CHAPTER 9

Strong normalisation and the typed lambda calculus

In the previous chapter we looked at some reduction rules for intuitionistic natural deduction proofs and we have seen that by applying these in a particular way such proofs can eventually be brought in a normal form, meaning that to the resulting derivation no more reduction steps can be applied.

It turns out that more is true: suppose someone starts from a derivation in intuitionistic natural deduction and starts to apply these reduction rules in some random way. Will this person end up with a proof in normal form? It turns out that the answer is *yes*: however one applies the reduction rules one must eventually end up with a proof in normal form. This is called *strong normalisation* and was proved by Prawitz in 1971.

It turns out that even more is true: suppose two people start applying these reduction rules completely independently from each other in some random way. Will they end up with the *same* proof in normal form? The answer is again *yes*: normal forms are also unique.

Proofs of these facts are notoriously complicated. Actually, we will leave it to the reader to prove uniqueness of normal forms from strong normalisation and concentrate on strong normalisation instead. Our proof here combines several insights from several people (Curry, Howard, Tait amongst others) and requires us to make a detour via the *typed lambda calculus*, a system we will now introduce.

1. Typed lambda calculus

1.1. Basic syntax.

DEFINITION 1.1. The (simple) *types* over a set A of atomic (or base) types are defined inductively as follows:

(1) every element $\sigma \in A$ is a type.
(2) if σ and τ are types, then so are $\sigma \times \tau$ and $\sigma \to \tau$.

We will assume that for each type σ we have a countable set of variables of that type and that for distinct types these variables are distinct. In addition, we have certain constants ("combinators"): for each pair of types ρ, σ combinators $\mathbf{p}^{\rho,\sigma}, \mathbf{p}_0^{\rho,\sigma}, \mathbf{p}_1^{\rho,\sigma}$ of types $\rho \to (\sigma \to \rho \times \sigma)$, $\rho \times \sigma \to \rho$ and $\rho \times \sigma \to \sigma$, respectively.

DEFINITION 1.2. The λ-terms of a certain type are defined inductively as follows:

(1) each variable or constant of type σ will be a λ-term of type σ.
(2) if s is a λ-term of type $\sigma \to \tau$ and t is a λ-term of type σ, then st is a λ-term of type τ.

(3) if x^σ is a variable of type σ and t is a λ-term of type τ, then $\lambda x^\sigma.t$ is a λ-term of type $\sigma \to \tau$.

REMARK 1.3. In step (2) of the previous definition we say that st is obtained by *applying* s to t. The convention is that application associates to the left, meaning that an expression like $fxyz$ has to be read as $(((fx)y)z)$.

REMARK 1.4. In step (3) of the previous definition we say that $\lambda x.t$ is obtained by *lambda abstracting* x in t. The result is an expression in which the variable x is no longer free: it has become bound by λx.

This might be a good point to introduce our conventions concerning bound and free variables. Similar conventions will be in place when we will move to predicate logic.

- We will identify two λ-terms if they can be obtained from each other by a systematic renaming of bound variables (that is, if they are "α-equivalent"). So, officially, λ-terms are α-equivalence classes of syntactic expressions.
- In practice, we will work with representatives, that is, concrete syntactic expressions. For convenience, we will assume that we have chosen a representative in which no variable occurs both free and bound. Indeed, bound variables can always be renamed in such a way that this happens.
- The result of substituting a λ-term t for a variable x in a λ-term s will be denoted $s[t/x]$. In this case we will always assume that the substitution was safe in the sense that no variable occurring in t has become bound in $s[t/x]$. Again, bound variables can always be renamed in such a way that this is the case.

1.2. Reduction.

DEFINITION 1.5. An expression on the left of the table below is called a *redex*. If t is a redex and t' is the corresponding expression on the right of the table, then we will say that t *converts to* t' and we will write t conv t'.

$(\lambda x.s)t$	$s[t/x]$
$\mathbf{p}_i(\mathbf{p}t_0t_1)$	t_i

What we explore in this section is what happens if one starts from any expression in the typed lambda calculus and one starts rewriting it using the rules above.

DEFINITION 1.6. The *reduction relation* $\succeq$ is inductively defined by:

$$t \succeq t$$
$$t \text{ conv } t' \Rightarrow t \succeq t'$$
$$t \succeq t' \Rightarrow t''t \succeq t''t'$$
$$t \succeq t' \Rightarrow tt'' \succeq t't''$$
$$t \succeq t', t' \succeq t'' \Rightarrow t \succeq t''$$

If $t \succeq t'$ we shall say that t *reduces* to t'. We write $t \succ_1 t'$ if t' is obtained from t by converting a single redex in t. A sequence $t_1 \succ_1 t_2 \succ_1 \succ_1 \ldots \succ_1 t_n$ is called a *reduction sequence*. Note that $t \succeq t'$ if and only if there is a reduction sequence starting from t and ending with t' (in other words, $\succeq$ is the transitive and reflexive closure of $\succ_1$).

LEMMA 1.7. *If $t \succeq t'$ and t is of type σ, then so is t'.*

DEFINITION 1.8. A term t is in *normal form*, if t does not contain a redex.

DEFINITION 1.9. A term t is *normalisable* if there is a term t' in normal form such that $t \succeq t'$. We will say that t is *strongly normalisable* if every reduction path is finite: this means that there is some number $n = \nu(t)$ such that there is a reduction sequence $t = t_1 \succ_1 t_2 \succ_1 \ldots \succ_1 t_n$ of length n but there are no reduction sequences of greater length.

Our goal will be to show that every term in the typed lambda calculus is strongly normalisable.

1.3. Strong normalisation. In order to show this we use a *computability predicate*. This method was first employed by Tait and we will do the same here.

DEFINITION 1.10. The *computable terms* are defined by induction on the type structure as follows:

(1) A term t of an atomic type is computable if it is strongly normalisable.
(2) A term t of type $\sigma \to \tau$ is computable if for any computable term t' of type σ the term tt' is computable as well.
(3) A term t of type $\sigma \times \tau$ is computable if both $\mathbf{p}_0 t$ and $\mathbf{p}_1 t$ are computable.

DEFINITION 1.11. An expression is *neutral* if it is *not* of one of the following forms:

$$\mathbf{p}t_1t_2, \quad \lambda x.t.$$

LEMMA 1.12. (i) *If s is computable, then s is strongly normalisable.*
(ii) *If s is computable and $s \succeq t$, then t is computable.*
(iii) *If t is neutral and every s such that $t \succ_1 s$ is computable, then t is computable. (In particular, if t is neutral and normal, then t is computable.)*

PROOF. We prove (i)-(iii) by simultaneous induction on the type structure.

Base types:

(i) is immediate.
(ii) If every reduction path from s is finite and s reduces to t, then any reduction path from t must also be finite.
(iii) Any reduction path from t must go through some s with $t \succ_1 s$. If all reduction paths from such s eventually terminate, then all reduction paths from t must eventually terminate as well.

Product types:

(i) If s is computable, then so is $\mathbf{p}_0 s$ and hence $\mathbf{p}_0 s$ is strongly normalisable by induction hypothesis. But since every reduction sequence $s \succ_1 s_1 \succ s_2 \succ_1 \ldots$ gives rise to a reduction sequence $\mathbf{p}_0 s \succ_1 \mathbf{p}_0 s_1 \succ_1 \mathbf{p}_0 s_2 \succ_1 \ldots$, such reduction sequences must all eventually terminate, and therefore s is strongly normalisable.
(ii) If $s \succeq t$ then $\mathbf{p}_i s \succeq \mathbf{p}_i t$. So if s is computable, then so is t.
(iii) Suppose t is neutral and every one-step reduct s from t is computable. The fact that t is neutral means that t is not of the form $\mathbf{p}t_1t_2$ and therefore every one-step reduct s' from $\mathbf{p}_i t$ is of the form $\mathbf{p}_i s$ with $t \succ_1 s$. Therefore both $\mathbf{p}_i t$ are computable by induction hypothesis and hence so is t.

Function types:

(i) Suppose s of type $\sigma \to \tau$ is computable. Let x be a variable of type σ and note that by induction hypothesis applied to (iii), x is computable; therefore sx is computable as well. But since every reduction sequence $s \succ_1 s_1 \succ s_2 \succ_1 \dots$ gives rise to a reduction sequence $sx \succ_1 s_1x \succ_1 s_2x \succ_1 \dots$, such reduction sequences must all eventually terminate, and s is strongly normalisable.

(ii) Suppose that s of type $\sigma \to \tau$ is computable and $s \succeq t$. Then for every computable u of type σ we have that su is computable and $su \succeq tu$. So tu is computable by induction hypothesis, and therefore t is computable.

(iii) Suppose t is a neutral expression of type $\sigma \to \tau$ and every s such that $t \succ_1 s$ is computable. We need to show that tu is computable whenever u is computable and because we know that computable terms of type σ are strongly normalisable by induction hypothesis, we can prove this by induction on $\nu(u)$. So, to show that tu is computable, consider an s' with $tu \succ_1 s'$. Then, because t is neutral, we must have $s' = su$ with $t \succ_1 s$ or $s' = tu'$ with $u \succ_1 u'$.

(a) If $s' = su$ with $t \succ_1 s$, then s is computable by our assumption on t and therefore s' is computable as well (by the definition of computability for function types).

(b) If $s' = tu'$ with $u \succ_1 u'$, then $s = tu'$ is computable by induction hypothesis (because $\nu(u') < \nu(u)$).

In both cases s' will be computable and therefore tu is computable by induction hypothesis applied to (iii). We conclude that t is computable.

□

LEMMA 1.13. *For computable t_1, t_2 the expression $\mathbf{p}t_1t_2$ is also computable.*

PROOF. Since we already know that computable terms are strongly normalising, we can show by induction on $\nu(t_1) + \nu(t_2)$ that $\mathbf{p}_i(\mathbf{p}t_1t_2)$ is computable.

Suppose $\nu(t_1) + \nu(t_2) = m$ and the statement is true for all numbers strictly smaller than m. If $\mathbf{p}_i(\mathbf{p}t_1t_2) \succ_1 s$, then there are two possibilities for s:

(i) $s = t_i$. In this case s is computable by assumption.

(ii) $s = \mathbf{p}_i(\mathbf{p}t_1't_2)$ with $t_1 \succ_1 t_1'$ or $s = \mathbf{p}_i(\mathbf{p}t_1t_2')$ with $t_2 \succ_1 t_2'$. In both cases s is computable by induction hypothesis.

In all cases s is computable, so $\mathbf{p}_i(\mathbf{p}t_1t_2)$ is computable by part (iii) from the previous lemma. □

LEMMA 1.14. *If for all computable t of type σ and variables x of type σ, the λ-term $s[t/x]$ is computable, then so is $\lambda x.s$.*

PROOF. We have to show that $(\lambda x.s)t$ is computable for all computable t. Since s is computable too (variables are computable and $s = s[x/x]$), we can argue by induction on $\nu(s) + \nu(t)$. The argument is now similar to the one in the previous lemma and left to the reader. □

THEOREM 1.15. *All terms are computable. In particular, all terms are strongly normalisable.*

PROOF. The idea is to prove the following (stronger) statement by induction on the structure of s:

Let s be any term (not necessarily computable) and suppose the free variables of s are among $x_1, \ldots, x_n$ of types $\sigma_1, \ldots, \sigma_n$. If $t_1, \ldots, t_n$ are computable terms of types $\sigma_1, \ldots, \sigma_n$, then $s[t_1/x_1, \ldots, t_n/x_n]$ is computable.

(The statement that all terms s are computable follows by considering $t_i = x_i$.)

- The case for variables is obvious.
- The computability of combinators $\mathbf{p}_0$ and $\mathbf{p}_1$ is immediate from the definition, while that of $\mathbf{p}$ is immediate from Lemma 1.13.
- If $s = uv$, then, by induction hypothesis, $u[\underline{t}/\underline{x}]$ and $v[\underline{t}/\underline{x}]$ are computable. From this and the definition of computability for arrow types, it follows that $s[\underline{t}/\underline{x}] = u[\underline{t}/\underline{x}]v[\underline{t}/\underline{x}]$ is computable.
- If $s = \lambda y.v$, then, by induction hypothesis, $v[\underline{t}/\underline{x}, u/y]$ is computable for all computable u. But then the previous lemma tells us that $s[\underline{t}/\underline{x}] = \lambda y.v[\underline{t}/\underline{x}]$ is computable.

□

2. Term assignments

In this section we use the ideas from the previous section to show that a fragment of intuitionistical natural deduction is strongly normalising with respect to the reduction rules from the previous chapter. The fragment we will consider is that of conjuction and implication (no disjunction) and we will also ignore the ex falso rule.

The idea is to assign to every formula in every natural deduction proof in this fragment a term from the typed lambda calculus and do this in such a way that if one applies a reduction step to the natural deduction proof one can track this by applying one or several reduction steps applied to the term assigned to the conclusion. Then strong normalisation for reduction on natural deduction proofs follows from strong normalisation for the typed lambda calculus.

Consider P, the set of propositional variables, and types over P. Then we can define by induction over formulas the *type* of that formula:

(1) The type of p is p itself.
(2) If the type φ is σ and the type of ψ is τ, then the type of $\varphi \wedge \psi$ is $\sigma \times \tau$ and the type of $\varphi \to \psi$ is $\sigma \to \tau$.

Now consider intuitionistic natural deduction proofs without ex falso and disjunction and we will decorate every formula φ in the proof tree with a term t from the typled lambda calculus having the type of φ. Let us define decorated natural deduction trees as follows.

0. If x is a variable having the type of φ, then $x\colon\varphi$ is a decorated proof tree, with uncancelled assumption and conclusion $x\colon\varphi$.

1a. If $\mathcal{D}_1$ is a decorated proof tree with conclusion $t_1\colon\varphi_1$ and $\mathcal{D}_2$ is a decorated proof tree with conclusion $t_2\colon\varphi_2$, then also

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ t_1\colon\varphi_1 & t_2\colon\varphi_2\end{array}}{\mathbf{p}t_1t_2\colon\varphi_1\wedge\varphi_2}$$

is a decorated proof tree.

1b. If $\mathcal{D}$ is a decorated proof tree with conclusion $t\colon\varphi\wedge\psi$, then also

$$\frac{\begin{array}{c}\mathcal{D}\\ t\colon \varphi \wedge \psi\end{array}}{\mathbf{p}_0 t\colon \varphi} \quad \text{and} \quad \frac{\begin{array}{c}\mathcal{D}\\ t\colon \varphi \wedge \psi\end{array}}{\mathbf{p}_1 t\colon \psi}$$

are decorated proof trees.

2a. If $\mathcal{D}$ is a decorated proof tree with conclusion $t\colon \psi$, then also

$$\frac{\begin{array}{c}[x\colon \varphi]\\ \mathcal{D}\\ t\colon \psi\end{array}}{\lambda x.t\colon \varphi \to \psi}$$

is a decorated proof tree; here by putting a $[x\colon \varphi]$ on top of $\mathcal{D}$ we mean that *every* occurence of the assumption $x\colon \varphi$ in $\mathcal{D}$ must now be cancelled.

2b. If $\mathcal{D}_1$ is a decorated proof tree with conclusion $t\colon \varphi$ and $\mathcal{D}_2$ is a proof tree with conclusion $s\colon \varphi \to \psi$, then also

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ s\colon \varphi & t\colon \varphi \to \psi\end{array}}{st\colon \psi}$$

is a decorated proof tree.

THEOREM 2.1. *Every possible sequence of reductions on an intuitionistic natural deduction proof in the fragment without $\vee$ and without ex falso eventually terminates.*

PROOF. Imagine you have a proof tree in intuitionistic natural deduction without ex falso and disjunction. Then one may decorate it and suppose one sees $t\colon \varphi$ at the root. Then every reduction step in normalisation gives rise to a decorated proof tree with root $t'\colon \varphi$ where $t \succeq t'$ and $t \neq t'$. But since every reduction sequence in the typed lambda calculus must eventually terminate, the same must then be true for natural deduction. □