} = \lambda l.1 \, (\lambda el. \text{true}) \, \text{false}
\]

to not return \( \text{true} \) when passed a non-empty list, but, instead, the first parameter of the \( \text{cons} \) constructor, \( e \):

\[
\text{hd'} = \lambda l.1 \, (\lambda el.e) \, \text{false}
\]

To convince ourselves of the correctness of \( \text{hd'} \) when given a non-empty list, let us apply it to the encoding of the list \([1, 2]\):

\[
\text{hd'} (\lambda cn. c \, 1 \, (c \, 2 \, n)) \equiv (\lambda l.1 \, (\lambda el.e) \, \text{false}) \, (\lambda cn. c \, 1 \, (c \, 2 \, n)) \\
\rightarrow \beta (\lambda cn. c \, 1 \, (c \, 2 \, n)) \, (\lambda el.e) \, \text{false} \\
\rightarrow \beta (\lambda n.(\lambda el.e) \, 1 \, ((\lambda el.e) \, 2 \, n)) \, \text{false} \\
\rightarrow \beta (\lambda el.e) \, 1 \, ((\lambda el.e) \, 2 \, \text{false}) \\
\rightarrow \beta (\lambda l.1) \, ((\lambda el.e) \, 2 \, \text{false}) \\
\rightarrow \beta 1
\]
Let us take a look now at what happens when we apply the empty list to \( \text{hd}' \), an operation that is not mathematically defined:

\[
\text{hd}' (\lambda cn. n) \equiv (\lambda l. (\lambda e. \text{false}) (\lambda n. n)) \\
\rightarrow_\beta (\lambda cn. n) (\lambda e. \text{false}) \\
\rightarrow_\beta (\lambda n. \text{false}) \\
\rightarrow_\beta \text{false}
\]

which does not quite correspond to what one would intuitively expect. Given the definition of \( \lambda \)-definability, a better choice for \( \text{hd} \) would be:

\[
\text{hd} = \lambda l. (\lambda e. \text{false}) D
\]

where \( D \) is the divergent term \( (\lambda x. x) (\lambda x. x) \). This would ensure that \( \text{hd} (\lambda cn. n) \) has no normal form, which corresponds to the fact that \( \text{hd} \) is not defined on the empty list.

\( t_1 \) on lists. The list tail function is where we run into the limitations of our encoding approach. You can check that if we tried to define \( t_1 \) analogously to \( \text{hd} \), this would not work because the \( c \) and \( n \) constructors that are in the tail would get substituted and the structure of the term would be lost. It is possible to encode \( t_1 \), but the encoding is complex:

\[
t_1 = \lambda l. \text{fst} (l (\lambda ab. \text{pair} (\text{snd} b) (\text{cons} a (\text{snd} b))) (\text{pair} \text{nil} \text{nil}))
\]

and is beyond the level required for the exam. For instructive purposes, we will demonstrate that this encoding works by using it to retrieve the tail of the list \( [1, 2, 3] \). Let

\[
P = \lambda ab. \text{pair} (\text{snd} b) (\text{cons} a (\text{snd} b))
\]

Then:

\[
t_1 (\lambda cn. c \ 1 (c \ 2 (c \ 3 \ n))) \\
\equiv (\lambda l. \text{fst} (l P (\text{pair} \text{nil} \text{nil}))) (\lambda cn. c \ 1 (c \ 2 (c \ 3 \ n))) \\
\rightarrow_\beta \text{fst} ((\lambda cn. c \ 1 (c \ 2 (c \ 3 \ n))) P (\text{pair} \text{nil} \text{nil})) \\
\rightarrow_\beta \text{fst} (P \ 1 (P \ 2 (\text{pair} \text{nil} \text{nil}))) \\
= \beta \text{fst} (P \ 1 (P \ 2 ((\lambda ab. \text{pair} (\text{snd} b) (\text{cons} a (\text{snd} b))) \ 3 \ (\text{nil}, \text{nil}))) \\
\rightarrow_\beta \text{fst} (P \ 1 (P \ 2 (\text{pair} \text{nil} (\text{cons} \text{nil} \text{nil})))) \\
= \beta \text{fst} (P \ 1 (\text{pair} (\text{snd} b) (\text{cons} a (\text{snd} b))) \ 2 \ (\text{nil}, [3])) \\
\rightarrow_\beta \text{fst} (P \ 1 (\text{pair} [3] (\text{cons} 2 [3]))) \\
= \beta \text{fst} (\text{pair} [2, 3] (\text{cons} 1 [2, 3])) \\
\rightarrow_\beta [2, 3]
\]

Addition of natural numbers. For addition, we are required to, given \( m = \lambda sz. s \ (\ldots (s \ z) \ldots) \), and \( n = \lambda sz. s \ (\ldots (s \ z) \ldots) \), produce \( m + n = \lambda sz. s \ (\ldots (s \ z) \ldots) \). This amounts to substituting the \( z \) in the abstraction body of \( m \) with the abstraction body of \( n \) (obtained by \( n \ s \ z \)):

\[
\lambda mnsz. m \ (n \ s \ z)
\]
Multiplication of natural numbers. For multiplication, we are required to, given $m = \lambda sz. s (\ldots (s z) \ldots )$, and $n = \lambda sz. s (\ldots (s z) \ldots )$, produce $m \cdot n = \lambda sz. s (\ldots (s z) \ldots )$. The principle is similar to encoding addition; the goal is to somehow substitute the $s$ (of which there are $m$) in the abstraction body of $m$ with some modification of $n$; this way we will get $m \cdot n$ applications of $s$. Therefore, we start from:

$$z = \lambda mnsz. m \ ? z$$

The first attempt could be to try $? = n s z$, like for addition, but this will not work:

$$m \ (n s z) z \equiv (\lambda s. \lambda z. s (\ldots (s z) \ldots )) (n s z) z = (\lambda s. \lambda z. s (\ldots (s z) \ldots )) (s^n z) z$$

and the inside $z$ cannot be propagated further. The trick to propagate the $z$ is to evaluate $n$ only partially, leaving the $\lambda z$: $? = n s$:

$$m \ (n s z) z = (\lambda s. \lambda z. s (\ldots (s z) \ldots )) (n s z) z$$

$$= (\lambda s. \lambda z. s (\ldots (s z) \ldots )) (\lambda z. s^n z) z$$

$$= \beta ((\lambda z. s^n z) \ldots ((\lambda z. s^n z) z) \ldots )$$

$$\rightarrow (\lambda z. s^n z) \ldots ((\lambda z. s^n z)(s^n z)) \ldots )$$

$$\rightarrow (\lambda z. s^n z) \ldots ((\lambda z. s^n z)(s^{2n} z)) \ldots )$$

Therefore, we have that natural number multiplication can be encoded as:

$$z = \lambda mnsz. m \ (n s) z$$