CHAPTER 4

THE CURRY-HOWARD CORRESPONDENCE

1 INTRODUCTION

In this chapter we study a remarkable correspondence between two independently defined systems: intuitionistic natural deduction and the typed $\lambda$-calculus. This correspondence is known as the Curry-Howard isomorphism and establishes a precise relation between logic and (functional) computation.

The $\lambda$-calculus can be regarded as the theoretical foundation of functional programming. Indeed, it is the canonical form of the pure fragment of such languages. It is a system consisting of functional abstraction and application, which are two universal features in programming languages: abstraction is the mechanism that corresponds to procedure definitions, and application corresponds to a procedure call. The $\lambda$-calculus has also contributed to the design of modern languages, for example the notion of polymorphism that we find presently in many languages was first developed for the $\lambda$-calculus. Also, the $\lambda$-calculus is used as the meta-language for defining the semantics (specifically denotational) of different kinds of programming languages.

The Curry-Howard isomorphism establishes a tight relationship between the typed $\lambda$-calculus (a restricted form of the $\lambda$-calculus) and intuitionistic propositional logic. The isomorphism can be explained by considering the term formation rules for the typed $\lambda$-calculus which can be given as a natural deduction system. We write $t : \sigma$ to represent that a term $t$ of the $\lambda$-calculus has type $\sigma$. There are three term constructions in the $\lambda$-calculus, which are variables (denoted by $x, y, z, \ldots$), abstraction (denoted by $\lambda x.t$) and application (denoted by juxtaposition of terms, i.e. $tu$). The rules for building terms are given below:

\[
\begin{align*}
\text{x : } \sigma & \rightarrow x : \sigma \\
\ \Gamma, x : \sigma & \rightarrow t : \tau \\
\Gamma & \rightarrow (\lambda x.t) : \sigma \rightarrow \tau \\
\Gamma & \rightarrow t : \sigma \rightarrow \tau \\
\Delta & \rightarrow u : \sigma \\
\Gamma, \Delta & \rightarrow (tu) : \tau
\end{align*}
\]

If the terms are taken out of the above presentation we are left with the following, which, if we replace the functional arrow ($\rightarrow$) by implication ($\Rightarrow$) and types by
formulas, is the natural deduction presentation of (the implicational fragment of) intuitionistic logic:

\[
\frac{\Gamma \vdash \Phi}{\Phi \to \Phi} \\
\frac{\Gamma, \Phi \to \Phi'}{\Gamma \to \Phi \to \Phi'} \\
\frac{\Gamma \to \Phi \Rightarrow \Phi'}{\Gamma, \Delta \to \Phi'}
\]

These rules are called Axiom \((Ax)\), implication introduction \((\Rightarrow I)\) and implication elimination \((\Rightarrow E)\) respectively. From this presentation one sees immediately that there is a correspondence between types of the \(\lambda\)-calculus and formulae of intuitionistic logic: all we do is systematically replace each type \(\sigma, \tau\) by formulae; here \(\Phi, \Phi'\). Slightly more hidden is the fact that there is a correspondence between the terms of the calculus and the proofs of the logic: variables correspond to the rule \((Ax)\), abstraction corresponds to the \((\Rightarrow I)\) rule and application corresponds to the \((\Rightarrow E)\) rule. Moreover, an analysis of the process of normalisation in intuitionistic logic and the normalisation process of the \(\lambda\)-calculus yields a further correspondence: the process of \(\beta\)-reduction in the \(\lambda\)-calculus corresponds to the normalisation procedure in logic.

If we put all this together, taking programs to be terms from the \(\lambda\)-calculus, the Curry-Howard isomorphism can be seen as:

- programs \(\sim\) proofs
- types \(\sim\) formulae
- computation \(\sim\) normalisation

and is known under a series of other names including \textit{formulae-as-types} and \textit{proofs-as-programs}.

The most interesting aspect of the correspondence is the relationship between normalisation and computation. This gives a dynamical aspect of logic, and a logical side to operational semantics of programming languages. The correspondence permits results to be carried over from one framework to the other, for example normalisation theorems.

This kind of correspondence is by no means restricted to intuitionistic propositional logic and the typed \(\lambda\)-calculus. One of the most well-known examples of this correspondence is the term calculus for second order propositional intuitionistic logic, known as System \(F\). Second order propositional intuitionistic logic extends the system \(NJ_0\) with the following:

\[
\frac{\Gamma \to \Phi}{\Gamma \to \forall X.\Phi} \quad \frac{\Gamma \to \forall X.\Phi}{\frac{\Phi[\Phi'/X]}{\Gamma \to \Phi}} (\forall I) (\forall E)
\]

where the \((\forall I)\) rule has the side condition that \(X\) is not free in \(\Gamma\). Later we shall see that we can annotate derivations with terms giving a calculus for this logic. Results
obtained in the calculus, for example strong normalisation, confluence and consistency, can then be applied directly to the logic, once the isomorphism is established.

Throughout this chapter all the logical systems will be given using the multiplicative presentations (cf. Chapter 3, Section 6), but the results are by no means restricted to this presentation.

1.1 Functions

We are all familiar with the notion of a function which is an input/output relation. For example we can construct functions using names such as:

\[ f \quad : \quad \text{IN} \to \text{IN} \]
\[ f(x) = x + 1 \]

where \( f : \text{IN} \to \text{IN} \) states that the function \( f \) has the set of natural numbers as domain and codomain, and \( f(x) = x + 1 \) gives the algorithm to compute the value of the function for each element of \( \text{IN} \). In this terminology we can write integer expressions (e.g. \( x + 1 \)) as functions as above, but there is a discrepancy between the status of functions and expressions. There is no reason at all why we shouldn’t consider functions themselves as expressions, i.e. make them first class citizens. To facilitate this we use the \( \lambda \)-notation — a way of writing functions as first class data objects anonymously (without having to resort to giving it a name, such as \( f \) as above). To motivate this notation, we consider some examples:

\[ \text{succ} = \lambda x. x + 1 \]
\[ \text{apply} = \lambda f. \lambda x. fx \]

Where we think of \( \lambda x. t \) as a notation for:

\[ \text{function (x)} \]
\[ t \]

In other words, \( \lambda x. e \) is a function that returns a value \( t \) depending on the argument \( x \). For the examples above, \( \text{succ} \) is quite obvious in that it is a function which takes an argument, and returns the successor. The function \( \text{apply} \) is slightly more complicated in that it is a function that takes two arguments; the first is a function and the second is an argument for that function, and the result is the application of that function to the argument. Understanding that a function can take a function as an argument is a fundamental aspect of the \( \lambda \)-calculus. Before giving a formal definition of the calculus, let us see how we can use this notation. There is an evident notion of function application:

\[ (\lambda x. t)u \]

Note that in this notation it is more common to have the parentheses around the function rather than the argument. Function application is simply substituting the
occurrence of the variable \( x \) in the term \( t \) with the term \( u \). (Compare with \( f(x) = x + 1 \) and the application \( f(3) \) for example.) Hence we have a rewriting rule that we write like:

\[
(\lambda x.t)u \rightarrow t[u/x]
\]

where \( t[u/x] \) is the notation for substitution that we will formalise shortly. Here is an example in a familiar setting:

\[
(\lambda x.x + 1)3 \rightarrow (x + 1)[3/x] = 3 + 1
\]

which can then be reduced to the final result 4.

To show how functions themselves can be used as arguments, here is the application of \((\text{apply succ})0\) in this notation. To shorten the trace of the computation, we assume that substitutions are done immediately.

\[
(\lambda f.\lambda x.fx)(\lambda x.x + 1)0 \rightarrow (\lambda x.(\lambda x.x + 1)x)0
\]

\[
\rightarrow (\lambda x.x + 1)0
\]

\[
\rightarrow 0 + 1
\]

which again can be reduced to give 1 as the answer.

When using this notation, one has to take great care in understanding the scope of a variable. In the example above, we see that we have the variable \( x \) appearing several times in the term \((\lambda x.(\lambda x.x + 1)x)0\). Variables are either bound by a \( \lambda \), or they are free. In the example, all variables are bound, but if we look at the subterm \((\lambda x.x + 1)x\) there are two occurrences of \( x \) of which one is free, and the other (underlined) is bound by the \( \lambda \). To overcome this confusion, we will adopt a convention that all free variables are named differently from bound variables.

We refer the reader to other texts for a more thorough introduction into the \( \lambda \)-calculus, see for example Hankin (1994) and Barendregt (1984; 1992).

2 TYPED \( \lambda \)-CALCULUS AND NATURAL DEDUCTION

Here we define the theory of the typed \( \lambda \)-calculus without recourse to the corresponding system of natural deduction. In the following subsection we will make the correspondence precise.

2.1 Typed \( \lambda \)-calculus

Definition 4.1 We define a set of types \( T \) inductively as:

1. A collection of type variables: \( \alpha, \beta, \gamma, \ldots \)

2. If \( \sigma \) and \( \tau \) are types, then:
(a) \((\sigma \times \tau)\) is a (product) type.

(b) \((\sigma \rightarrow \tau)\) is a (function) type.

The set of typed \(\lambda\)-terms are generated from the following. We write \(t : \sigma\) to mean that term \(t\) has type \(\sigma\).

1. A collection of typed variables \(x : \sigma, \ldots\)

2. If \(u : \sigma\) and \(v : \tau\) are typed terms, then the product \(\langle u, v \rangle : \sigma \times \tau\) is a typed term.

3. If \(t : \sigma \times \tau\) is a typed term, then:

\[
\begin{align*}
(a) \ & \text{fst}(t) : \sigma \\
(b) \ & \text{snd}(t) : \tau
\end{align*}
\]

are typed terms.

4. If \(v : \tau\) is a term and \(x : \sigma\) is a variable, then the abstraction \(\lambda x^\sigma . v : \sigma \rightarrow \tau\) is a typed term.

5. If \(t : \sigma \rightarrow \tau\) and \(u : \sigma\) are terms, then the application \((tu) : \tau\) is also a typed term.

There is also a system of \(\lambda\)-calculus without types, called the untyped \(\lambda\)-calculus, or simply the \(\lambda\)-calculus. The untyped version is exactly the same as the calculus that we have presented here, except that more terms are admitted since the term construction does not depend on the types.

When referring to the typed \(\lambda\)-calculus we shall often write abstractions \(\lambda x^\sigma . t : \sigma \rightarrow \tau\) as just \(\lambda x . t\) when we are not so interested in the type, and similarly for other terms.

There are several accepted syntactic conventions for both types and typed terms. For types, we assume that \(\rightarrow\) associates to the right, and binds more strongly than \(\times\), and outermost parentheses can be dropped. These are exactly the same conventions that we used for logical connectives \(\Rightarrow\) and \(\land\). For typed terms, we adopt the following conventions:

- Application associates to the left and we drop outermost parentheses, for example:

\[
\begin{align*}
\ ((xy)z) \ & \text{becomes} \ xyz \\
\ (x(yz)) \ & \text{becomes} \ x(yz)
\end{align*}
\]

- Multiple \(\lambda\)'s can be abbreviated, for example:

\(\lambda x . \lambda y . \lambda z . t\) becomes \(\lambda xyz . t\)
Proof Theory and Automated Deduction
Goubault-Larrecq, J.; Mackie, I.
1997, 444 p., Softcover
ISBN: 978-1-4020-0368-4