Chapter 4

The Curry-Howard Correspondence

4.1 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:

$$ x : \sigma \rightarrow x : \sigma $$

$$ \frac{\Gamma, x : \sigma \rightarrow t : \tau}{\Gamma \rightarrow (\lambda x.t) : \sigma \rightarrow \tau} $$

$$ \Gamma \rightarrow t : \sigma \rightarrow \tau \quad \Delta \rightarrow u : \sigma $$

$$ \frac{\Gamma, \Delta \rightarrow (tu) : \tau}{\Gamma, \Delta \rightarrow u : \sigma} $$

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:

$$ \frac{\phi \rightarrow \phi}{\phi \rightarrow \phi} $$

$$ \frac{\Gamma, \phi \rightarrow \phi' \quad \Delta \rightarrow \phi}{\Gamma \rightarrow \phi \Rightarrow \phi'} $$

These rules are called Axiom ($A\phi$), 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 \((\forall x)\), 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 formulae-as-types and 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 \(NJ\) with the following:

\[
\frac{\Gamma \rightarrow \Phi}{\Gamma \rightarrow \forall X.\Phi} \quad (\forall I) \quad \frac{\Gamma \rightarrow \forall X.\Phi}{\Gamma \rightarrow \Phi[f/x]} \quad (\forall E)
\]

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 3.6), but the results are by no means restricted to this presentation.

### 4.1.1 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:

\[
\begin{align*}
f &: \mathbb{N} \rightarrow \mathbb{N} \\
f(x) &= x + 1
\end{align*}
\]

where \(f : \mathbb{N} \rightarrow \mathbb{N}\) 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 \(\mathbb{N}\). 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:

\[
\begin{align*}
succ &= \lambda x.x + 1 \\
apply &= \lambda f.\lambda x.fx
\end{align*}
\]

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.

$$(\lambda f.\lambda x.f x)(\lambda x.x + 1)0 \rightarrow (\lambda x.(\lambda x.x + 1)x)0 \rightarrow (\lambda x.x + 1)0 \rightarrow 0 + 1$$

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).

### 4.2 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.

### 4.2.1 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 λ-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:
   
   (a) \( \text{fst}(t) : \sigma \)
   
   (b) \( \text{snd}(t) : \tau \)

   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 λ-calculus without types, called the *untyped λ-calculus*, or simply the λ-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 λ-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:

  \[
  ((xy)z) \quad \text{becomes} \quad xyz
  \]

  \[
  (x(yz)) \quad \text{becomes} \quad x(yz)
  \]

- Multiple λ's can be abbreviated, for example:

  \[
  \lambda x. \lambda y. \lambda z. t \quad \text{becomes} \quad \lambda xyz.
  \]

- There is a notion of renaming that allows us to write the terms using different variable names. For example, \( \lambda x^\sigma . x \) and \( \lambda y^\sigma . y \) are obviously defining the same function. This renaming operation is known as \( \alpha \)-conversion (or \( \alpha \)-renaming) and is not at all trivial. For instance, in the term \( \lambda x. \lambda y . y \), a renaming of the variable \( x \) to \( y \) would give \( \lambda y. \lambda y . y \) which is not at all the same function that we originally had. We refer the interested reader to the literature for more details on this point, and just remark that one should take care with variable renamings.

Before going on, we give a few examples of typed λ-terms that the reader can easily check are well formed from the definition.

\[
\begin{align*}
\lambda x . x : \sigma \rightarrow \sigma \\
\lambda f . f x : (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau \\
\lambda x . \lambda y . x : \sigma \rightarrow \tau \rightarrow \sigma \\
\langle \lambda x . x , \lambda x . x \rangle : (\sigma \rightarrow \tau) \times (\tau \rightarrow \tau)
\end{align*}
\]

In contrast, the term \( \lambda x . xx \) does not have a type in the system given, since this would require that we have both \( x : \sigma \) and \( x : \sigma \rightarrow \sigma \) at the same time. The term \( \lambda x x : \sigma \times \tau \) is an example of a badly formed typed term, but remark that the term and the type are both well formed; they just don’t correspond to each other.

A λ-term is said to be closed (or a combinator) if there are no free variables in the term, otherwise it is open. For example \( \lambda x . y \) is an open term since the variable \( y \) is not bound to any \( \lambda \). In contrast, the term \( \lambda y . x . y \) is closed since all variables are bound. Formally, we define the free variables (FV) of a λ-term as follows:
where

\[ FV(x) = \{ x \} \]
\[ FV(\lambda x. t) = FV(t) \setminus \{ x \} \]
\[ FV(tu) = FV((t, u)) = FV(t) \cup FV(u) \]
\[ FV(fst(t)) = FV(snd(t)) = FV(t) \]

and the bound variables (BV) as:

\[ BV(x) = \emptyset \]
\[ BV(\lambda x. t) = BV(t) \cup \{ x \} \]
\[ BV(tu) = BV((t, u)) = BV(t) \cup BV(u) \]
\[ BV(fst(t)) = BV(snd(t)) = BV(t) \]

It is easy to see that the variables of a term are precisely \( FV(t) \cup BV(t) \).

**Definition 4.2 (Redex)** A redex (reducible expression) is a \( \lambda \)-term of one of the following forms: \( (\lambda x. t)u \), \( fst((t, u)) \), \( snd((t, u)) \).

The redex \( fst((t, u)) \) is a term constructed by building the pair \( (t, u) \) and then applying the projection \( fst \). It seems rather evident that having both \( t \) and \( u \) and then applying a projection function on the first element of the pair should just be the same term \( t \). Hence we would expect that we have the equality \( fst((t, u)) = t \).

Similarly, the redex \( (\lambda x. t)u \) has been constructed by building the function \( \lambda x. t \) (from the term \( t \)) and then building the application of this function with \( u \). This process (building a function and applying it) can be eliminated in the term, giving just the term \( t \) where the variable \( x \) has been replaced by the term \( u \). Again, we should expect the equality \( (\lambda x. t)u = t[u/x] \).

It is clear that the right-hand sides of the equalities are in some way simpler than the terms on the left. Formally, we have the following reduction rules for the typed \( \lambda \)-calculus, which define a reduction relation written as \( \rightarrow_\beta \) and called \( \beta \)-reduction.

\[
\begin{align*}
(\lambda x. t)u & \rightarrow t[u/x] \\
\text{fst}((t, u)) & \rightarrow t \\
\text{snd}((t, u)) & \rightarrow u
\end{align*}
\]

where substitution \( t[u/x] \) is defined as:

\[
\begin{align*}
x[v/x] & = v \\
y[v/x] & = y \\
(\lambda y. t)[v/x] & = \lambda y. (t[v/x]) \\
(tu)[v/x] & = (t[v/x])(u[v/x]) \\
\langle t, u \rangle[v/x] & = \langle t[v/x], u[v/x] \rangle \\
\text{fst}(t)[v/x] & = \text{fst}(t[v/x]) \\
\text{snd}(t)[v/x] & = \text{snd}(t[v/x])
\end{align*}
\]

We adopt a common convention, called the variable convention, that states that all bound variables are different from free variables, so that it is assumed that for the third case there are no free occurrences of the variable \( y \) in term \( t \) that could become bound. (Thus eliminating the problems of \( \alpha \)-conversion that we mentioned previously).

We also have the following reduction rules

\[
\begin{align*}
\lambda x. t & x \rightarrow t \ (x \notin FV(t)) \\
\langle \text{fst}(t), \text{snd}(t) \rangle & \rightarrow t
\end{align*}
\]

which we again can think of as a reduction from left to right. These reductions are the duals of the \( \beta \)-reductions and are called \( \eta \)-reductions, which we write as \( \rightarrow_\eta \). The notation \( \rightarrow_\beta \eta \) will be used for the union of \( \rightarrow_\beta \) and \( \rightarrow_\eta \), but we will often omit the subscripts and simply write \( \rightarrow \), where the meaning will be clear from the context. \( t \rightarrow u \) captures the fact that \( t \) rewrites to \( u \) in one step. The notation \( \rightarrow^* \) is used
for the transitive reflexive closure of $\rightarrow$, and $\rightarrow^+$ represents one or more steps of reduction. We can also define a notion of conversion, denoted by $\equiv_{\beta}$, to be the symmetric closure of $\rightarrow^*_\beta$. The reduction relation applies in any context, which we formalise as follows.

**Definition 4.3** A context $C[]$ is defined inductively by:

- A variable $x$ is a context.
- $[]$ is a context, called a hole.
- The application $C_1[.]C_2[.]$ is a context if both $C_1[.]$ and $C_2[.]$ are.
- The abstraction $\lambda x.C_1[.]$ is a context if $C_1[.]$ is a context.
- If $C_1[.]$ and $C_2[.]$ are contexts then so are $\text{fst}(C_1[.])$, $\text{snd}(C_1[.])$ and $\langle C_1[.], C_2[.] \rangle$.

If $t$ is a $\lambda$-term and $C[.]$ is a context, then $C[t]$ denotes the context $C[.]$ where all the holes are replaced by the term $t$.

We can now formally state what it means for a reduction to take place in any context:

**Lemma 4.4** If $t \rightarrow u$ then, for all contexts $C[.]$, $C[t] \rightarrow C[u]$.

**Proof:** A straightforward induction over the structure of contexts. 

Finally, a remark on substitutions. When writing terms with multiple substitutions, we have to be a little careful. The following lemma gives one result about how we can reorder them.

**Lemma 4.5 (Substitution)** If $x$ and $y$ are distinct variables and $x \notin FV(v)$, then

$$t[u/x][v/y] = t[v/y][u[v/y]/x]$$

**Proof:** The proof is a straightforward induction over the structure of $t$. See Exercise 4.1.

We refer the reader to the literature cited for a more thorough treatment of the $\lambda$-calculus, where the reader can find additional material on both the syntactic and semantic aspects of the calculus.

**EXERCISE 4.1**

Prove the Substitution Lemma 4.5: If $x$ and $y$ are distinct variables and $x \notin FV(v)$, then $t[u/x][v/y] = t[v/y][u[v/y]/x]$.

**EXERCISE 4.2**

We said that is was clear that the reductions make the terms simpler. However, give an example of a term such that reduction makes the term bigger. In what sense can it be said that reductions make the terms simpler?

### 4.2.2 Term Assignment for Natural Deduction

There are other ways that we could have presented the $\lambda$-calculus term formation rules. One of the most common is to present the rules as a deduction system where we derive sequents of the form:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \rightarrow t : \tau$$

where the $x_i$ are distinct variables, and $t$ is a $\lambda$-term that depends on the variables $x_i$. We can then define deduction rules that build terms of the right form. In Figure 4.1 we give such a set of rules. There are three kinds of rules to this system. The first is for the variable, which allows us to construct typed variables. The next group contains three structural rules which allow manipulations on the context: exchange ($X$), weakening ($W$) and contraction ($C$). Throughout this chapter we will work modulo the exchange rule. The final group, term formation rules, gives all the possible ways that we can construct terms. Each term
4.2. TYPED λ-CALCULUS AND NATURAL DEDUCTION

Variable:

\[
\frac{}{x : \sigma \rightarrow x : \sigma} \quad \text{(VAR)}
\]

Structural rules:

\[
\frac{\Gamma, x : \sigma, y : \tau, \Delta \rightarrow t : \gamma}{\Gamma, y : \tau, x : \sigma, \Delta \rightarrow t : \gamma} \quad \text{(X)}
\]

\[
\frac{\Gamma \rightarrow t : \gamma}{\Gamma, x : \sigma \rightarrow t : \gamma} \quad \text{(W)}
\]

\[
\frac{\Gamma, x : \sigma, y : \sigma \rightarrow t : \gamma}{\Gamma, z : \sigma \rightarrow t[z/x, z/y] : \gamma} \quad \text{(C)}
\]

Term formation rules:

\[
\frac{\Gamma \rightarrow t : \sigma \quad \Delta \rightarrow u : \tau}{\Gamma, \Delta \rightarrow \langle t, u \rangle : \sigma \times \tau} \quad \text{(PAIR)}
\]

\[
\frac{\Gamma \rightarrow t : \sigma \times \tau}{\Gamma \rightarrow \text{fst}(t) : \sigma} \quad \text{(FST)}
\]

\[
\frac{\Gamma \rightarrow t : \sigma \times \tau}{\Gamma \rightarrow \text{snd}(t) : \tau} \quad \text{(SND)}
\]

\[
\frac{\Gamma, x : \sigma \rightarrow t : \tau}{\Gamma \rightarrow \lambda x.t : \sigma \rightarrow \tau} \quad \text{(ABS)}
\]

\[
\frac{\Gamma \rightarrow t : \sigma \rightarrow \tau \quad \Delta \rightarrow u : \sigma}{\Gamma, \Delta \rightarrow tu : \tau} \quad \text{(APP)}
\]

\[
\frac{\Gamma \rightarrow \text{inl}(t) : \sigma + \tau}{\Gamma \rightarrow u : \tau} \quad \text{(INR)}
\]

\[
\frac{\Gamma \rightarrow t : \sigma + \tau \quad \Delta, x : \sigma \rightarrow u : \gamma \quad \Delta, y : \tau \rightarrow v : \gamma}{\Gamma, \Delta \rightarrow \text{case } t \text{ of } \text{inl}(x) \Rightarrow u \mid \text{inr}(y) \Rightarrow v : \gamma} \quad \text{(CASE)}
\]

Figure 4.1: Term Assignment for Natural Deduction
construction has a corresponding rule, for example a function \( \lambda x.t \) is built using the (ABS) rule, application is built using the (APP) rule, etc. One can easily show that if there is a derivation ending in \( \Gamma \rightarrow t : \sigma \), then \( t \) is a well typed term.

Remark that in this system we have the following property: If \( \Gamma \rightarrow t : \sigma \), then \( \Gamma, \Delta \rightarrow t : \sigma \). This property holds simply by application of the weakening rule (W).

We have also introduced three extra kinds of typed \( \lambda \)-term in the deduction system: \text{inl}, \text{inr} and \text{case}. These correspond to the \text{sum}, written as \(+\), which is the \( \lambda \)-calculus counterpart of the \( \lor \) connective. In Exercise 4.3 we give the reader the opportunity to define the rules for \( \beta \) and \( \eta \)-reduction to complete the definition of the typed \( \lambda \)-calculus.

In this formulation of the typed \( \lambda \)-calculus a number of interesting connections arise. There is a striking similarity between the term formation rules and the system of natural deduction that we presented in Chapter 3, Section 3.2.1. For instance the rule defining application (APP) and implication elimination (\( \Rightarrow E \)) are essentially \textit{the same rule}. More precisely, the types of terms correspond to logical formulae: \( \rightarrow \) corresponds to \( \Rightarrow \), \( \times \) to \( \wedge \) and \( + \) to \( \lor \), and the term forming rules correspond to the logical rules, in full we have:

\[
\begin{align*}
\text{(ABS)} & \sim (\Rightarrow I) \\
\text{(APP)} & \sim (\Rightarrow E) \\
\text{(PAIR)} & \sim (\wedge I) \\
\text{(FST)} & \sim (\wedge E_1) \\
\text{(SNR)} & \sim (\wedge E_2) \\
\text{(INR)} & \sim (\lor I) \\
\text{(INL)} & \sim (\lor E_1) \\
\text{(CASE)} & \sim (\lor E)
\end{align*}
\]

and it is also the case that the axiom and structural rules coincide.

With this perspective we can see the typed \( \lambda \)-calculus as a term assignment to natural deduction proofs, that is to say that we can decorate \( \text{NJ}_0 \) derivations with \( \lambda \)-terms yielding the system of typed \( \lambda \)-calculus. Having two systems that are isomorphic in this way gives a new perspective on both systems. For example, we could take the \textit{logical view} where the formulae and their proofs are primary, and the terms are nothing more than notations for proofs. We can also take the \textit{computational view} where terms (functional programs) are primary, and the natural deduction system is a \textit{type inference system} for terms. However, the Curry-Howard isomorphism states that these two views exist together; we can switch perspective as we wish.

We give an example. Consider the following derivation in \( \text{NJ}_0 \).

\[
\begin{array}{c}
\Phi \rightarrow \Phi \\
\Phi \Rightarrow \Phi \Rightarrow I \\
\Phi \Rightarrow I \Rightarrow I \\
(\Phi \Rightarrow \Phi) \land (\Phi' \Rightarrow \Phi')
\end{array}
\]

If we annotate this proof of \((\Phi \Rightarrow \Phi) \land (\Phi' \Rightarrow \Phi')\) with \( \lambda \)-terms throughout, we get the term \( \langle \lambda x.x, \lambda y.y \rangle \) at the root. Replacing formulae by types and logical connectives with type constructors, we see that this term has type \( (\tau \rightarrow \tau) \times (\tau \rightarrow \tau) \).

We have seen that there is a precise connection between formulae and types, and proofs and terms (programs). However, there is a third and most important part which connects the process of normalisation in natural deduction with reduction (computation) in the typed \( \lambda \)-calculus. The Curry-Howard isomorphism is then summarised by the following table:

| formulae | ~ | types |
| proofs | ~ | programs |
| normalisation | ~ | computation |

The first two parts of the isomorphism are clear from what we have said above. Here we focus on the third part where we see most of the applications of this connection. If we look at the calculus, we have ways of \textit{constructing} terms, and ways of \textit{destructing} them. The following table gives a summary:
4.2. TYPED λ-CALCULUS AND NATURAL DEDUCTION

Here we establish some of the key results for the typed λ-calculus. The third part of the Curry-Howard isomorphism states that each of the reduction rules for the λ-calculus corresponds to a proof reduction.

For example, the term \( \text{fst}(\langle t, u \rangle) \) is an occurrence of such a reduction: it is a product constructor followed by a product destructor. In the calculus we gave the equation \( \text{fst}(\langle t, u \rangle) \to t \). The derivations for \( \text{fst}(\langle t, u \rangle) \) and \( t \) are given by the following:

\[
\frac{(\pi_1) \ldots (\pi_2) \quad (\pi_1) \ldots (\pi_2)}{\Gamma \vdash t : \sigma \quad \Delta \vdash u : \tau \quad \Gamma, \Delta \vdash \langle t, u \rangle : \sigma \times \tau \quad \Gamma, \Delta \vdash \text{fst}(\langle t, u \rangle) : \sigma}
\]

Removing the terms yields exactly the same reduction for natural deduction normalisation that we saw in Chapter 3. The same applies to \( (\lambda x.t)u \to t[u/x] \):

\[
\frac{(\pi_1) \ldots (\pi_2)}{\Gamma, x : \sigma \to t : \tau \quad \Delta \vdash u : \sigma \quad \Gamma \to (\lambda x.t) : \sigma \to \tau \quad \Delta \to u : \sigma \quad \Gamma, \Delta \vdash (\lambda x.t)u : \tau \quad \Gamma, \Delta \vdash t[u/x] : \tau}
\]

This case makes a little clearer the notion of substituting the hypotheses of \( \sigma \) by proofs of \( \sigma \), since we have the notation from the λ-calculus, i.e. \( t[u/x] \).

η-reductions are also reflected in the logic. Here we show the case \( \langle \text{fst}(t), \text{snd}(t) \rangle = t \).

\[
\frac{\Gamma \to t : \sigma \times \tau \quad \Gamma \to t : \sigma \times \tau}{\Gamma \to \text{fst}(t) : \sigma \quad \Gamma \to \text{snd}(t) : \tau \quad \Gamma \to \langle \text{fst}(t), \text{snd}(t) \rangle : \sigma \times \tau \quad \Gamma \to t : \sigma \times \tau}
\]

The remaining cases follow in the same style.

It should be emphasized that a proof \( \bar{\pi} \) of \( \Gamma \to t : \sigma \) is given in terms of \( t \), so that the proof \( \bar{\pi} \) can be reconstructed, in a unique way (modulo applications of the structural rules), from the term \( t \). This is because the term contains the history of the proof. There are various refinements that can be made to the λ-calculus which take into account the structural rules, thus capturing all the history of the proof. Such a calculus would capture the substitution process, thus a calculus of explicit substitutions.

**EXERCISE 4.3**

Give the \( \beta \) and \( \eta \)-reductions for the case construct. [Hint: write the rules for normalisation in NJ first.]

### 4.2.3 Properties

Here we establish some of the key results for the typed λ-calculus. By the Curry-Howard isomorphism, all of these results apply also to intuitionistic natural deduction.
Subject reduction. If we apply one of the reduction rules of the typed \(\lambda\)-calculus to a typed \(\lambda\)-term, we would hope that we still have a term of the same type. The notion of subject reduction states that types are preserved under reduction. This gives a kind of semantic soundness for reduction; nothing bad will happen. The statement of the theorem and proof follow.

Theorem 4.6 If \(\Gamma \rightarrow t : \sigma\) and \(t \rightarrow^\beta_\eta t'\) then \(\Gamma \rightarrow t' : \sigma\).

In proving this theorem, it is useful to factor out a very general lemma.

Lemma 4.7 (Substitution) If \(\Gamma, x : \sigma \rightarrow t : \tau\) and \(\Delta \rightarrow u : \sigma\), then \(\Gamma, \Delta \rightarrow t[u/x] : \tau\).

Proof: By induction over the length of the derivation \(\Gamma, x : \sigma \rightarrow t : \tau\).

We are now in a position where we can prove the subject reduction theorem. Proof: (of Theorem 4.6)

By induction over the derivation of \(\Gamma \rightarrow t : \sigma\). If the derivation ends in a structural rule, or the reduction \(t \rightarrow t'\) takes place in a proper subterm of \(t\), then the result follows by induction. We consider only the case where the reduction takes place at the root, and we prove the theorem for each kind of reduction.

1. Case: \((\lambda x.t)u \rightarrow^\beta t[u/x]\)

Without loss of generality, we assume that the derivation ends in

\[
\frac{\Gamma, x : \sigma \rightarrow t : \tau}{\Gamma \rightarrow \lambda x.t : \sigma \rightarrow \tau} \text{ (ABS)}
\]

\[
\frac{\Delta \rightarrow u : \sigma}{\Gamma, \Delta \rightarrow (\lambda x.t)u : \tau} \text{ (APP)}
\]

(The case where there are structural rules between APP and ABS follows in a similar way.)

We can now apply Lemma 4.7 on the premises of the above to obtain \(\Gamma, \Delta \rightarrow t[u/x] : \tau\) as required.

2. Case: \(\lambda x.tx \rightarrow^\eta t\)

As in the previous case, we can assume without loss of generality, that the derivation ends in

\[
\frac{\Gamma \rightarrow t : \sigma \rightarrow \tau}{x : \sigma \rightarrow x : \sigma} \text{ (VAR)}
\]

\[
\frac{\Delta \rightarrow u : \sigma}{\Gamma, x : \sigma \rightarrow tx : \tau} \text{ (APP)}
\]

\[
\frac{\Gamma \rightarrow \lambda x.tx : \sigma \rightarrow \tau}{\Gamma \rightarrow t : \sigma \rightarrow \tau} \text{ (ABS)}
\]

Then \(\Gamma \rightarrow t : \sigma \rightarrow \tau\) follows from the first premise.

3. Case: \(\text{fst}(\langle t, u \rangle) \rightarrow^\beta t\).

Assume without loss of generality that the derivation ends in

\[
\frac{\Gamma \rightarrow t : \sigma}{\Delta \rightarrow u : \tau} \text{ (PAIR)}
\]

\[
\frac{\Gamma, \Delta \rightarrow \langle t, u \rangle : \sigma \times \tau}{\Gamma, \Delta \rightarrow \text{fst}(\langle t, u \rangle) : \sigma} \text{ (FST)}
\]

Then \(\Gamma, \Delta \rightarrow t : \sigma\) follows from the first premise, by using the weakening rule.

4. Case: \(\text{snd}(\langle t, u \rangle) \rightarrow^\beta u\). This is only a slight variation from the previous case.
5. Case: \(\langle \text{fst}(t), \text{snd}(t)\rangle \rightarrow^* t\).

Assume, again without loss of generality, that the derivation ends in

\[
\begin{array}{c}
\Gamma \rightarrow t : \sigma \times \tau \\
\Gamma \rightarrow \text{fst}(t) : \sigma \\
\Gamma \rightarrow \text{snd}(t) : \tau
\end{array}
\]

\[
\frac{}{\Gamma \rightarrow \langle \text{fst}(t), \text{snd}(t)\rangle : \sigma \times \tau} \quad \text{(PAIR)}
\]

Then \(\Gamma \rightarrow t : \sigma \times \tau\) from one of the premises, together with the contraction rule.

\[\square\]

For intuitionistic natural deduction, the subject reduction theorem says that contracting a proof using one step normalisation gives a proof of the same formula.

**Normalisation.** Normalisation results generally come in two flavours. First, there is a notion of the existence of a reduction strategy that terminates. This is the property that we prove here. There is also a stronger variety which states that all reduction sequences terminate. Here we just consider the \(\beta\)-reductions, but the results extend to \(\eta\) in a straightforward way. We begin with a definition of the class of terms that have already terminated.

**Definition 4.8** A term \(t\) is said to be in \(\beta\)-normal form if there are no sub-terms of the form \((\lambda x t)u\), \(\text{fst}(\langle t, u \rangle)\) or \(\text{snd}(\langle t, u \rangle)\).

We can now formalise two normalisation results.

1. Weak normalisation. There exists a finite sequence

\[t \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n\]

such that \(t_n\) is a \(\beta\)-normal form.

2. Strong normalisation. All reduction sequences are finite.

Before proving either of these, we define a number of measures on typed terms.

**Definition 4.9**

- The degree \(\text{deg}(\sigma)\) of a type \(\sigma\) is defined as:

\[
\begin{align*}
\text{deg}(\alpha) & = 1 \\
\text{deg}(\sigma \times \tau) & = \text{deg}(\sigma) + \text{deg}(\tau) = \max\{\text{deg}(\sigma), \text{deg}(\tau)\} + 1
\end{align*}
\]

- The rank \(R(r)\) of a redex \(r\) is defined as:

\[
\begin{align*}
1. \text{If } r & = (\lambda x t)u, \text{ and } \lambda x t : \sigma \rightarrow \tau, \text{ then} \\
R((\lambda x t)u) & = \text{deg}(\sigma \rightarrow \tau)
\end{align*}
\]

\[
\begin{align*}
2. \text{If } r & = \text{fst}(\langle t, u \rangle) \text{ or } r = \text{snd}(\langle t, u \rangle), \text{ and } \langle t, u \rangle : \sigma \times \tau, \text{ then} \\
R(\text{fst}(\langle t, u \rangle)) & = R(\text{snd}(\langle t, u \rangle)) = \text{deg}(\sigma \times \tau)
\end{align*}
\]

- The degree \(d(t)\) of a term \(t\) is defined as the supremum of the ranks of the redexes in \(t\):

\[
d(t) = \max\{R(r_i) : r_i \text{ is a redex in the term } t\}.
\]

If \(t\) is in normal form, then \(d(t) = 0\).

**Lemma 4.10** If \(x : \sigma\) then \(d(t[u/x]) \leq \max\{d(t), d(u), \text{deg}(\sigma)\}\).
**Proof:** If the substitution does not create any new redexes, the result trivially holds. The only problematic case is the creation of a new redex, which must be one of: $(\lambda x.t')u$, $\text{fst}(u)$ or $\text{snd}(u)$. In all three cases, the degree of the new redex is $\deg(\sigma)$. \qed

We are now ready to state the main result.

**Theorem 4.11 (Weak Normalisation)** If there is a derivation ending in $\Gamma \rightarrow t_0 : \sigma$ then there is a finite sequence of reductions $t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$ such that $t_n$ is in $\beta$-normal form.

**Proof:** It suffices to show that given a reducible term $t$ we can always find a $u$ such that $t \rightarrow^+ u$ and $d(t) > d(u)$. For this we will show that reducing all of the innermost redexes of $t$ with maximal rank $r$, we obtain a term $u$ in which all the redexes have rank less than $r$.

We first assume that there is only one innermost redex with maximal rank, and show that by reducing this redex we decrease the degree of the term.

We can do this by an induction over the structure of terms. The cases for variables and abstractions are trivial. The only interesting case is for applications. There are three cases to consider. If $tu \rightarrow tu'$ then the result follows by hypothesis. The first interesting case is when $(\lambda x.t)u \rightarrow t[u/x]$, then the result follows by the previous Lemma 4.10:

$$d(t[u/x]) \leq \deg(\sigma) < \deg(\sigma \rightarrow \tau) = d((\lambda x.t)u)$$

The final case is when $tu \rightarrow t'u$. If this reduction creates a redex, for example $t \rightarrow \lambda x.t''$, then we must show that the hypothesis holds, which is again straightforward.

Now, the result for a term with $n$ innermost redexes with maximal rank follows by a simple induction. \qed

This result can indeed be strengthened to give the strong normalisation property for the typed $\lambda$-calculus, but we will not cover that here.

Of course, this result (and method of proof) also holds for the logic, thus providing a proof of the normalisation theorem for $\text{NJ}_0$ (Theorem 3.6 of Chapter 3).

**Church-Rosser.** The Church-Rosser property tells us that the choice of reduction order is not important.

**Theorem 4.12 (Church-Rosser)** If $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$ then there exists a term $t_3$ such that $t_1 \rightarrow^* t_3$ and $t_2 \rightarrow^* t_3$.

**Corollary 4.13 (Uniqueness of Normal Form)** A term $t : \sigma$ has exactly one $\beta$-normal form.

**Proof:** Assume $t : \sigma$, and that $t$ has two $\beta$-normal forms, say $u$ and $v$. By the Church-Rosser theorem there is a term $w$ such that $u \rightarrow^* w$ and $v \rightarrow^* w$. By assumption $u$ and $v$ cannot be reduced further, hence $u \equiv v \equiv w$. \qed

**Consistency.** From the Church-Rosser theorem, we obtain the fact that the calculus is consistent.

**Corollary 4.14** The $\lambda$-calculus is consistent — it is not possible to prove $u \equiv_\beta v$ for arbitrary terms.

**Proof:** Take two terms $t$ and $u$. If $t \rightarrow^* u$ then $t \equiv_\beta u$, and if $t \equiv_\beta u$ then it is easy to show from Church-Rosser that there is a common term $w$ such that $t \rightarrow^* w$ and $u \rightarrow^* w$. If we take two variables $x$ and $y$, which are in normal form, then there is no such term $w$. Hence it is not possible to show $x \equiv_\beta y$. \qed

Again, all of these results apply to intuitionistic natural deduction, showing that the system is consistent.
Summary. We have seen a sequence of results for the typed $\lambda$-calculus that are applicable to intuitionistic natural deduction proofs, thanks to the Curry-Howard isomorphism. In Section 4.4 we show some other applications of this isomorphism which do not rely on two pre-existing systems.

EXERCISE 4.4

Build normal derivations of the following formulae:

$$(i) \quad A \Rightarrow B \Rightarrow A$$

$$(ii) \quad (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$$

$$(iii) \quad (A \land B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow C)$$

$$(iv) \quad (A \Rightarrow B \Rightarrow C) \Rightarrow (A \land B \Rightarrow C)$$

Annotate each derivation with $\lambda$-terms, replace $\Rightarrow$ by $\rightarrow$, propositions by type variables, and write the corresponding $\lambda$-term for each formula.

4.3 Combinatory Logic and Hilbert-style Axioms

In this section we look at another formulation of the Curry-Howard correspondence, which applies to the Hilbert-style axioms of intuitionistic logic and a system called combinatory logic. We restrict our attention here to the implicational fragment of the logic.

4.3.1 Combinatory Logic

Combinatory logic is another formal system, like the $\lambda$-calculus, which is also regarded as being a foundation of functional programming. Indeed, it has proved fruitful as an implementation technique for this kind of language, see for example Peyton Jones (1987).

Combinatory logic is built up from only two typed elements, called combiners:

$$K_{\sigma\tau} : \sigma \rightarrow \tau \rightarrow \sigma$$

$$S_{\sigma\tau\gamma} : (\sigma \rightarrow \tau \rightarrow \gamma) \rightarrow (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \gamma$$

together with a collection of typed variables, $x : \sigma$, etc. Terms can be combined using only the rule of application:

$$\Gamma \rightarrow M : \sigma \rightarrow \tau \quad \Delta \rightarrow N : \sigma$$

$$\Gamma, \Delta \rightarrow (MN) : \tau$$

Note that this is nothing more than modus ponens, or the $(\Rightarrow E)$ rule. As usual, we assume application associates to the left, and drop excessive parentheses.

This theory comes equipped with the following two axioms:

$$K P Q = P$$

$$S P Q R = P R (Q R)$$

where we have dropped types for clarity (we will avoid writing types whenever possible). As with the $\lambda$-calculus we can think of this as a reduction from left to right, and use $\rightarrow_w$ instead of the equality and $\rightarrow^*_w$ for the reflexive and transitive closure of $\rightarrow_w$. These two rewrite rules are allowed in any context (which is the same as for the $\lambda$-calculus, Definition 4.3, except that there is no case for abstraction). The $\rightarrow_w$ notation is used to signify that reduction in combinatory logic is the so-called weak reduction. We will come back to this issue below.

A simple example of reduction is given by:

$$SKK M \rightarrow_w K M (K M)$$

$$\rightarrow_w M$$

which shows that, for any $M$, we have $SKK M \rightarrow^*_w M$, thus $SKK$ behaves as an identity combinator. It is convenient to have the identity combinator as primitive in the theory, which is denoted by $I_{\sigma} : \sigma \rightarrow \sigma$. The
reader will easily verify that the type we assigned is in fact the correct one inferred from the application of SKK.

Combinatory logic is very close in fact to the typed \( \lambda \)-calculus. The main difference between the two systems is the missing abstraction rule from the term construction in combinatory logic, and the missing two combinators from the \( \lambda \)-calculus. However, it is possible to translate between the two systems. Let us first look at coding combinatory logic in the \( \lambda \)-calculus, where all we are required to do is give a \( \lambda \)-term corresponding to each of the combinators \( S, K \) and \( I \).

**Definition 4.15** We define a translation \((\cdot)_\lambda\) from (typed) combinators to the (typed) \( \lambda \)-calculus as follows:

\[
\begin{align*}
(x)_\lambda &= x \\
(I_x)_\lambda &= \lambda x.x : \sigma \rightarrow \sigma \\
(K_{\sigma,\tau})_\lambda &= \lambda xy.x : \sigma \rightarrow \tau \rightarrow \sigma \\
(S_{\sigma,\tau,\gamma})_\lambda &= \lambda yz.xz(yz) : (\sigma \rightarrow \tau \rightarrow \gamma) \rightarrow (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \gamma \\
(MN)_\lambda &= (M)_\lambda(N)_\lambda
\end{align*}
\]

We leave the reader to verify that the typed terms are indeed well formed. Next, we verify that the given \( \lambda \)-terms have the required behaviour.

**Theorem 4.16** If \( P \rightarrow^* Q \) then \( (P)_\lambda \rightarrow^* (Q)_\lambda \).

**Proof:** A straightforward computation shows the result:

- \((I_x)_\lambda = (\lambda x.x)x \rightarrow x\).
- \((K_{xy})_\lambda = (\lambda xy.x)y \rightarrow (\lambda y.y)x \rightarrow x\).
- \((S_{xyz})_\lambda = (\lambda yz.xz(yz))xyz \rightarrow xyz \rightarrow xy(yz) \) as required. 

\( \square \)

Finally, as an example, we show the reduction of \((SKKM)_\lambda\) in the \( \lambda \)-calculus. Here are a few snapshots of the reduction:

\[
(SKKM)_\lambda = (\lambda yz.xz(yz))(\lambda xy.x)(\lambda xy.x)M \\
\rightarrow (\lambda z.(\lambda xy.x)(\lambda xy.x))M \\
\rightarrow (\lambda x.((\lambda xy.x)z)((\lambda xy.x)z))M \\
\rightarrow (\lambda y.((\lambda xy.x)M)((\lambda xy.x)M)) \\
\rightarrow* (\lambda y.(y.M))(\lambda y.M) \\
\rightarrow M
\]

The reader will be able to add the types to the above to check that all the reductions were indeed well-formed.

Next, we turn to understanding the \( \lambda \)-calculus in combinatory logic.

**Definition 4.17** Let us define a translation of terms from the (implicational fragment of) \( \lambda \)-calculus into combinatory logic. We do this in two steps, first we define a translation \((\cdot)_{CL}\) into a system close to combinatory logic where we allow abstraction. In other words, we add to combinatory logic the following abstraction operation: \([\cdot]\). We now define the translation as follows (again, written without types for clarity):

\[
\begin{align*}
(x)_{CL} &= x \\
(tu)_{CL} &= (t)_{CL}(u)_{CL} \\
(\lambda x.t)_{CL} &= [x]_{CL}t
\end{align*}
\]

Next, we must eliminate this additional abstraction operation to complete the coding into combinatory logic. This is achieved by defining \([\cdot]\) as the following function:
\[
\begin{align*}
[x]x &= I \\
[x]t &= Kn (\text{if } x \notin FV(t)) \\
[x](tu) &= S([x]t)([x]u)
\end{align*}
\]

where \(FV\) is the obvious free variable function for combinatory logic \((FV(S) = FV(K) = \emptyset, FV(x) = x, \text{ and } FV(MN) = FV(M) \cup FV(N))\). Note that there is no rule of the form \([x][y]t\), in other words, the translation works from the inside out.

Let us look at an example of this translation. We will compute \((\lambda xy.x)_{CL}\) as follows:

\[
\begin{align*}
(\lambda xy.x)_{CL} &= [x][y]x \\
&= [x](Kx) \\
&= S([x]K)([x]x) \\
&= S(KK)I
\end{align*}
\]

Again we leave the reader to check that the types are well behaved under this translation.

Corresponding to Theorem 4.16 we would like to prove an analogous results stating that reduction in the \(\lambda\)-calculus is preserved under this translation. However, this is not the case since reduction in combinatory logic is weak. To illustrate what we mean weak reduction, we consider an example. Take the term \(KI\) which we can see by inspection of the rules for \(\to\) is in normal form — no reduction can be applied. However, if we look at the corresponding \(\lambda\)-term, using the translation given previously, we get \((SK\lambda) = (\lambda xy.x)(\lambda yby)\) which is a \(\lambda\)-term not in normal form.

We refer the interested reader to the literature, especially Barendregt (1984, Chapter 7), for a discussion on how to add additional axioms to combinatory logic so that \(\beta\)-reduction is preserved under the translation of \(\lambda\)-calculus into combinatory logic.

4.3.2 The Correspondence with Hilbert Axioms

For the \(\lambda\)-calculus we showed that if we remove the terms from typing rules, we get exactly the natural deduction system for intuitionistic logic. Here we show that we can do exactly the same trick for combinatory logic. If we take away the terms from the following:

\[
\begin{align*}
&K_{\sigma,\tau} : \sigma \to \tau \to \sigma \\
&S_{\sigma,\tau,\gamma} : (\sigma \to \tau \to \gamma) \to (\sigma \to \tau) \to \sigma \to \gamma
\end{align*}
\]

and replace the functional arrow \(\to\) by implication \(\Rightarrow\), and types by formulas, as is familiar by now. We are then left with the Hilbert-style axiom schema for the implicational fragment for intuitionistic logic:

\[
\Phi \Rightarrow \Phi' \Rightarrow \Phi \\
(\Phi \Rightarrow \Phi' \Rightarrow \Phi'' \Rightarrow (\Phi \Rightarrow \Phi') \Rightarrow \Phi \Rightarrow \Phi''
\]

As we have already seen, the rule for application in combinatory logic is nothing more than modus ponens \((MP)\). \(S\) and \(K\) correspond to the axioms of \(H\) (Chapter 3, Section 3.2.4), terms built out of the combinators correspond to Hilbert-style proofs, and types correspond to formulae.

As with all formulations of this style of correspondence, the main application is that we have two different view points on the same theory. The objects for manipulation in each case are quite different and lend themselves well to certain manipulations. To give an example of this, we show how we can give an alternative proof of Theorem 3.2 from Chapter 3. We are required to prove that the system \(NJ_0\) and Hilbert axioms are equivalent systems, in the sense that the same formulae are provable. Recasting this theorem into the corresponding formalisms requires that we prove that the \(\lambda\)-calculus and combinatory logic are equivalent systems, in other words, we need to show that we can translate any derivation (modulo the structural rules) in the \(\lambda\)-calculus \(\Gamma \vdash t : \sigma\) into a derivation in combinatory logic, and vice-versa. But this is precisely what we did in Definitions 4.17 and 4.15 above.

There is no doubt that the proof at the level of \(\lambda\)-terms and combinators is more manageable. Inductions over the structure of terms are not only easier to grasp, but also easier to write down than the corresponding inductions over the length of the derivation. Of course, the two proofs, the first relating \(NJ_0\) and the
hilbert axioms $\mathbf{H}$ and the second relating $\lambda$-calculus with combinatory logic, are essentially the same proofs because of the Curry-Howard correspondence.

A second application of this correspondence is that we get another way of building proofs in Hilbert-style intuitionistic logic. We give an example and give a proof of $\Phi \Rightarrow \Phi$ from the axioms. To do this, we have to build a term $\mathbf{I}$ out of the combinators such that $\mathbf{I} : \sigma \rightarrow \sigma$ and satisfies $\mathbf{I}x = x$. Now, $\mathbf{SKK}$ is such a term, as we have already seen. The term $\mathbf{SKK}$ is considerably more manageable than the proof that we used for this in Chapter 3, Section 3.2.4.

We refer the reader to Howard (1980), and Girard et al. (1989) for further details on the Curry-Howard isomorphism.

**EXERCISE 4.5**

Compute the following, using the translations ($\cdot$)$_{CL}$ and ($\cdot$)$_{\lambda}$ given. Where necessary, reduce the terms to normal form (after performing the translation!).

\[
\begin{align*}
(i) & \quad (\lambda x.x)_{CL} & (ii) & \quad (\lambda x.yx)_{CL} \\
(iii) & \quad (\lambda x.yx)_{CL} & (iv) & \quad ((\lambda x.yx)(\lambda x.x))_{CL} \\
(v) & \quad (\mathbf{KI})_{\lambda} & (vi) & \quad (xy)_{\lambda} \\
(vii) & \quad (x\mathbf{I})_{\lambda} & (viii) & \quad (S(KS)K)_{\lambda}
\end{align*}
\]

### 4.4 Applications of the Curry-Howard Correspondence

We have already seen several applications of the Curry-Howard correspondence in the form of simpler proof methods for the typed $\lambda$-calculus (programming language) which carry over to the logic. In this section we will study some other applications, in particular to the development of new languages.

It is worth remarking that the correspondence between the $\lambda$-calculus and intuitionistic natural deduction, and combinatory logic and Hilbert axioms, are two of the simplest cases of the isomorphism. The ideas can in fact be generalised to many other logics. Probably one of the most well-known additions is Girard’s (1972) System $F$, which is a term assignment to second order intuitionistic logic. By establishing an isomorphism between System $F$ and the logic all results obtained in the calculus can be carried over. Girard also gave a proof of strong normalisation for System $F$ using a proof technique called “Candidate de Reductibilité” (See Gallier (1990) for a good exposition), which, thanks to the Curry-Howard isomorphism, gives a strong normalisation result of second order intuitionistic logic in purely logical terms. These ideas were then generalised to the calculus $F_\omega$ giving a strong normalisation result for higher-order intuitionistic logic using only proof theory (a result previously conjectured by Takeuti; see for example (Girard, 1987b; Girard, 1972) for details).

Another example is the $\lambda\mu$-calculus developed by Michel Parigot (1992) as the term assignment of classical logic, thus giving a programming language based on classical logic. This work, starting from work of Griffin (1990) extends the Curry-Howard style isomorphism to classical logic, and allows us to get the computational content from a classical logic proof. Double negation elimination, $\neg\neg A \Rightarrow A$, turns out to be the type of Felleisen’s (1988) “catch” operation, thus giving classical logic the status of a calculus to reason about control and continuation operations.

A further example is the search for a programming language based on linear logic giving a linear functional programming language. With linear logic we have a system that enables us to talk about the fine details of the reduction process of a program. In particular it brings out features of copying and discarding which are strongly acknowledged as the “expensive” parts of a computation. This computational-logic connection has been used to develop programming languages that keep a tighter control over resource management. See for example (Abramsky, 1993).

This list goes on, and is rather long! We now outline one of these applications in a little more detail.

#### 4.4.1 System $F$

One of the most significant applications of the Curry-Howard isomorphism is the work on System $F$, which extends the isomorphism to second order intuitionistic logic.
4.4. APPLICATIONS OF THE CURRY-HOWARD CORRESPONDENCE

In the same way as the $\lambda$-calculus allows functions to become first class citizens (they can be passed as arguments to other functions), the idea of System $F$ can be understood intuitively as making types first class citizens. If we write a function $\lambda x.t : \sigma \to \tau$, then the type for this function is fixed. The idea is to extend the system to allow types to be passed as arguments. Hence we need a notion of type abstraction and type application; a way of making terms depend on types.

We write $\Lambda X.t$ as a term $t$ that depends on a type $X$, to reflect this in the type, assuming $t : \sigma$, we write $\Lambda X.t : \forall X \sigma$, where $\forall$ is the second order universal quantification. Here is an example of type abstraction: $\Lambda X.\lambda x.x : \forall X.X \to X$. This is a function that is the identity function at all types $X$. (A function working on any type is called polymorphic.)

To use this function, we have to apply it first to some type. For example, assume to have already defined types such as integers $(\mathbb{N})$, Booleans $(\mathbb{B})$, etc. Then the following application says that we want to use the identity function only on integers:

$$(\Lambda X.\lambda x.x : \forall X.X \to X)\mathbb{N}$$

There is a corresponding reduction rule which is similar to $\beta$-reduction that allows us to reduce this to the expected $\lambda x.x : \mathbb{N} \to \mathbb{N}$.

We will now be a little more formal about the types and terms of System $F$.

**Definition 4.18 (Type schemes)** The types of System $F$ are given by the set $\mathcal{T}$:

- Type variables, denoted by $X, Y, \ldots$.
- If $\sigma, \tau \in \mathcal{T}$ then $\sigma \to \tau \in \mathcal{T}$.
- If $\sigma \in \mathcal{T}$ and $X$ is a type variable, then $\forall X.\sigma \in \mathcal{T}$.

Note that these types are nothing more than propositions quantified at the second order (cf. Chapter 2, Section 2.5.3.)

**Definition 4.19 (Terms)** The set of terms in System $F$ are given by extending the $\lambda$-calculus with the following operations:

- Type abstraction: If $t : \sigma$ is a term, then $\Lambda X.t : \sigma$ is a term of type $\forall X.\sigma$ provided that $X$ does not occur free in any of the free variables of $t$.
- Type application: If $t : \forall X.\sigma$ is a typed term, and $\tau$ is a type, then $t\tau$ is a term of type $\sigma[\tau/X]$.

We can now give the term assignment for System $F$ to second order propositional logic.

$$
\begin{align*}
\Gamma \vdash t : \sigma & \quad \text{(Ax)} \\
\Gamma \vdash \Lambda X.t : \forall X.\sigma & \quad \text{(Ax)} \\
\Gamma \vdash t : \forall X.\sigma, \tau \in \mathcal{T} & \quad \text{(Ax)} \\
\Gamma \vdash t\tau : \sigma[\tau/X] & \quad \text{(Ax)}
\end{align*}
$$

where there is the usual side condition on the (Ax) rule that $X$ is not free in $t$ or $\Gamma$.

There are two reductions in this calculus. The usual $\beta$-reduction as we saw for the $\lambda$-calculus, together with an additional rule for type application, denoted $\beta^\tau$. The two reductions are given by:

$$
\begin{align*}
\beta : \quad (\lambda x.t)u & \to t[u/x] \\
\beta^\tau : \quad (\Lambda X.t)\tau & \to t[\tau/X]
\end{align*}
$$

where term substitution is already defined (Page 97), and type substitution is defined in the obvious way.

System $F$ is a very powerful calculus. It captures all of second order arithmetic, and thus all of current mathematics. It can also be seen as a programming language: all the usual data-types (inductive types) that one uses in programming languages are indeed definable in System $F$. Note in particular that we restricted the calculus to the implication fragment. We can recover products and sums in a natural way. For example, define $\sigma \times \tau$ as $\forall X.(\sigma \to \tau \to X) \to X$, which gives the term $\langle u, v \rangle : \sigma \times \tau$ as $\Lambda X.\lambda x.xu \cdot xv$. We leave the definition of the projections $\text{fst}$ and $\text{snd}$ as an exercise. The reader can find many more examples of this kind of data structure in Girard et al. (1989).
If we look at the reduction step in second order propositional logic, we see that there is an additional reduction in the case where we have an \((\forall I)\) followed by a \((\forall E)\). As usual, this can be eliminated from the derivation in the following way:

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

becomes simply the derivation ending in \(\Gamma \rightarrow \Phi[\Phi'/X]\).

This calculus has been shown to be strongly normalising, and satisfies the Church-Rosser property. This gives a purely proof theoretic account of showing consistency of second order propositional calculus. Remark also that via the double negation translations of classical logic into intuitionistic logic given in Chapter 3, Section 3.5, this result covers classical logic too. However, it is also possible to work directly with classical logic and develop a corresponding calculus, this is what has been done by Parigot (1992) with the \(\lambda\mu\)-calculus.

\textbf{EXERCISE 4.6}

Define in System \(F\) the operations \(\text{fst}\) and \(\text{snd}\) so that they satisfy the usual properties, i.e. \(\text{fst}((t, u)) \rightarrow^* t\) and \(\text{snd}((t, u)) \rightarrow^* u\).

\textbf{EXERCISE 4.7}

Give a typing system for the following calculus, which is a restriction of the \(\lambda\)-calculus where all variables occur exactly once in the body of a term.

<table>
<thead>
<tr>
<th>Term</th>
<th>Constraint</th>
<th>(FV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td></td>
<td>(x)</td>
</tr>
<tr>
<td>(\lambda x.t)</td>
<td>(x \in FV(t))</td>
<td>(FV(t) \setminus {x})</td>
</tr>
<tr>
<td>(tu)</td>
<td>(FV(t) \cap FV(u) = \emptyset)</td>
<td>(FV(t) \cup FV(u))</td>
</tr>
</tbody>
</table>

Then prove strong normalisation for this calculus, and hence also for the corresponding logic.