Universidade Técnica de Lisboa
Instituto Superior Técnico

λ-calculus and beyond

Luís Calhorda Cruz-Filipe

Diploma Thesis

Supervisor:
Prof. José Félix Costa

May 2001

Acknowledgments

It would be a difficult task to name all the people who helped me throughout the nine months I spent preparing this Diploma Thesis; I would like to thank them all for their contributions, which, however small they might have been, were always useful. I would also like to thank in particular the following people:

- Professor José Félix Costa for supervising this work;
- Professor Amílcar Sernadas for having originally suggested this topic for my thesis and for the occasional talks we had and the interesting ideas he put on my mind;
- Professor Cristina Sernadas for having introduced me to Computer Science and a wide variety of its subjects;
- Professor H. P. Barendregt and his group at Nijmegen University for the interesting talks we had during two days and for the bibliographic references that were so useful to me;
- Paula Gouveia for her help with the natural deduction systems for logics used throughout this work;
- João Boavida and João Rasga for suggesting some more-or-less awkward LaTeX packages that were needed for typesetting some of the work;
- Gonçalo Pereira, Manuel João Morais and Paulo Mateus for various precious bits of help with the final version;
- my colleagues Rita, João and Gonçalo for their help and support whenever completing this work seemed an impossible task;
- my mother and father for their patience.

Contents

Chapter 1

Introduction

Since the 1930's, many models of computability have been widely studied; some of them gave rise to diverse interesting and widely applicable topics in Computer Science. In this work we briefly introduce a few of these models and discuss their properties and some of their areas of application.

This work was motivated by the exercises proposed in [2][1]; we start by presenting the theoretical background on untyped λ-calculus and type theory for these problems in Chapters 2 and 4 respectively, and present in Chapter 5 the resolution of some of those exercises as examples of the properties previously enunciated. Chapter 3 is a digression through combinatory algebras, which are also related to λ-calculus, and introduces some original work in the area.

λ-calculus[2] was originally introduced in 1932 by Alonzo Church as a formal theory to deal with the concept of functional abstraction. Here notions of *abstraction*, *application* and *substitution* are formalized and freed of ambiguity.

Nowadays λ-calculus is a very wide subject of mathematics, with many different applications; a major reference on the subject is [1].

In this work we will only study those aspects of λ-calculus that are essential for the comprehension of later chapters. We will introduce λ-calculus in Chapter 2, presenting the general concepts and definitions motivated from the viewpoint of someone who is constructing a model of computation and stating some of its properties that will be useful further along.

Combinatory algebras and combinatory logic were introduced first by Schönfinkel in 1924 and independently, and with a different formulation, by Haskell Curry, in 1930. Until the late 1930's they were studied many important logicians (and mathematicians in general). Besides being useful

[1] The paper *Problems in Type Theory* by H. Barendregt.

[2] actually, *untyped* λ-calculus

in the study of logic, they also provide an interesting model of computation where variables are not needed.

Combinatory logic can be presented as an equational theory (as is done, for example, in [1] and [5]) or as the study of a class of properties of combinatory algebras. The first approach, which is due to Curry, is usually followed when focusing in the relationship with λ-calculus; the second one, which we will follow, is more general and directed to the study of the properties of combinatory algebras in themselves. When looked at from this point of view, combinatory algebras become an intuitive and very powerful model of computation which is more general than untyped λ-calculus.

In Chapter 3 we introduce combinatory algebras. We explore them in some detail, proving in particular that they provide a model of computation in which every partial recursive function can be represented in some way (which, by the Church thesis, implies that every effective procedure is definable). In order to do this, we adapt the proof in [1] to the more general framework of combinatory algebras, making use of many constructions from [7] and [5]. In spite of this, our main result is not to be found among the major works published in the subject.

We end this chapter relating combinatory algebras to λ-calculus; we show that Λ is a combinatory algebra, hence inheriting all properties of these structures, and briefly debate the question of interpreting an arbitrary combinatory algebra in Λ.

In Chapter 4 we explain why untyped λ-calculus is not powerful enough for some applications, and show how this limitations can be solved. Following the same line of reasoning as in [3], we introduce the typed λ-calculus systems $\lambda \rightarrow$ and $\lambda 2$, due to Church, and examine some of the properties of these systems. We then show how to generalize the ideas behind the constructions of these systems to more abstract structures—Pure Type Systems—and present properties of these systems that make them objects of theoretical interest as well as the mathematical background for subjects such as automatic proof-checking.

In Chapter 5 we present and comment several examples that illustrate the good properties of these systems. These examples are taken from the exercises proposed in [2], most of them also appearing (either as examples or as proposed exercises) in [3]. In Section 5.2 we show how several logics with their deductive systems can be coded in Pure Type Systems, and in Section 5.3 we present a few examples of how we can, in a way that is similar to what we did in combinatory algebras, represent computable functions.

Chapter 2

Untyped λ-calculus

2.1 General Concepts

In this section, an introduction to λ-calculus will be given, containing the basic definitions and results that will be needed throughout.

We begin by specifying the language with which we will be working.

DEFINITION 2.1.1 An *alphabet* is a non-empty set. $\square$

This definition is perfectly general; however, in the context of λ-calculus we need to specify our alphabet in more detail.

DEFINITION 2.1.2 The *alphabet* for the λ-calculus is composed by:

i. a denumerable sequence $\{v_i\}_{i\in\mathbb{N}}$;

ii. the special symbol λ;

iii. punctuation marks $.$, $($, $)$;

The elements v_i are called *variables*; we denote arbitrary variables by italic lowercase letters: $x, y, z, \ldots$ $\square$

We also allow some special symbols, apart from the variables, whose meaning we might want to specify and use. These symbols will usually be given as forming a set C, called the *context*.

DEFINITION 2.1.3 The set of λ-*terms* over a context C, $\Lambda(C)$ is defined inductively as follows:

i. $v_i \in \Lambda(C)$, for all $i \in \mathbb{N}$;

ii. $C \subseteq \Lambda(C)$;

iii. if $M, N \in \Lambda(C)$, then $(MN) \in \Lambda(C)$;

iv. if $M \in \Lambda(C)$, then $(\lambda v_i.M) \in \Lambda(C)$.

We denote generic λ-terms by $M, N, \ldots$. □

The operation defined on λ-terms that assigns to arbitrary M, N the term (MN) is called *application*; the operation that assigns to an arbitrary λ-term M and to an arbitrary variable x the term $\lambda x.M$ is called *abstraction*. Intuitively, λ-terms represent functions which receive other λ-terms as arguments; an abstraction term $\lambda x.M$ represents in some manner a function with an argument x; an application term (MN) denotes the function M applied to the argument N (hence the names). In the context of an application MN, the subterm M is sometimes referred to as the *operator* and the subterm N as the *argument*.

Application is by convention associative to the left, which means that the λ-term $((\ldots((M_1M_2)M_3)\ldots)M_n)$ is simply written down as $M_1M_2\ldots M_n$. Likewise, the term $(\lambda x_1.(\lambda x_2.\ldots(\lambda x_k.M)\ldots))$ is usually written simply as $\lambda x_1\ldots x_k.M$ or (taking $\overrightarrow{x}$ as the sequence $x_1\ldots x_k$) as $\lambda\overrightarrow{x}.M$.

The outermost parenthesis in a λ-term are usually omitted.

When the context C is empty we denote the set of λ-terms over C simply by Λ.

As is usual in logics, we need to define the free variables of a λ-term.

Definition 2.1.4 Let M be a λ-term. The set of *free variables* of M, $FV(M)$, is defined inductively by

i. $FV(x) = \{x\}$, where x is an arbitrary variable;

ii. $FV(c) = \{\}$, if $c \in C$;

iii. $FV(MN) = FV(M) \cup FV(N)$;

iv. $FV(\lambda x.M) = FV(M) \setminus \{x\}$.

An occurrence of a variable x in a λ-term M is called *free* if $x \in FV(M)$ and *bound* otherwise. A term in which all variable occurrences are bound is called *closed*.

The set of all closed λ-terms is denoted by $\Lambda^0(C)$ (or simply by Λ^0, if C is empty). □

Another important concept, also usual in logics, is the concept of substitution. Intuitively, substituting the free variable x for the term L in M is simultaneously replacing **all** free occurrences of x in M by L; however, care must be taken that free variables in L do not become bound.

DEFINITION 2.1.5 Let L, M be λ-terms and x an arbitrary variable. The *substitution* of x for L in M, denoted by $M[x := L]$, is defined inductively by:

i. $x[x := L] = L$;

ii. $M[x := L] = M$, if $x \notin FV(M)$;

iii. $(PQ)[x := L] = (P[x := L])(Q[x := L])$;

iv. $(\lambda y.P)[x := L] = \lambda y.P[x := L]$, if $x \in FV(\lambda y.P)$[1] and $y \notin FV(L)$;

v. $(\lambda y.P)[x := L] = \lambda z.(P[y := z])[x := L]$, if $x \in FV(\lambda y.P), y \in FV(L)$ and z is a fresh variable not occurring (free or bound) in L or P[2]. $\square$

We will omit parenthesis whenever it is clear from the context to which term a substitution is being applied. For instance, in the last item of the last definition we will usually write $\lambda z.P[y := z][x := L]$ instead of $\lambda z.(P[y := z])[x := L]$. The operation of substitution is supposed to have higher priority than term-forming operations, so that we will not omit parenthesis in $(\lambda z.P[y := z])[x := L]$, for example.

The following result, which comes in handy whenever we are dealing with substitution, is easily proved by structural induction:

LEMMA 2.1.6 Let L, M, N be λ-terms and x, y be variables. Then $M[x := L][y := N] \equiv M[y := N[x := L]][x := L]$. $\square$

2.2 Reduction

In classical λ-calculus terms are related by different notions of reduction. Intuitively, for a given reduction R, when we say that a term M R-reduces to N we mean that when we perform a computation starting with M we eventually get to N. According to the intended meaning of "computation" we can specify different notions of reduction.

[1] and therefore x and y are different variables

[2] we can make this more precise by requiring z to be the variable v_i with the least i such that v_i does not occur in P and L

Definition 2.2.1 A binary relation on $\Lambda(C)$ is called a *notion of reduction*. □

However, it turns out almost immediately that this definition is too general to be interesting; the following concept applies to binary relations possessing some useful properties.

Definition 2.2.2 A notion of reduction $\to$ is called a *reduction schema* iff it satisfies the following rules, presented here in natural deduction style:

$$\frac{M \to N}{ML \to NL} \quad \text{(right congruence)}$$

$$\frac{M \to N}{LM \to LN} \quad \text{(left congruence)}$$

$$\frac{M \to N}{\lambda x.M \to \lambda x.N} \quad \text{(abstraction)}$$

□

Definition 2.2.1 turns out to be useful, however, as it is very simple to generate reduction schemata from arbitrary notions of reduction.

Proposition 2.2.3 Let R be a notion of reduction. Then R induces the following reduction schemata:

- the *one-step R-reduction*, $\to_R$, is defined inductively by:
  1. $R \subseteq \to_R$;
  2. if $M \to_R N$ and $L \in \Lambda(C)$, then $LM \to_R LN$;
  3. if $M \to_R N$ and $L \in \Lambda(C)$, then $ML \to_R NL$;
  4. if $M \to_R N$ and x is a variable, then $\lambda x.M \to_R \lambda x.N$.
- the *R-reduction*, $\to_R^*$, is defined inductively by:
  1. $\to_R \subseteq \to_R^*$;
  2. if $M \in \Lambda(C)$, then $M \to_R^* M$;
  3. if $M \to_R^* N$ and $N \to_R^* L$, then $M \to_R^* L$.
- the *R-equality*, $=_R$, is defined inductively by:
  1. $\to_R^* \subseteq =_R$;
  2. if $M =_R N$, then $N =_R M$;

3. if $M =_R N$ and $N =_R L$, then $M =_R L$.

It is easy to see that these relations are indeed reduction schemata. $\square$

Notice that these reduction schemata can be concisely described as follows: $\rightarrow_R$ is the smallest reduction schema containing R; $\rightarrow_R^*$ is the reflexive and transitive closure of $\rightarrow_R$; $=_R$ is the equivalence relation induced by $\rightarrow_R^*$.

We also have the following result, which is easy to prove by structural induction:

Lemma 2.2.4 Let M,M',N be λ-terms, x a variable and R a notion of reduction such that $M \rightarrow_R M'$. Then $M[x := N] \rightarrow_R^* M'[x := N]$. $\square$

Three reduction schemata are specially important in classical λ-calculus.

Definition 2.2.5

1. the notion of reduction α is defined by $\alpha = \{\langle \lambda x.M, \lambda y.M[x := y]\rangle\}$;

2. the notion of reduction β is defined by $\beta = \{\langle (\lambda x.M)N, M[x := N]\rangle\}$;

3. the notion of reduction η is defined by $\eta = \{\langle \lambda x.Mx, M\rangle\}$. $\square$

These notions of reduction induce three very important reduction schemata known as, respectively, α-equivalence, β-reduction and η-reduction.

α-equivalence ($=_\alpha$), usually denoted simply by $\equiv$, is a syntactical notion useful to identify λ-terms which are, in practice, indistinguishable. It is sometimes given as a syntactical rule in the formation of terms; we prefer to present it as a reduction schema, but from now on we will identify α-equivalent terms. This convention, which is almost universally adopted, is useful for instance in defining substitution: when specifying the meaning of $(\lambda y.P)[x := L]$ we can assume that x and y denote different variables, eventually replacing $\lambda y.P$ with an α-equivalent term.

β-reduction is the most widely used notion of reduction; we can say it represents the default method of computation in λ-calculus, and expresses the intuitive meaning of an application: the term $(\lambda x.P)Q$ is interpreted as the operator $\lambda x.P$ applied to the argument Q. Intuitively, this should be the result of substituting the variable x in P by Q—giving precisely $P[x := Q]$.

β-reduction is also extremely important in typed λ-calculus, and we will be studying some of its properties in this section.

η-reduction is included here for aesthetic reasons, as it is also quite central to classical λ-calculus. We will not, however, dwell on it in what follows.

2.3 Properties of reduction

We will now proceed to define some interesting properties that reduction schemata can have, and analyze the consequences of those properties.

From now on, we will use the term *reduction* whenever it is not relevant whether we are talking about reduction schemata or notions of reduction.

DEFINITION 2.3.1 Let R be a reduction. An *R-redex* is a term M such that $M \to_R N$, for some λ-term N. $\square$

DEFINITION 2.3.2 A λ-term M is said to be *in R-normal form* iff whenever $M \to_R N$, then $M = N$. An R-normal form N is said to be *an R-normal form of M* iff $M =_R N$. $\square$

Intuitively, a term is in normal form if no more computation may be performed on it. Accordingly, if a term M has N as R-normal form, we refer to N as the *result* of the computation of M; however, in general, it is not the case that this N is unique. Nevertheless, uniqueness of the normal form comes in many interesting cases as a consequence of a stronger (and important in itself) property.

DEFINITION 2.3.3 Let $\to$ be a notion of reduction. We say that $\to$ satisfies the *diamond property*, denoted by $R \models \lozenge$, iff for all λ-terms M, P, Q, if $M \to P$ and $M \to Q$ then there exists a λ-term N such that $P \to N$ and $Q \to N$. $\square$

We can picture this property as a diagram:

$$\begin{tikzcd}
& M \arrow[dl] \arrow[dr] & \\
P \arrow[dr, dashed] & & Q \arrow[dl, dashed] \\
& N &
\end{tikzcd}$$

The meaning of this property is the following: if $\to$ has the diamond property, then there can not exist two diverging reduction sequences starting from the same term.

In general, this property is too strong to ask of a notion of reduction; however, it is useful to single out the notions of reduction that give birth to reduction schemata satisfying the diamond property.

DEFINITION 2.3.4 A notion of reduction R is said to be *Church-Rosser* (or simply CR) iff $\rightarrow_R^*$ satisfies the diamond property. $\square$

The following results are direct consequences of this definition:

PROPOSITION 2.3.5 Let R be a CR reduction. Then:

1. if N is an R-normal form of M, then $M \rightarrow_R^* N$.
2. any term can have at most one normal form. $\square$

Thus, CR reductions are the ones that correspond to our intended meaning of "computation"—namely, any computation starting with one term can have at most one output.

The most important result for our work is the following:

THEOREM 2.3.6 β-reduction is Church-Rosser. $\square$

Another reduction that is usually studied is $\beta\eta$-reduction, which is the reduction induced by the reunion of the reduction schemata β and η. The following result also holds:

THEOREM 2.3.7 $\beta\eta$-reduction is Church-Rosser. $\square$

Proof of these results can be found in Chapters 3 and 11 of [1].

Chapter 3

Combinatory Algebras

3.1 General Notions

Combinatory algebras were first introduced as a model of computation. Later on, they were shown to be strongly related to untyped λ-calculus, providing a semantics for that theory. This relationship allows us to translate some results in either theory to the other.

In this chapter we will study some of these properties proving them *within* the theory of combinatory algebras; although they were originally proved for λ-calculus, we will adapt the proofs in [1] and show that they are valid in a more general frame. In the end we will relate combinatory algebras to untyped λ-calculus.

DEFINITION 3.1.1 A *combinatory algebra* is a quadruple $\langle \mathcal{D}, \cdot, \mathbf{K}, \mathbf{S} \rangle$ satisfying:

i. $\mathcal{D}$ is a set;

ii. $\cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a binary operation;

iii. $\mathbf{K}, \mathbf{S} \in \mathcal{D}$ are such that

$$\begin{array}{lcl} & \mathbf{K} \neq \mathbf{S} & \\ \text{(K)} & (\mathbf{K} \cdot x) \cdot y = x, & \text{for all } x, y \in \mathcal{D} \\ \text{(S)} & ((\mathbf{S} \cdot x) \cdot y) \cdot z = (x \cdot z) \cdot (y \cdot z) & \text{for all } x, y, z \in \mathcal{D} \end{array}$$

The elements of $\mathcal{D}$ are called *combinators*. □

In other words, a combinatory algebra is an algebraic structure $\langle \mathcal{D}, \cdot \rangle$ with two distinguished elements $\mathbf{K}$ and $\mathbf{S}$ satisfying (K) and (S) as above. We will often refer to the combinatory algebra $\langle \mathcal{D}, \cdot, \mathbf{K}, \mathbf{S} \rangle$ simply as $\mathcal{D}$, and we will use bold typeface for generic elements of the algebra. Also, we will usually omit the symbol for the operation in the algebra $(\cdot)$, writing $\mathbf{AB}$ instead of $\mathbf{A} \cdot \mathbf{B}$. As before, we will assume left associativity.

The reason for the choice of $\mathbf{S}$ and $\mathbf{K}$ lies in the following property of combinatory algebras: any term $t(x_1, \ldots, x_n)$ with free variables among $x_1, \ldots, x_n$ can be represented in a combinatory algebra $\mathcal{D}$ in the sense that there exists an element $\mathbf{T} \in \mathcal{D}$ such that $\mathbf{T}x_1 \ldots x_n = t(x_1, \ldots, x_n)$. We will make this notion more precise in the following.

Definition 3.1.2 Let $\{x_1, \ldots, x_n\}$ be a finite set of variables. The set of *terms over* $\{x_1, \ldots, x_n\}$ is inductively defined as follows:

i. x_i is a term over $\{x_1, \ldots, x_n\}$, for every $1 \leq i \leq n$;

ii. $\mathbf{A}$ is a term over $\{x_1, \ldots, x_n\}$, for every $\mathbf{A} \in \mathcal{D}$;

iii. if t_1 and t_2 are terms over $\{x_1, \ldots, x_n\}$, then so is $(t_1 t_2)$. $\square$

Generic terms over $\{x_1, \ldots, x_n\}$ are denoted as $t(x_1, \ldots, x_n)$; a *term* is an expression that is a term over some finite set of variables $\{x_1, \ldots, x_n\}$. We will write simply t instead of $t(x_1, \ldots, x_n)$ when the underlying set $\{x_1, \ldots, x_n\}$ is irrelevant.

Definition 3.1.3 An algebraic structure $\langle \mathcal{D}, \cdot \rangle$ is *combinatory complete* iff for every term $t(x_1, \ldots, x_n)$ over $\{x_1, \ldots, x_n\}$ there is an element $\mathbf{T} \in \mathcal{D}$ such that, for all $\mathbf{A_1}, \ldots, \mathbf{A_n} \in \mathcal{D}$, $\mathbf{TA_1} \ldots \mathbf{A_n} = t(\mathbf{A_1}, \ldots, \mathbf{A_n})$, where $t(\mathbf{A_1}, \ldots, \mathbf{A_n})$ is the result of uniformly substituting each x_i by the constant $\mathbf{A_i}$ in $t(x_1, \ldots, x_n)$.

The element $\mathbf{T}$ is said to *represent* the term t. $\square$

We will now proceed towards a result that justifies our definition of combinatory algebra. Before we do this, however, we need an auxiliary result which is interesting in itself, as it offers some insight into our choice of $\mathbf{K}$ and $\mathbf{S}$.

Lemma 3.1.4 Let $\langle \mathcal{D}, \cdot, \mathbf{K}, \mathbf{S} \rangle$ be a combinatory algebra and $n \in \mathbb{N} \setminus \{0\}$. If $t(x_1, \ldots, x_n)$ is a term over $\{x_1, \ldots, x_n\}$, then there is a term $t'(x_1, \ldots, x_{n-1})$ over $\{x_1, \ldots, x_{n-1}\}$ such that:

i. the only constants occurring in t' are $\mathbf{K}$, $\mathbf{S}$ and the constants that already occur in t;

ii. $t'(x_1, \ldots, x_{n-1}) \cdot x_n = t(x_1, \ldots, x_n)$.

PROOF. We will proceed by induction on the structure of the term t.

- t is x_i, where $i \neq n$: then we can take t' to be $\mathbf{K}t$:

  $$\mathbf{K}tx_n = t$$

  in virtue of (K).

- t is a constant: again, we can take t' to be $\mathbf{K}t$:

  $$\mathbf{K}tx_n = t$$

  in virtue of (K).

- t is x_n: then we can take t' to be $\mathbf{SKK}$:

  $$\begin{array}{rcl} \mathbf{SKK}x_n & = & \mathbf{K}x_n(\mathbf{K}x_n) \\ & = & x_n \end{array}$$

  according to both (S) and (K).

- t is $(t_1 t_2)$: by induction hypothesis there exist terms t_1' and t_2' over $\{x_1, \ldots, x_{n-1}\}$ such that $t_1' x_n = t_1$ and $t_2' x_n = t_2$. Define $t' = \mathbf{S}t_1' t_2'$. Then:

  - by construction, t' is a term over $\{x_1, \ldots, x_{n-1}\}$;
  - due to our choice of t_1' and t_2', we have that

    $$\begin{array}{rcl} t'x_n & = & \mathbf{S}t_1't_2'x_n \\ & = & t_1'x_n(t_2'x_n) \\ & = & t_1 t_2 \\ & = & t \end{array}$$

As in this construction the only constants introduced are $\mathbf{K}$ and $\mathbf{S}$, this ends the proof of our lemma. $\square$

The main result is the following:

THEOREM 3.1.5 (Schönfinkel) Let $\langle \mathcal{D}, \cdot \rangle$ be an algebraic structure. Then $\langle \mathcal{D}, \cdot \rangle$ is combinatory complete iff there are elements $\mathbf{K}, \mathbf{S} \in \mathcal{D}$ such that

$$\begin{array}{rl} \mathbf{K}xy = x, & \text{for all } x, y \in \mathcal{D} \\ \mathbf{S}xyz = xz(yz), & \text{for all } x, y, z \in \mathcal{D} \end{array}$$

In other words, $\langle \mathcal{D}, \cdot \rangle$ is combinatory complete iff there are elements $\mathbf{K}, \mathbf{S} \in \mathcal{D}$ such that $\langle \mathcal{D}, \cdot, \mathbf{K}, \mathbf{S} \rangle$ is a combinatory algebra.

PROOF. We prove this result by induction on the number of variables in t[1].

Suppose t has no variables; then it can only be constructed from the elements of $\mathcal{D}$ using the operation of the algebra. Therefore t itself is an element of the algebra, and represents itself.

Suppose that t is a term over $\{x_1, \ldots, x_n\}$. By Lemma 3.1.4 there is a term t' over $\{x_1, \ldots, x_{n-1}\}$ such that $t' \cdot x_n = t$ and the only constants occurring in t' that do not already occur in t are $\mathbf{K}$ and $\mathbf{S}$, which are also elements of $\mathcal{D}$. By induction hypothesis there is an element $\mathbf{T} \in \mathcal{D}$ that represents t'; but then $\mathbf{T}\mathbf{A_1}\ldots\mathbf{A_n} = t'(\mathbf{A_1}, \ldots, \mathbf{A_{n-1}})\mathbf{A_n}$, which by definition of t' is simply t; therefore $\mathbf{T}$ also represents t.

Reciprocally, if $\mathcal{D}$ is combinatory complete, then there are elements $\mathbf{K}$ and $\mathbf{S}$ in $\mathcal{D}$ satisfying respectively (K) and (S), as x_1x_2 and $x_1x_3(x_2x_3)$ are, respectively, terms over $\{x_1, x_2\}$ and over $\{x_1, x_2, x_3\}$. $\square$

Next, we will define two important binary relations between closed terms over combinatory algebras.

DEFINITION 3.1.6 The *contraction* in a combinatory algebra $\langle \mathcal{D}, \cdot, \mathbf{S}, \mathbf{K} \rangle$, is the binary relation $\Vdash$ inductively defined by:

(Axiom)	$\mathbf{T} \Vdash \mathbf{T}$, for $\mathbf{T} \in \mathcal{D}$;
(K)	$\mathbf{K}t_1t_2 \Vdash t_1$, for all terms t_1 and t_2;
(S)	$\mathbf{S}t_1t_2t_3 \Vdash t_1t_3(t_2t_3)$, for all terms t_1, t_2 and t_3;
(Congruence)	If $t_1 \Vdash t_1'$ and $t_2 \Vdash t_2'$, then $t_1t_2 \Vdash t_1't_2'$.

The reflexive and transitive closure of $\Vdash$ is called *weak reduction* and denoted by $\rightarrow_w$. $\square$

Weak reduction is the most natural and the most important reduction in the context of combinatory algebras; as such, we will sometimes omit the adjective "weak" and write simply $\rightarrow$ whenever there is no risk of confusion. Also, we will sometimes write $t \overset{\neq}{\rightarrow} t'$ when we want to point out that $t \rightarrow t'$ and t and t' are not identical.

Although we can define weak reduction directly, we follow here the approach of Engeler (see [5]) and introduce contraction. This will make it easier to prove properties about weak reduction, as we will be able to work with sequences of contractions which are simpler to manipulate.

A term over a combinatory algebra is said to be *in normal form* whenever it is in w-normal form (recall Definition 2.3.2); similarly we say that a term has a normal form, etc..

[1]that is, the smallest n such that t is a term over $\{x_1, \ldots, x_n\}$.

Weak reduction will later on be related to β-reduction in λ-calculus. At this stage, we will just state without proof the following result:

PROPOSITION 3.1.7 Weak reduction has the diamond property. $\square$

The proof of this result can be found in [5].

3.2 Other Properties of Weak Reduction

Besides combinatory completeness, combinatory algebras enjoy lots of other interesting properties. In the following sections we will present a few of them; our aim will be proving that combinatory algebras provide a computational model in which all of Kleene's partial recursive functions are representable in a sense which we will define.

We will begin by proving some properties of weak reduction, and relate them to important properties of combinatory algebras.

DEFINITION 3.2.1 Let t, t' be terms over a combinatory algebra such that $t \Vdash t'$. We say that t contracts to t' *by contraction on the leftmost redex*, which we denote by $t \textcircled{$\Vdash$} t'$, iff:

i. t is $\mathbf{K}t_1t_2$ and t' is t_1;

ii. t is $\mathbf{S}t_1t_2t_3$ and t' is $t_1t_3(t_2t_3)$;

iii. t is t_1t_2, $t_1 \textcircled{$\Vdash$} t_1'$, $t_2 \Vdash t_2'$ and t' is $t_1't_2'$.

We will write $t \Rightarrow t'$ iff $t \to t'$ and one of the contractions in the contraction sequence from t to t' is made on the leftmost redex[2]; the relation $\Rightarrow$ is also called *leftmost (weak) reduction*.

Also, when there is need, we will write $t \not{\textcircled{$\Vdash$}} t'$ to signify that $t \Vdash t'$ but it is not the case that $t \textcircled{$\Vdash$} t'$. $\square$

Notice that contraction on the leftmost redex is persistent: if a term t is contracted, then either the leftmost redex is contracted or it can still be in the resulting term (even though some of its subterms may have changed).

We recall that a term t is in normal form iff whenever $t \to t'$, then t' is t. Also, we say that t has t' as normal form iff $t \to t'$ and t' is in normal form. The following characterizations will also be useful:

LEMMA 3.2.2 If a term t has a normal form t', then there is a sequence $t_0, \ldots, t_n$ such that:

[2] In particular, notice that if $t \Rightarrow t'$ then $t \overset{\neq}{\to} t'$.

i. t_0 is t;

ii. $t_i \textcircled{\Vdash}\, t_{i+1}$, for $0 \leq i < n$;

iii. t_n is t'.

Proof. Suppose t has t' as a normal form. Then there is a sequence $t_0, \ldots, t_n$ such that t_0 is t, t_n is t' and $t_i \Vdash t_{i+1}$.

Our strategy will be the following: in each step we will change the first contraction that is not on the leftmost redex, and show that we can still get from t_0 to t_n in n or less steps. Therefore, we can get to t_n by only making leftmost contractions.

Suppose that $t_0 \not\textcircled{\Vdash}\, t_1$, and let t be the subterm containing the leftmost redex in t_0.

We then know that t_0 is of the form $t^* t t_0^1 \ldots t_0^k$, where $k \geq 0$[3] and t^* does not contract.

There are several cases to consider:

- t is $\mathbf{K} v_0 x_0$: then t_0 is $t^*(\mathbf{K} v_0 x_0) t_0^1 \ldots t_0^k$. As contraction can not make different subterms interact, somewhere in the sequence $t_0, \ldots, t_n$ there is a term t_i such that:

  - t_j is $t^*(\mathbf{K} v_j x_j) t_j^1 \ldots t_j^k$, for $1 \leq j \leq i$;
  - $v_{j-1} \Vdash v_j$, $x_{j-1} \Vdash x_j$ and $t_{j-1}^m \Vdash t_j^m$, for $1 \leq j \leq i$ and $1 \leq m \leq k$;
  - t_{i+1} is $t^* v_i t_{i+1}^1 \ldots t_{i+1}^k$.

  If we now define t_j' as $t^* v_{j-1} t_j^1 \ldots t_j^k$, for $1 \leq j \leq i$, and t_j' as t_j for $i < j \leq n$, we obtain a sequence of n contractions from t_0 to t_n where the first contraction is on the leftmost subterm.

- t is $\mathbf{K} v_0 x_0 w_0^1 \ldots w_0^m$: analogous.

- t is $\mathbf{S} v_0 x_0 z_0$: then take t_0 is $t^*(\mathbf{S} v_0 x_0 z_0) t_0^1 \ldots t_0^k$. Again, as contraction can not make different subterms interact, somewhere in the sequence $t_0, \ldots, t_n$ there is a term t_i such that:

  - t_j is $t^*(\mathbf{S} v_j x_j z_j) t_j^1 \ldots t_j^k$, for $1 \leq j \leq i$;
  - $v_{j-1} \Vdash v_j$, $x_{j-1} \Vdash x_j$, $z_{j-1} \Vdash z_j$ and $t_{j-1}^m \Vdash t_j^m$, for $1 \leq j \leq i$ and $1 \leq m \leq k$;
  - t_{i+1} is $t^*(v_i z_i (x_i z_i)) t_{i+1}^1 \ldots t_{i+1}^k$.

[3] If $k = 0$, then t_0 is simply $t^* t$.

Once more, define t'_j as $t^*(v_{j-1}z_{j-1}(x_{j-1}z_{j-1}))t^1_j \dots t^k_j$, for $1 \leq j \leq i$, and t'_j as t_j, for $i < j \leq n$. We thus obtain a sequence of n contractions from t_0 to t_n where the first contraction is on the leftmost subterm.

- t is $\mathbf{S}v_0x_0z_0w^1_0 \dots w^k_0$: analogous.

We can now consider the reduction sequence $t'_0, \dots, t'_n$. By iteratively choosing the least i such that $t'_i \not\circledVdash t'_{i+1}$ and repeating the above process we eventually get a sequence of leftmost contractions from t_0 to t_n. $\square$

The following result is somewhat more general:

PROPOSITION 3.2.3 If a term t has a normal form, then there can be no infinite sequence $t_0, \dots, t_n, \dots$ such that:

i. t_0 is t;

ii. $t_i \Vdash t_{i+1}$, for every $i \in \mathbb{N}$;

iii. an infinite number of contractions are made on the leftmost term.

PROOF. Suppose that t has t' as a normal form; then there is a sequence $t_0, \dots, t_n$ such that t_0 is t, $t_i \Vdash t_{i+1}$ for $0 \leq t < n$ and t_n is t'. By the previous lemma we can suppose without loss of generality that $t_i \circledVdash t_{i+1}$.

Suppose that $t'_0, \dots, t'_k, \dots$ is an infinite sequence such that, for every $i \in \mathbb{N}$, $t'_i \Vdash t'_{i+1}$, where t'_0 is t, and that in this sequence an infinite number of reductions are made in the leftmost redex.

We claim that for every $0 \leq i \leq n$ there is some k_i such that $t_i \rightarrow t'_{k_i}$. We reason as follows: let k_1 be the first natural number satisfying $t'_{k_1-1} \circledVdash t'_{k_1}$; then $t_1 \rightarrow t'_{k_1}$. We show this by cases[4]:

- the leftmost redex in t_0 is $\mathbf{K}u_1u_2$: then, as in the previous proof, t_0 is $t^*(\mathbf{K}u_1u_2)t^1 \dots t^k$; also, t'_{k_1-1} is $t^*(\mathbf{K}u'_1u'_2)v^1 \dots v^k$, where $u_1 \rightarrow u'_1$, $u_2 \rightarrow u'_2$ and $t^i \rightarrow v^i$, and t'_{k_1} is $t^*u'_1v^1 \dots v^k$. Also t_1 is $t^*u_1t^1 \dots t^k$, and as $\rightarrow$ is a congruence we have $t_1 \rightarrow t'_{k_1}$.

- the leftmost redex in t_0 is $\mathbf{K}u_1u_2 \dots u_k$: analogous.

- the leftmost redex in t_0 is $\mathbf{S}u_1u_2u_3$: then t_0 is $t^*(\mathbf{S}u_1u_2u_3)t^1 \dots t^k$, whence t'_{k_1-1} is $t^*(\mathbf{S}u'_1u'_2u'_3)v^1 \dots v^k$ with $u_1 \rightarrow u'_1$, $u_2 \rightarrow u'_2$, $u_3 \rightarrow u'_3$ and $t^i \rightarrow v^i$, and t'_{k_1} is $t^*(u'_1u'_3(u'_2u'_3))v^1 \dots v^k$. Also t_1 must be $t^*(u_1u_3(u_2u_3))t^1 \dots t^k$, and as $\rightarrow$ is a congruence relation we can conclude that $t_1 \rightarrow t'_{k_1}$.

[4]In all cases, the relevant step is that contractions cannot make different subterms of a redex either interact or disappear.

- the leftmost redex in t_0 is $\mathbf{S}u_1u_2u_3\dots u_k$: analogous.

We can now consider the following two sequences of contractions starting from t_1: the sequence $t_1,\dots,t_n$, where all contractions are made on the leftmost subterm, and the sequence $t_1,\dots,t'_{k_1},t'_{k_1+1},\dots$ where we contract from t_1 to t'_{k_1} and proceed as in the original sequence $t'_0,\dots,t'_n,\dots$.

In the second sequence, an infinite number of contractions are made on the leftmost subterm; reasoning as before, there is a term t^* in that sequence such that $t_2 \to t^*$. There are two cases to consider:

i. t^* occurs in the sequence between t_1 and t'_{k_1}: then we can take k_2 to be k_1, and we have $t_2 \to t'_{k_2}$;

ii. t^* occurs after t'_{k_1}; then t^* is in the original sequence, therefore it is t'_{k_2} for some $k_2 \geq k_1$.

Proceeding inductively, we can find numbers $k_1,\dots,k_n$ such that $t_i \to t_{k_i}$ for $1 \leq i \leq n$. In particular, there is some natural number k_n such that $t_n \to t'_{k_n}$; but t_n is t', and as such in normal form; hence t'_{k_n} cannot contract any further. This contradicts out original supposition that $t'_0,\dots,t'_k,\dots$ is an infinite sequence of contractions where an infinite number of contractions is made on the leftmost redex, as all leftmost contractions must be made in the first k_n steps. Therefore this sequence cannot exist.

Hence, if t has a normal form, then in every sequence of contractions starting from t there is only a finite number of leftmost contractions. $\square$

The following result is just a rephrasing of the last proposition:

THEOREM 3.2.4 Let t be a term. If there is an infinite sequence $t_0,\dots,t_n,\dots$ such that

i. t_0 is t;

ii. $t_i \Rightarrow t_{i+1}$

then t has no normal form. $\square$

We will now proceed to strengthen Schönfinkel's Theorem as follows:

THEOREM 3.2.5 If $\langle \mathcal{D},\cdot,\mathbf{K},\mathbf{S}\rangle$ is a combinatory algebra, then every term t over $\{x_1,\dots,x_n\}$, with $n>0$, can be represented by an element $\mathbf{T}\in\mathcal{D}$ such that, for every $\mathbf{A_1},\dots,\mathbf{A_n}\in\mathcal{D}$, $\mathbf{TA_1}\dots\mathbf{A_n} \Rightarrow t(\mathbf{A_1},\dots,\mathbf{A_n})$.

Proof. In order to prove this result, we will have to look at the original proof of Lemma 3.1.4, which we needed for our inductive proof of Theorem 3.1.5.

We start by proving that, if $t(x_1, \ldots, x_n)$ is a term over $\{x_1, \ldots, x_n\}$, then there is a term $t'(x_1, \ldots, x_{n-1})$ over $\{x_1, \ldots, x_{n-1}\}$ such that $t'(x_1, \ldots, x_{n-1}) \cdot x_n \Rightarrow t(x_1, \ldots, x_n)$. We show that we can take t' as before:

- t is $\mathbf{T} \in \mathcal{D}$ (eventually $\mathbf{K}$ or $\mathbf{S}$) or x_i, with $i \neq n$: then t' is $\mathbf{K}t$ and

$$\mathbf{K}tx_n \Vdash t.$$

- t is x_n: then t' is $\mathbf{SKK}$, and

$$\begin{aligned} \mathbf{SKK}x_n &\Vdash \mathbf{K}x_n(\mathbf{K}x_n) \\ &\Vdash x_n \end{aligned}$$

- t is t_1t_2: then by induction hypothesis there are terms t'_1 and t'_2 over $\{x_1, \ldots, x_{n-1}\}$ such that $t'_i(x_1, \ldots, x_{n-1}) \cdot x_n \Rightarrow t_i(x_1, \ldots, x_n)$, where $i = 1, 2$. Again we take t' as $\mathbf{S}t'_1t'_2$:

$$\begin{aligned} \mathbf{S}t'_1t'_2x_n &\Vdash t'_1x_n(t'_2x_n) \\ &\Rightarrow t_1(t'_2x_n) \\ &\rightarrow t_1t_2 \end{aligned}$$

  Notice that none of the contractions in the last reduction is in the leftmost redex; however, we still have $t'x_n \Rightarrow t$, as intended.

That proves the generalization of Lemma 3.1.4; to prove our result, we proceed as before: from $t(x_1, \ldots, x_n)$ we construct a term $t^1(x_1, \ldots, x_{n-1})$ as above; from this we obtain $t^2(x_1, \ldots, x_{n-2})$, until we find a term $t^n = \mathbf{T}$ with no variables. By construction, we have, for every $\mathbf{A_1}, \ldots, \mathbf{A_n} \in \mathcal{D}$:

$$\begin{aligned} \mathbf{TA_1} \ldots \mathbf{A_n} &\Rightarrow t^{n-1}(\mathbf{A_1})\mathbf{A_2} \ldots \mathbf{A_n} \\ &\Rightarrow \ldots \\ &\Rightarrow t^1(\mathbf{A_1}, \ldots, \mathbf{A_{n-1}})\mathbf{A_n} \\ &\Rightarrow t(\mathbf{A_1}, \ldots, \mathbf{A_n}) \end{aligned}$$

from which we can conclude that not only $\mathbf{T}$ represents t, but also that for every $\mathbf{A_1}, \ldots, \mathbf{A_n} \in \mathcal{D}$ we have $\mathbf{TA_1} \ldots \mathbf{A_n} \Rightarrow t(\mathbf{A_1}, \ldots, \mathbf{A_n})$. $\square$

This theorem will be useful when we want to define a term representing a partial recursive function. Using it together with Proposition 3.2.4, we will be able to prove that whenever the original function applied to some arguments is undefined, the term that represents it applied to some representation of its arguments will not have a normal form—we will be able to find an infinite sequence of contractions where an infinite number of them are made on the leftmost term.

Before we can start representing functions, we still need another result.

Definition 3.2.6 Let $\langle \mathcal{D}, \cdot, \mathbf{K}, \mathbf{S} \rangle$ be a combinatory algebra. A *combinatory equation* in $\mathcal{D}$ is an expression of the form $Tx_2 \ldots x_n = t(T, x_2, \ldots, x_n)$, where $t(x_1, \ldots, x_n)$ is a term over $\{x_1, \ldots, x_n\}$. $\square$

Proposition 3.2.7 Every combinatory equation in a combinatory algebra has a solution; that is, if $Tx_2 \ldots x_n = t(T, x_2, \ldots, x_n)$ is a combinatory equation, then there is an element $\mathbf{T} \in \mathcal{D}$ such that, for every $\mathbf{A_2}, \ldots, \mathbf{A_n} \in \mathcal{D}$,

$$\mathbf{TA_2} \ldots \mathbf{A_n} = t(\mathbf{T}, \mathbf{A_2}, \ldots, \mathbf{A_n}).$$

Furthermore, $\mathbf{TA_2} \ldots \mathbf{A_n} \Rightarrow t(\mathbf{T}, \mathbf{A_2}, \ldots, \mathbf{A_n})$.

Proof. Let $Tx_1 \ldots x_n = t(T, x_2, \ldots, x_n)$ be a combinatory equation, and define $t'(x_1, \ldots, x_n)$ as $t(x_1 x_1, \ldots, x_n)$. By Theorem 3.2.5 there is an element $\mathbf{T}' \in \mathcal{D}$ such that

$$\mathbf{T}'\mathbf{A_1} \ldots \mathbf{A_n} \Rightarrow t'(\mathbf{A_1}, \ldots, \mathbf{A_n}).$$

Take $\mathbf{T}$ as $\mathbf{T}'\mathbf{T}'$. Then we have:

$$\begin{aligned}
\mathbf{TA_2} \ldots \mathbf{A_n} &= \mathbf{T}'\mathbf{T}'\mathbf{A_2} \ldots \mathbf{A_n} \\
&\Rightarrow t'(\mathbf{T}', \mathbf{A_2}, \ldots, \mathbf{A_n}) \\
&= t(\mathbf{T}'\mathbf{T}', \mathbf{A_2}, \ldots, \mathbf{A_n}) \\
&= t(\mathbf{T}, \mathbf{A_2}, \ldots, \mathbf{A_n})
\end{aligned}$$

as intended. $\square$

3.3 Definability and Computability

We will now proceed to prove computational completeness of combinatory algebras. We first introduce some auxiliary notions, namely a representation of natural numbers, and analyse some of their properties[5].

The results presented herein are closely related to those in Chapter 8 of [1]; however, as we are working within a combinatory algebra, many of

[5] This representation is not the same one we will be using when we work in type theory; however, it is a convenient one to use in combinatory algebras.

the proofs have to be adapted to our case. Most of the hard work, however, was already done in the previous section, where we showed some results for which, although there are equivalent results in λ-calculus, the corresponding proof can not be readily adapted.

Throughout this section, we will assume we are working in a combinatory algebra $\langle \mathcal{D}, \cdot, \mathbf{K}, \mathbf{S} \rangle$. To simplify notation, we begin by naming some elements in $\mathcal{D}$.

DEFINITION 3.3.1 According to Theorem 3.2.5, the following elements of $\mathcal{D}$ exist:

- $\mathbf{I}$ such that $\mathbf{I}x \Rightarrow x$; in particular, we will take $\mathbf{I} \stackrel{\text{def}}{=} \mathbf{SKK}$;
- $\mathbf{T}$ such that $\mathbf{T}xy \Rightarrow yx$;
- $\mathbf{V}$ such that $\mathbf{V}xyz \Rightarrow zxy$.

The names for these combinators are traditional and of standard use in combinatory algebras. $\square$

DEFINITION 3.3.2 The *numerals* in $\mathcal{D}$ are inductively defined as follows:

- $\ulcorner 0 \urcorner$ is $\mathbf{I}$;
- $\ulcorner n+1 \urcorner$ is $\mathbf{V}(\mathbf{KI})\ulcorner n \urcorner$. $\square$

This definition of numerals is given in [7]; it is a trivial adaptation to combinatory algebras of the numeral system due to Barendregt and used in [1].[6]

The following result is easily proved by induction.

PROPOSITION 3.3.3 For every $n, m \in \mathbb{N}$, $\ulcorner n \urcorner = \ulcorner m \urcorner$ iff $n = m$. $\square$

At this point, it should be quite obvious what the meaning of "the term $\mathbf{F}$ represents the function $f : \mathbb{N}^k \to \mathbb{N}$" should be whenever f is defined: if $f(n_1, \ldots, n_k)$ is defined, we would like to have

$$\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner = \ulcorner f(n_1, \ldots, n_k) \urcorner.$$

However, at this point it is not clear what $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ should be whenever it is the case that $f(n_1, \ldots, n_k)$ is undefined. One possibility would be simply to ask for $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ not to be a numeral; unfortunately, everything does not work out too well in this case. We need another auxiliary definition before we can say what $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ should be in this case.

[6] The numeral system we choose has no influence in the validity in general of the properties we will prove; as is shown in [1], they are immediately translatable into analogous properties for any numeral system with reasonable characteristics.

DEFINITION 3.3.4 An element $\mathbf{T}$ is said to be *solvable* iff there are $k \in \mathbb{N}$ and elements $\mathbf{N_1}, \ldots, \mathbf{N_k} \in \mathcal{D}$ such that $\mathbf{TN_1} \ldots \mathbf{N_k} = \mathbf{I}$. $\square$

Notice that, as $\mathbf{I}$ is in normal form, if $\mathbf{T}$ is solvable then there are $k \in \mathbb{N}$ and $\mathbf{N_1}, \ldots, \mathbf{N_k} \in \mathcal{D}$ such that $\mathbf{TN_1} \ldots \mathbf{N_k} \to \mathbf{I}$.

The following are direct consequences of the definition:

PROPOSITION 3.3.5 If, for all $k \in \mathbb{N}$ and $\mathbf{N_1}, \ldots, \mathbf{N_k}$, the term $\mathbf{TN_1} \ldots \mathbf{N_k}$ does not have a normal form, then $\mathbf{T}$ is not solvable.

PROOF. Suppose $\mathbf{T}$ is solvable; then we can find $k \in \mathbb{N}$ and elements $\mathbf{N_1}, \ldots, \mathbf{N_k} \in \mathcal{D}$ such that $\mathbf{TN_1} \ldots \mathbf{N_k} \to \mathbf{I}$; in particular, $\mathbf{TN_1} \ldots \mathbf{N_k}$ has a normal form (namely, $\mathbf{I}$). $\square$

PROPOSITION 3.3.6 If $\mathbf{T}$ is not solvable, then for every $k \in \mathbb{N}$ and for every $\mathbf{N_1}, \ldots, \mathbf{N_k} \in \mathcal{D}$ the element $\mathbf{TN_1} \ldots \mathbf{N_k}$ is not solvable. $\square$

A not-so-trivial result is the following:

PROPOSITION 3.3.7 Let $\mathbf{T}$ and $\mathbf{T}'$ be such that $\mathbf{T} \to \mathbf{T}'$. If $\mathbf{T}'$ is not solvable, then neither is $\mathbf{T}$.

PROOF. Suppose that $\mathbf{T} \to \mathbf{T}'$ and $\mathbf{T}$ is solvable. Then there are $k \in \mathbb{N}$ and combinators $\mathbf{N_1}, \ldots, \mathbf{N_k} \in \mathcal{D}$ such that

$$\mathbf{TN_1} \ldots \mathbf{N_k} \to \mathbf{I}.$$

By hypothesis, as $\mathbf{T} \to \mathbf{T}'$, we also know that

$$\mathbf{TN_1} \ldots \mathbf{N_k} \to \mathbf{T}'\mathbf{N_1} \ldots \mathbf{N_k}$$

and, as $\to \models \Diamond$ and $\mathbf{I}$ is in normal form, we immediately conclude that $\mathbf{T}'\mathbf{N_1} \ldots \mathbf{N_k} \to \mathbf{I}$. Hence $\mathbf{T}'$ is solvable.

Therefore, if $\mathbf{T} \to \mathbf{T}'$ and $\mathbf{T}'$ is not solvable, then neither is $\mathbf{T}$. $\square$

We will now relate the concept of solvability with the concept of numeral in order to get a suitable way of representing the notion of "undefined".

PROPOSITION 3.3.8 Let $\ulcorner n \urcorner$ be a numeral. Then $\ulcorner n \urcorner$ is solvable; furthermore, $\ulcorner n \urcorner \mathbf{KII} \Rightarrow \mathbf{I}$.

PROOF. We will consider two cases:

i. $n = 0$: then $\ulcorner n \urcorner$ is $\mathbf{I}$ and we have

$$\begin{aligned} \mathbf{IKII} &\Rightarrow \mathbf{KII} \\ &\Rightarrow \mathbf{I} \end{aligned}$$

ii. $n \neq 0$: then there is an m such that $\ulcorner n \urcorner$ is $\ulcorner m+1 \urcorner = \mathbf{V}(\mathbf{KI})\ulcorner m \urcorner$ and we have

$$\begin{aligned} \mathbf{V}(\mathbf{KI})\ulcorner m \urcorner\mathbf{KII} &\Rightarrow \mathbf{K}(\mathbf{KI})\ulcorner m \urcorner\mathbf{II} \\ &\Rightarrow \mathbf{KIII} \\ &\Rightarrow \mathbf{II} \\ &\Rightarrow \mathbf{I} \end{aligned}$$

which ends our proof. □

This result motivates the following definition:

Definition 3.3.9 Let $k \in \mathbb{N} \setminus \{0\}$ and $f : \mathbb{N}^k \not\rightarrow \mathbb{N}$ be a partial function. The combinator $\mathbf{F}$ is said to *represent* f iff the following two conditions are satisfied:

i. if $f(n_1, \ldots, n_k) \downarrow$, then $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \rightarrow \ulcorner f(n_1, \ldots, n_k) \urcorner$;

ii. if $f(n_1, \ldots, n_k) \uparrow$, then $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ is not solvable. □

Notice that solvability is not decidable: if a term $\mathbf{T}$ is solvable by terms $\mathbf{A_1}, \ldots, \mathbf{A_k} \in \mathcal{D}$ then we can check this fact, but in general there is no effective way of deciding whether or not a given term is solvable and, in the affirmative case, who $\mathbf{A_1}, \ldots, \mathbf{A_k}$ should be. However, as we already proved that numerals are solvable in an uniform way, this will not be a problem for our future work.

Lemma 3.3.10 If $\mathbf{F}$ is a combinator representing the partial function $f : \mathbb{N}^k \not\rightarrow \mathbb{N}$, where $k \in \mathbb{N} \setminus \{0\}$, then:

i. if $f(n_1, \ldots, n_k) \downarrow$, then $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner\mathbf{KII} \Rightarrow \mathbf{I}$;

ii. if $f(n_1, \ldots, n_k) \uparrow$, then $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner\mathbf{KII}$ is not solvable.

Proof. Suppose $f(n_1, \ldots, n_k)$ is defined. Then, as $\mathbf{F}$ represents f, we know that $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ is a numeral, hence solvable by $\mathbf{KII}$. By Proposition 3.3.8 we have:

$$\begin{aligned} \mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner\mathbf{KII} &\rightarrow \ulcorner f(n_1, \ldots, n_k) \urcorner\mathbf{KII} \\ &\Rightarrow \mathbf{I} \end{aligned}$$

Suppose $f(n_1, \ldots, n_k)$ is undefined. Then, by hypothesis, $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ is not solvable; by Lemma 3.3.6, neither is $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}$. $\square$

We will now go on to prove computational completeness.

As is usual when working with partial functions, we use $\perp$ to represent "undefined", $f(n_1, \ldots, n_k) \uparrow$ to mean "$f(n_1, \ldots, n_k)$ is undefined" and $f(n_1, \ldots, n_k) \downarrow$ to mean "$f(n_1, \ldots, n_k)$ is defined". The notation $f : \mathbb{N}^k \not\rightarrow \mathbb{N}$ should be read "f is a *partial* function from $\mathbb{N}^k$ into $\mathbb{N}$", that is, there may be some $n_1, \ldots, n_k \in \mathbb{N}$ such that $f(n_1, \ldots, n_k) \uparrow$.

We start by defining the class of partial recursive functions:

Definition 3.3.11 The class $\mathcal{PR}$ of *partial recursive functions* is the class inductively defined by:[7]

i. the function $0 : \mathbb{N} \rightarrow \mathbb{N}$ such that, for all $n \in \mathbb{N}$, $0(n) = 0$ (the *zero* function) is in $\mathcal{PR}$;

ii. the function $Suc : \mathbb{N} \rightarrow \mathbb{N}$ such that, for all $n \in \mathbb{N}$, $Suc(n) = n + 1$ (the *successor* function) is in $\mathcal{PR}$;

iii. for every $k \in \mathbb{N} \setminus \{0\}$ and $1 \leq i \leq k$, the function $U_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$ such that, for all $n_1, \ldots, n_k \in \mathbb{N}$, $U_i^k(n_1, \ldots, n_k) = n_i$ (the *projection* function with parameters n and i) is in $\mathcal{PR}$;

iv. for every $k \in \mathbb{N} \setminus \{0\}$, if $g_i : \mathbb{N}^k \not\rightarrow \mathbb{N} \in \mathcal{PR}$ for $i \in \{1, \ldots, m\}$ and $h : \mathbb{N}^m \not\rightarrow \mathbb{N} \in \mathcal{PR}$ then so is the function $f : \mathbb{N}^k \not\rightarrow \mathbb{N}$ defined by *composition* from $g_1, \ldots, g_m$ and h such that, for all $n_1, \ldots, n_k \in \mathbb{N}$,

$$f(n_1, \ldots, n_k) = h(g_1(n_1, \ldots, n_k), \ldots, g_m(n_1, \ldots, n_k)),$$

being undefined whenever any of the evaluations in the righthandside of the above expressions are undefined;

v. for every $k \in \mathbb{N} \setminus \{0\}$, if $g : \mathbb{N}^k \not\rightarrow \mathbb{N} \in \mathcal{PR}$ and $h : \mathbb{N}^{k+2} \not\rightarrow \mathbb{N} \in \mathcal{PR}$, then so is the function $f : \mathbb{N}^{k+1} \not\rightarrow \mathbb{N}$ defined by *primitive recursion* from g and h such that, for all $n_1, \ldots, n_{k+1} \in \mathbb{N}$,

$$\begin{aligned} f(n_1, \ldots, n_k, 0) &= g(n_1, \ldots, n_k) \\ f(n_1, \ldots, n_k, m+1) &= h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)) \end{aligned}$$

being undefined whenever any of the evaluations in the righthandside of the above expressions are undefined;

[7] There are several common definitions of the class $\mathcal{PR}$ in current use. We use the one in [6] (Definition II.1.1); however, the results we will prove are independent of the precise definition used, as is discussed in that same reference.

vi. for every $k \in \mathbb{N} \setminus \{0\}$, if $g : \mathbb{N}^{k+1} \not\rightarrow \mathbb{N} \in \mathcal{PR}$, then so is the function $f : \mathbb{N}^k \not\rightarrow \mathbb{N}$ defined by *minimalization* from g such that, for all $n_1, \ldots, n_k \in \mathbb{N}$,

$$f(n_1, \ldots, n_k) = \begin{cases} & g(n_1, \ldots, n_k, m) \downarrow \text{ for all } m < m^* \\ m^* \quad \text{if} & g(n_1, \ldots, n_k, m) \neq 0 \text{ for all } m < m^* \\ & g(n_1, \ldots, n_k, m^*) = 0 \\ & \\ \perp \quad \text{otherwise} & \end{cases}$$

□

By saying that a combinatory algebra $\mathcal{D}$ is computationally complete, we mean that every partial recursive function is represented by some element of $\mathcal{D}$. In order to prove that every combinatory algebra has this property, we will appeal to a result due to Kleene, the proof of which can be found in [6].

PROPOSITION 3.3.12 Let $f \in \mathcal{PR}$. Then f can be obtained from total functions in $\mathcal{PR}$ by applying only composition and minimalization.[8] □

This will allow us to consider only functions defined by primitive recursion from total functions.

PROPOSITION 3.3.13 The basic functions (Zero, Successor and Projections) are all representable in $\mathcal{D}$.

PROOF. For each of these functions we will define a combinator that represents it. Notice that all of them are total functions (that is, everywhere defined), so we have only to verify condition (i) of Definition 3.3.9.

- *Zero*: the combinator $\mathbf{K}\ulcorner 0 \urcorner$ trivially represents the basic function Zero:

$$\mathbf{K}\ulcorner 0 \urcorner \ulcorner n \urcorner = \ulcorner 0 \urcorner = \ulcorner Zero(n) \urcorner$$

- *Successor*: the combinator $\sigma \stackrel{\text{def}}{=} \mathbf{V}(\mathbf{KI})$ represents the successor function:

$$\begin{aligned} \sigma \ulcorner n \urcorner &= \mathbf{V}(\mathbf{KI}) \ulcorner n \urcorner \\ &= \ulcorner n+1 \urcorner \\ &= \ulcorner Suc(n) \urcorner \end{aligned}$$

and, as $\rightarrow$ is reflexive, $\sigma \ulcorner n \urcorner \rightarrow \ulcorner n+1 \urcorner$.

[8] This is Theorem II.1.2 of [6], stated in a slightly different form.

- *Projections*: let $f : \mathbb{N}^k \to \mathbb{N}$ be such that $f(n_1, \ldots, n_k) = n_i$, with $k \in \mathbb{N} \setminus \{0\}$ and $1 \leq i \leq k$. By Theorem 3.2.5, as x_i is a term over $\{x_1, \ldots, x_k\}$, there is a combinator $\mathbf{F}$ such that $\mathbf{F}\mathbf{A_1} \ldots \mathbf{A_k} \Rightarrow \mathbf{A_i}$, for every $\mathbf{A_1}, \ldots, \mathbf{A_k} \in \mathcal{D}$. In particular, $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \ulcorner n_i \urcorner$.

This ends our proof. □

We will now consider functional composition; in this case we must be careful: although we can readily define composition by calling upon Schönfinkel's Theorem, this is not enough to guarantee that our resulting combinator works well whenever the function we are representing is undefined. We will use the same trick that is used in [1] to prove computational completeness of λ-calculus.

PROPOSITION 3.3.14 If $f : \mathbb{N}^k \not\rightharpoonup \mathbb{N}$ is defined by composition of the representable functions $h : \mathbb{N}^m \not\rightharpoonup \mathbb{N}$ and $g_1, \ldots, g_m : \mathbb{N}^k \not\rightharpoonup \mathbb{N}$, then f is representable.

PROOF. Let $\mathbf{G_1}, \ldots, \mathbf{G_m}$ and $\mathbf{H}$ represent, respectively, $g_1, \ldots, g_m$ and h. We would like to apply Schönfinkel's Theorem, or rather Theorem 3.2.5, to the term

$$\mathbf{H}(\mathbf{G_1}x_1 \ldots x_k) \ldots (\mathbf{G_m}x_1 \ldots x_k).$$

In fact, if all of the intervening functions are defined, all works well. The problem arises if one of them is not; for instance, if h is the first projection and g_1 is defined at $(n_1, \ldots, n_k)$, then $f(n_1, \ldots, n_k)$ will also be, regardless of whether $g_2, \ldots, g_m$ are defined or not at $(n_1, \ldots, n_k)$.

We must, then, change the definition somewhat to cover also these cases. Define the term $t(x_1, \ldots, x_k)$ as

$$(\mathbf{G_1}x_1 \ldots x_k\mathbf{KII}) \ldots (\mathbf{G_m}x_1 \ldots x_k\mathbf{KII})(\mathbf{H}(\mathbf{G_1}x_1 \ldots x_k) \ldots (\mathbf{G_m}x_1 \ldots x_k))$$

By Theorem 3.2.5 there is a combinator $\mathbf{F}$ such that, for all $\mathbf{A_1}, \ldots, \mathbf{A_k} \in \mathcal{D}$, $\mathbf{F}\mathbf{A_1} \ldots \mathbf{A_k} \Rightarrow t(\mathbf{A_1}, \ldots, \mathbf{A_k})$. We claim that $\mathbf{F}$ represents f. We proceed by cases:

i. Suppose $f(n_1, \ldots, n_k) \downarrow$. Then, in particular, both $g_i(n_1, \ldots, n_k) \downarrow$, for $1 \leq i \leq m$, and $h(g_1(n_1, \ldots, n_k), \ldots, g_m(n_1, \ldots, n_k)) \downarrow$. Let $v_i = g_i(n_1, \ldots, n_k)$. As $\mathbf{G_1}, \ldots, \mathbf{G_m}$ and $\mathbf{H}$ represent, respectively, $g_1, \ldots, g_m$ and h, we know that:

   - $\mathbf{G_i}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \ulcorner v_i \urcorner$, for $1 \leq i \leq m$;
   - $\mathbf{H}\ulcorner v_1 \urcorner \ldots \ulcorner v_m \urcorner \to \ulcorner h(v_1, \ldots, v_m) \urcorner$;

- $\mathbf{G_i}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII} \to \mathbf{I}$, by Lemma 3.3.10.

By definition of $\mathbf{I}$, we can readily conclude that

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner} \\ \to & (\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \\ & \qquad (\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \\ \Rightarrow & \mathbf{I} \ldots \mathbf{I}(\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \\ \Rightarrow & (\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \\ \to & \mathbf{H}\ulcorner g_1(n_1, \ldots, n_k) \urcorner \ldots \ulcorner g_m(n_1, \ldots, n_k) \urcorner \\ \Rightarrow & \ulcorner h(g_1(n_1, \ldots, n_k), \ldots, g_m(n_1, \ldots, n_k)) \urcorner \\ = & \ulcorner f(n_1, \ldots, n_k) \urcorner \end{array}$$

ii. $f(n_1, \ldots, n_k) \uparrow$: there are two cases to consider.

a) for some j, $g_j(n_1, \ldots, n_k) \uparrow$. Choose the least j for which that happens; then we have:

- $\mathbf{G_i}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \ulcorner g_i(n_1, \ldots, n_k) \urcorner$, for $1 \leq i < j$;
- for $1 \leq i < j$, $\mathbf{G_i}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII} \to \mathbf{I}$, as $\mathbf{G_i}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ reduces to a numeral and every numeral is solvable by $\mathbf{KII}$, in virtue of Proposition 3.3.8.
- $\mathbf{G_j}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}$ is unsolvable;

By definition of $\mathbf{F}$, we know that we have

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner} \\ \Rightarrow & (\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \\ & \qquad (\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \\ \Rightarrow & (\mathbf{G_j}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \\ & \qquad (\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \end{array}$$

in virtue of the definition of $\mathbf{I}$; but this last term is unsolvable, in virtue of Proposition 3.3.6. Therefore, $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ too is unsolvable, by Proposition 3.3.7.

b) $g_i(n_1, \ldots, n_k) \downarrow$ for $1 \leq i \leq m$, but $h(g_1(n_1, \ldots, n_k), \ldots, g_m(n_1, \ldots, n_k))$ is undefined. Then we can conclude, reasoning as in the first case, that

$$\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$$

$$
\begin{aligned}
&\Rightarrow (\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{KII}) \\
&\qquad\qquad (\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \\
&\Rightarrow \mathbf{I} \ldots \mathbf{I}(\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \\
&\Rightarrow (\mathbf{H}(\mathbf{G_1}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner) \ldots (\mathbf{G_m}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner)) \\
&\to \mathbf{H}\ulcorner g_1(n_1, \ldots, n_k) \urcorner \ldots \ulcorner g_m(n_1, \ldots, n_k) \urcorner
\end{aligned}
$$

which by hypothesis is unsolvable, as $\mathbf{H}$ represents h and we assumed that $h(g_1(n_1, \ldots, n_k), \ldots, g_m(n_1, \ldots, n_k)) \uparrow$. Again, by Proposition 3.3.7, we conclude that $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ is unsolvable.

Therefore, $\mathbf{F}$ represents f, as we originally claimed. $\square$

Before we prove the analogous results for recursion and minimalization, we need some auxiliary facts. Recall that, to define a function by primitive recursion or minimalization, we need to make tests on some or all of the arguments. The following results show how to do that, and provide a general if-then-else construction.

Definition 3.3.15 The *boolean combinators* in $\mathcal{D}$ are defined as follows:

- $\mathbf{tt}$ (true) is $\mathbf{K}$;
- $\mathbf{ff}$ (false) is $\mathbf{KI}$. $\square$

A trivial but important result, proven in [5], is the following:

Proposition 3.3.16 $\mathbf{tt} \neq \mathbf{ff}$. $\square$

Definition 3.3.17 A combinator $\mathbf{P}$ is said to *represent* the property[9] $P \subseteq \mathbb{N}^k$, $k \in \mathbb{N} \setminus \{0\}$, iff for all $n_1, \ldots, n_k \in \mathbb{N}$, if $P(n_1, \ldots, n_k)$ holds then $\mathbf{P}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \mathbf{tt}$; otherwise, $\mathbf{P}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \mathbf{ff}$. $\square$

The following is an immediate consequence of these definitions:

Lemma 3.3.18 Let $\mathbf{P}$ represent the property $P \subseteq \mathbb{N}^k$, $k \in \mathbb{N} \setminus \{0\}$, and $\mathbf{A}, \mathbf{B} \in \mathcal{D}$. Then:

- if $P(n_1, \ldots, n_k)$ holds, then $\mathbf{P}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{AB} \Rightarrow \mathbf{A}$;
- if $P(n_1, \ldots, n_k)$ doesn't hold, then $\mathbf{P}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{AB} \Rightarrow \mathbf{B}$. $\square$

[9] We implicitly identify a set of numbers with the property of belonging to that same set.

We will also need the following result:

PROPOSITION 3.3.19 The combinator $\mathbf{Z}$ defined by $\mathbf{Z} \stackrel{\text{def}}{=} \mathbf{T}\mathbf{tt}$ represents the property $\{0\}$.

PROOF. We will prove that $\mathbf{Z}\ulcorner n \urcorner$ does reduce to either $\mathbf{tt}$ or $\mathbf{ff}$, according, respectively, to whether n is 0 or not.

- Suppose $n = 0$. Then

$$\begin{array}{rcl} \mathbf{Z}\ulcorner 0 \urcorner & = & \mathbf{T}\mathbf{tt}\mathbf{I} \\ & \Rightarrow & \mathbf{I}\mathbf{tt} \\ & \Rightarrow & \mathbf{tt} \end{array}$$

- Suppose $n \geq 0$. Then

$$\begin{array}{rcl} \mathbf{Z}\ulcorner n+1 \urcorner & = & \mathbf{T}\mathbf{tt}(\mathbf{V}(\mathbf{KI})\ulcorner n \urcorner) \\ & \Rightarrow & \mathbf{V}(\mathbf{KI})\ulcorner n \urcorner \mathbf{tt} \\ & \Rightarrow & \mathbf{tt}(\mathbf{KI})\ulcorner n \urcorner \\ & = & \mathbf{K}(\mathbf{KI})\ulcorner n \urcorner \\ & \Rightarrow & \mathbf{KI} \\ & = & \mathbf{ff} \end{array}$$

Therefore, $\mathbf{Z}$ represents the property $\{0\}$. □

We will now proceed to primitive recursion. As we will show computational completeness of combinatory algebras appealing to Proposition 3.3.12, we can assume that we will only apply primitive recursion to total functions.

We start by defining $\ominus \stackrel{\text{def}}{=} \mathbf{T}\mathbf{ff}$. Although $\ominus$ doesn't represent the predecessor function, we have the following result:

PROPOSITION 3.3.20 $\ominus\ulcorner n+1 \urcorner \Rightarrow \ulcorner n \urcorner$.

PROOF. By definition,

$$\begin{array}{rcl} \ominus\ulcorner n+1 \urcorner & = & \mathbf{T}\mathbf{ff}\ulcorner n+1 \urcorner \\ & = & \mathbf{T}(\mathbf{KI})(\mathbf{V}(\mathbf{KI})\ulcorner n \urcorner) \\ & \Rightarrow & \mathbf{V}(\mathbf{KI})\ulcorner n \urcorner(\mathbf{KI}) \\ & \Rightarrow & \mathbf{KI}(\mathbf{KI})\ulcorner n \urcorner \\ & \Rightarrow & \mathbf{I}\ulcorner n \urcorner \\ & \Rightarrow & \ulcorner n \urcorner \end{array}$$

$\square$

Having made this definition, we proceed to the result we need:

PROPOSITION 3.3.21 If $f : \mathbb{N}^k \to \mathbb{N}$ is defined by recursion from representable total functions $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$, then f is representable.

PROOF. By hypothesis, there exist combinators $\mathbf{G}$ and $\mathbf{H}$ representing respectively g and h; that is, for all $n_1, \ldots, n_{k+2} \in \mathbb{N}$, we have

$$\mathbf{G}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner \to \ulcorner g(n_1, \ldots, n_k)\urcorner$$

$$\mathbf{H}\ulcorner n_1\urcorner \ldots \ulcorner n_{k+2}\urcorner \to \ulcorner h(n_1, \ldots, n_{k+2})\urcorner$$

Consider now the following combinatory equation:

$$Fx_1 \ldots x_{k+1} = \mathbf{Z}x_{k+1}(\mathbf{G}x_1 \ldots x_k)(\mathbf{H}x_1 \ldots x_k(\ominus x_{k+1})(Fx_1 \ldots x_k(\ominus x_{k+1}))).$$

As $\mathbf{Z}, \mathbf{G}$ and $\mathbf{H}$ are known, Proposition 3.2.7 guarantees the existence of $\mathbf{F}$ such that, for all $\mathbf{A_1}, \ldots, \mathbf{A_{k+1}} \in \mathcal{D}$, $\mathbf{FA_1} \ldots \mathbf{A_{k+1}}$ reduces to

$$\mathbf{ZA_{k+1}}(\mathbf{GA_1} \ldots \mathbf{A_k})(\mathbf{HA_1} \ldots \mathbf{A_k}(\ominus \mathbf{A_{k+1}})(\mathbf{FA_1} \ldots \mathbf{A_k}(\ominus \mathbf{A_{k+1}}))).$$

We will now prove by induction in n_{k+1} that $\mathbf{F}$ represents f:

- if $n_{k+1} = 0$, then

$$\begin{array}{rcl} \mathbf{F}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner\ulcorner 0\urcorner & \to & \mathbf{Z}\ulcorner 0\urcorner(\mathbf{G}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner)(\mathbf{H} \ldots) \\ & \to & \mathbf{G}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner \\ & \to & \ulcorner g(n_1, \ldots, n_k)\urcorner \\ & = & \ulcorner f(n_1, \ldots, n_k, 0)\urcorner \end{array}$$

- if $n_{k+1} = m + 1$, the induction hypothesis allows us to conclude that $\mathbf{F}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner\ulcorner m\urcorner \to \ulcorner f(n_1, \ldots, n_k, m)\urcorner$. We then have:

$$\begin{array}{l} \mathbf{F}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner\ulcorner m+1\urcorner \\ \quad \to \ \mathbf{Z}\ulcorner m+1\urcorner(\mathbf{G} \ldots) \\ \qquad\qquad (\mathbf{H}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner(\ominus\ulcorner m+1\urcorner)(\mathbf{F}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner(\ominus\ulcorner m+1\urcorner))) \\ \quad \to \ \mathbf{H}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner(\ominus\ulcorner m+1\urcorner)(\mathbf{F}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner(\ominus\ulcorner m+1\urcorner)) \\ \quad \to \ \mathbf{H}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner\ulcorner m\urcorner(\mathbf{F}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner\ulcorner m\urcorner) \\ \quad \to \ \mathbf{H}\ulcorner n_1\urcorner \ldots \ulcorner n_k\urcorner\ulcorner m\urcorner\ulcorner f(n_1, \ldots, n_k, m)\urcorner \\ \quad \to \ \ulcorner h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m))\urcorner \\ \quad = \ \ulcorner f(n_1, \ldots, n_k, m+1)\urcorner \end{array}$$

Thus, $\mathbf{F}$ represents f, as we wanted to prove. $\square$

Finally we proceed to minimalization. To simplify our proof, we will first state an auxiliary result.

LEMMA 3.3.22 Let $\mathbf{P}$ be a combinator representing the property $P \subseteq \mathbb{N}^{k+1}$, $k \in \mathbb{N} \setminus \{0\}$. Then there is a combinator $\mu_{\mathbf{P}}$ satisfying, for every $n_1, \ldots, n_k \in \mathbb{N}$:[10]

- if there are $m \in \mathbb{N}$ such that $P(n_1, \ldots, n_k, m)$ holds and m^* is the least such m, then $\mu_{\mathbf{P}} \ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \Rightarrow \ulcorner m^* \urcorner$;
- else, $\mu_{\mathbf{P}} \ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ is not solvable.

PROOF. Consider the combinatory equation

$$Ax_1 \ldots x_k y = (\mathbf{P}x_1 \ldots x_k y) y (Ax_1 \ldots x_k(\sigma y)).$$

By Proposition 3.2.7, there is a combinator $\mathbf{A} \in \mathcal{D}$ such that, for all elements $\mathbf{A_1}, \ldots, \mathbf{A_k}, \mathbf{B} \in \mathcal{D}$,

$$\mathbf{A}\mathbf{A_1} \ldots \mathbf{A_k}\mathbf{B} \Rightarrow (\mathbf{P}\mathbf{A_1} \ldots \mathbf{A_k}\mathbf{B})\mathbf{B}(\mathbf{A}\mathbf{A_1} \ldots \mathbf{A_k}(\sigma\mathbf{B})).$$

It is obvious, as $\mathbf{P}$ represents P, and according to the if-then-else construction and definition of σ, that:

i. if $P(n_1, \ldots, n_k, m)$ holds, then $\mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m \urcorner \Rightarrow \ulcorner m \urcorner$;

ii. otherwise, then $\mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m \urcorner \Rightarrow \mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m+1 \urcorner$;

By Theorem 3.2.5, we can find $\mu_{\mathbf{P}}$ such that

$$\mu_{\mathbf{P}}\mathbf{A_1} \ldots \mathbf{A_k} \Rightarrow \mathbf{A}\mathbf{A_1} \ldots \mathbf{A_k}\ulcorner 0 \urcorner.$$

Let $n_1, \ldots, n_k \in \mathbb{N}$ be given. Then it is easy to see that:

- if there is a least m^* such that $P(n_1, \ldots, n_k, m^*)$ holds, then we have

$$\begin{array}{rcl} \mu_{\mathbf{P}}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner & \Rightarrow & \mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner 0 \urcorner \\ & \Rightarrow & \mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner 1 \urcorner \\ & \Rightarrow & \ldots \\ & \Rightarrow & \mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* \urcorner \\ & \Rightarrow & \ulcorner m^* \urcorner \end{array}$$

  according to our previous remark on the relationship between $\mathbf{A}$ and P.

[10] This construction is the same that is presented [7].

- suppose that, for all $m \in \mathbb{N}$, $P(n_1, \ldots, n_k, m)$ doesn't hold. Then we know that $\mathbf{P}(n_1, \ldots, n_k, m) \to \mathbf{ff}$. We then have:

$$\begin{array}{rcl} \mu_{\mathbf{P}}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner & \Rightarrow & \mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner 0 \urcorner \\ & \Rightarrow & \mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner 1 \urcorner \\ & \Rightarrow & \ldots \\ & \Rightarrow & \mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m \urcorner \\ & \Rightarrow & \ldots \end{array}$$

  and therefore, by Theorem 3.2.4, we can conclude that $\mu_{\mathbf{P}}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ has no normal form.

  Furthermore, for all $m \in \mathbb{N}$ and $\mathbf{N_1} \ldots \mathbf{N_m} \in \mathcal{D}$, we can easily append the sequence $\mathbf{N_1} \ldots \mathbf{N_m}$ to all terms in the previous sequence, and conclude that $\mu_{\mathbf{P}}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \mathbf{N_1} \ldots \mathbf{N_m}$ has no normal form. By Proposition 3.3.5, we conclude that $\mu_{\mathbf{P}}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ is not solvable.

Thus $\mu_{\mathbf{P}}$ has the required properties. $\square$

We can finally prove our result for minimalization:

Proposition 3.3.23 If $f : \mathbb{N}^k \not\to \mathbb{N}$ is defined by minimalization from a representable function $g : \mathbb{N}^{k+1} \not\to \mathbb{N}$, then f is representable.

Proof. By hypothesis there is an element $\mathbf{G} \in \mathcal{D}$ that represents g. Define $\mathbf{P}$ as the combinator obtained by Theorem 3.2.5 from the term $\mathbf{Z}(\mathbf{G}x_1 \ldots x_{k+1})$ and take $\mathbf{F}$ to be $\mu_{\mathbf{P}}$. Take $n_1, \ldots, n_k \in \mathbb{N}$; there are several cases to consider:

- $f(n_1, \ldots, n_k) \downarrow$: in this case we know that $f(n_1, \ldots, n_k)$ is the least m such that $g(n_1, \ldots, n_k, m) = 0$; but this is precisely the least m such that $P(n_1, \ldots, n_k, m)$, where P is the property represented by $\mathbf{P}$. Therefore, by Lemma 3.3.22, $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \ulcorner m \urcorner$.

- $f(n_1, \ldots, n_k) \uparrow$: there are two cases to consider.

  - $g(n_1, \ldots, n_k, m) \uparrow$ for some $m \in \mathbb{N}$: then it is easy to see that $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \mathbf{Z}(\mathbf{G}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* \urcorner)\ulcorner m^* \urcorner(\mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* + 1 \urcorner)$, where m^* is the minimal such m and $\mathbf{A}$ is the auxiliary combinator defined in Lemma 3.3.22. Now we have:

$$\begin{array}{l} \mathbf{Z}(\mathbf{G}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* \urcorner)\ulcorner m^* \urcorner(\mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* + 1 \urcorner) \\ \quad \Rightarrow \quad \mathbf{T}\mathbf{tt}(\mathbf{G}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* \urcorner)\ulcorner m^* \urcorner(\mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* + 1 \urcorner) \\ \quad \Rightarrow \quad (\mathbf{G}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* \urcorner)\ulcorner m^* \urcorner(\mathbf{A}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \ulcorner m^* + 1 \urcorner)\mathbf{tt} \end{array}$$

and the leftmost term in the last expression is by hypothesis not solvable, hence the original one can not be so. Therefore $\mathbf{F}$ works as expected in this case.

- for all $m \in \mathbb{N}$, $g(n_1, \ldots, n_k, m) \downarrow$ but $g(n_1, \ldots, n_k, m) \neq 0$. Then, by Lemma 3.3.22, $\mathbf{F}\ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner$ is not solvable.

Therefore $\mathbf{F}$ represents f. $\square$

We summarize Propositions 3.3.13, 3.3.14, 3.3.21 and 3.3.23 in the following result:

THEOREM 3.3.24 (Computational Completeness) If $f : \mathbb{N}^k \not\to \mathbb{N}$ is a partial recursive function, then there is an element $\mathbf{F} \in \mathcal{D}$ that represents f. $\square$

It is interesting to compare the proof of these results with the equivalent proof in [1] for λ-calculus, in face of the translation results presented in the next section.

The question also arises whether every representable function is partial recursive. We believe it to be so, although we know of no proof of this fact; Engeler shows that not all functions are representable by expressing the Halting Problem in combinatory algebras. See [5] for details.

3.4 Combinatory Algebras and λ-Calculus

To conclude, we will briefly show how untyped λ-calculus with empty context and combinatory algebras can be related.

DEFINITION 3.4.1 Let $\mathcal{D}$ be a combinatory algebra generated by $\mathbf{K}$ and $\mathbf{S}$. We define the *translation* of the elements of $\mathcal{D}$ into the set of λ-terms, which we will denote by $[\cdot]_\lambda : \mathcal{D} \to \Lambda$, inductively by:

i. $[x]_\lambda = x$, for all variables x;

ii. $[\mathbf{K}]_\lambda = (\lambda xy.x)$;

iii. $[\mathbf{S}]_\lambda = (\lambda xyz.xz(yz))$;

iv. $[\mathbf{XY}]_\lambda = [\mathbf{X}]_\lambda[\mathbf{Y}]_\lambda$. $\square$

Notice that, in particular, this definition tells us that Λ is a combinatory algebra.

For the reciprocal translation, we will start by showing how to simulate abstraction in $\mathcal{D}$; we capture the idea behind the proof of Theorem 3.1.5.

We start by simulating abstraction; this is done by defining, for every variable x, an auxiliary operator $\lambda^* x$ that associates to every term t a term t' such that x does not occur in t'.

DEFINITION 3.4.2 Let x be a variable. The operator $\lambda^* x$ is defined inductively by:

i. $\lambda^* x.x = \mathbf{I}$;

ii. $\lambda^* x.t = \mathbf{K}t$ if x does not occur in t;

iii. $\lambda^* x.t_1 t_2 = \mathbf{S}(\lambda^* x.t_1)(\lambda^* x.t_2)$. □

DEFINITION 3.4.3 The *translation* of λ-terms into $\mathcal{D}$, denoted by $[\cdot]_{CL} : \Lambda \to \mathcal{D}$, is defined by:

i. $[x]_{CL} = x$, for all variables x;

ii. $[PQ]_{CL} = [P]_{CL}[Q]_{CL}$;

iii. $[\lambda x.P]_{CL} = \lambda^* x.[P]_{CL}$. □

These translations relate combinatory algebras and λ-calculus; in particular, we have the following result, which is easily proven by induction (see [1]):

PROPOSITION 3.4.4 Let t and t' be terms over $\mathcal{D}$. If $t \to_w t'$, then $t \to_\beta^* t'$. □

As a consequence, if the translation of a term is in β-normal form, then that term is in w-normal form. Unfortunately the converse is not true—hence the reason for the adjective "weak" in the reduction in combinatory algebras. For example, consider $\mathbf{SK}$. This term is in w-normal form; however, when we consider the corresponding term in Λ, we get

$$\begin{array}{rcl} [\mathbf{SK}]_\lambda & = & (\lambda xyz.xz(yz))(\lambda xy.x) \\ & \to_\beta & \lambda yz.(\lambda x'y'.x')z(yz) \\ & \to_\beta & \lambda yz.z \end{array}$$

hence $[\mathbf{SK}]_\lambda$ is not in β-normal form.

A consequence of this is that β-equality is a subset of w-equality (as defined in Proposition 2.2.3) but not reciprocally. However, it is possible to add axioms to $\mathcal{D}$ to make the relationship between both theories into a real equivalence; this is done in some detail in Chapter 7 of [1].

Chapter 4

Type Theory

4.1 Typed λ-calculus

In chapter 2 we presented what is called the *pure* (or *untyped*) λ-calculus. We will now analyze some problems that emerge when we rethink the interpretation of λ-terms and proceed to define what is generally known as *typed* λ-calculus.

When we inductively defined λ-terms, we introduced the notions of abstraction and application, corresponding to functional abstraction and functional application. However, the simple structure of λ-terms makes it impossible to specify domains of the functions or to assert, in general, any property about the result of an application. Typed λ-calculus provides a solution for this problem.

The idea behind typed λ-calculus is the following: there is a given set of *types*, which correspond to our possible domains, and each variable is assigned a type. Also we have an operation on types, denoted by $\rightarrow$, that will help us in typing terms corresponding to functional abstraction given the types of the intervening variables. For example, if x is a variable of type α, then we want the term $\lambda x.x$ to be of type $(\alpha \rightarrow \alpha)$.

There are two different ways of introducing typed λ-calculus, usually known as typing *à la Curry* and typing *à la Church*. We will consider only the second kind, as it will be the one that will lead us into type theory.

We start by defining the set of types over a given set of type variables.

Definition 4.1.1 Let $\mathbb{V}$ be a set whose elements will be called *type variables*. The set of *types* over $\mathbb{V}$ is the set $\mathbb{T}$ inductively defined by:

i. $\mathbb{V} \subseteq \mathbb{T}$;

ii. if $\alpha, \beta \in \mathbb{T}$, then $(\alpha \rightarrow \beta) \in \mathbb{T}$.

We will use greek letters to denote both arbitrary type variables and arbitrary types; this does not cause any ambiguity, since any type variable is itself a type. □

We will now consider annotated λ-terms; these correspond to associating types to the variables in the terms. Only bounded variables will have a type—however we will have to assume types for other variables too, in order to be able to associate a type with a given annotated λ-term.

Generic variables are defined as in untyped λ-calculus. As before, we will use $x, y, z, \ldots$ to denote arbitrary variables.

DEFINITION 4.1.2 The *alphabet* for the system $\lambda\!\to$ of typed λ-calculus is the union of the alphabet for untyped λ-calculus, a set of type variables $\mathbb{V}$ and the set of special symbols $\{:, \to\}$. □

By *variable* we will mean one of the variables from the alphabet for untyped λ-calculus.

DEFINITION 4.1.3 The set of *annotated λ-terms* over $\mathbb{T}$ is the set $\Lambda_{\mathbb{T}}$ inductively defined by:

i. if x is a variable, then $x \in \Lambda_{\mathbb{T}}$;

ii. if x is a variable, $\alpha \in \mathbb{T}$ and $P \in \Lambda_{\mathbb{T}}$, then $\lambda x\!:\!\alpha.P \in \Lambda_{\mathbb{T}}$;

iii. if $P, Q \in \Lambda_{\mathbb{T}}$, then $(PQ) \in \Lambda_{\mathbb{T}}$.

As before, we will use latin uppercase letters to denote arbitrary annotated λ-terms. □

We will make some conventions, as before. Abstraction and application follow the same associative rules as in pure λ-calculus; the operation $\to$ defined on types is supposed to be right associative[1], that is,

$$\alpha_1 \to \alpha_2 \to \alpha_3 \text{ denotes } (\alpha_1 \to (\alpha_2 \to \alpha_3)).$$

Furthermore, we will omit the adjectives “typed” in “typed λ-calculus”, referring explicitly to untyped λ-calculus whenever there is need, and “annotated” in “annotated λ-term”. We will even write only “term” in the last case whenever there is no danger of confusion.

In order to assign types to λ-terms, we need the notion of context. Informally, a context is a set of type statements—that is, information about the types of a set of terms. We make this more precise in the following definition:

[1]The reason for this is the known identification of functions of many variables with functions from a one-dimensional space to a function space; that is, of $f : A \times B \to C$ with $\widehat{f} : A \to (B \to C)$, where $\widehat{f}$ is such that $\widehat{f}(x)(y) = f(x, y)$.

Definition 4.1.4 A *statement* is a pair $M:\sigma$, where M is a term and $\sigma \in \mathbb{T}$. M is called the *subject* of the statement and σ is called the *predicate*. $\square$

Definition 4.1.5 A *context* is a set Γ of statements such that there are no two statements in Γ with the same subject.

If Γ is a context and $M:\sigma$ is a statement, we will often write $\Gamma, M:\sigma$ instead of $\Gamma \cup \{M:\sigma\}$. $\square$

We will now define what the type of a term is.

Definition 4.1.6 A statement $M:\sigma$ is *derivable* in the system[2] $\lambda\!\rightarrow$ from the context Γ, denoted by $\Gamma \vdash M:\sigma$, iff it can be derived according to the following rules:

$$\frac{}{\Gamma \vdash x:\alpha}\ \textbf{Axiom} \qquad\qquad \text{if } (x:\alpha) \in \Gamma$$

$$\frac{\Gamma \vdash M:(\alpha\rightarrow\beta) \quad \Gamma \vdash N:\alpha}{\Gamma \vdash (MN):\beta} \rightarrow \mathbf{E}$$

$$\frac{\Gamma, x:\alpha \vdash M:\beta}{\Gamma \vdash (\lambda x:\alpha.M) : (\alpha\rightarrow\beta)} \rightarrow \mathbf{I}$$

We say that M is *of type* α, or that M *has type* α, in context Γ iff $\Gamma \vdash M:\alpha$.

We say that M is *typable* iff there is some type α such that M is of type α; the type α is said to be *inhabited* iff there is some term M such that M is of type α. $\square$

The following result is easy to prove by structural induction:

Proposition 4.1.7 Let $A \in \Lambda_{\mathbb{T}}$ and $\alpha, \beta \in \mathbb{T}$ be such that $\vdash A:\alpha$ and $\vdash A:\beta$. Then α and β coincide. $\square$

Thus it makes sense to speak of **the** type of a term A.

As an example, we show how to derive the types of (typed λ equivalents of) **K** and **S**:

Example 4.1.8 The following is a derivation in $\lambda\!\rightarrow$ of $\vdash (\lambda x:\alpha.\lambda y:\beta.x) : (\alpha\rightarrow\beta\rightarrow\alpha)$:

$$\dfrac{\dfrac{\dfrac{}{x:\alpha, y:\beta \vdash x:\alpha}\ \mathbf{Ax}}{x:\alpha \vdash (\lambda y:\beta.x) : (\beta\rightarrow\alpha)} \rightarrow \mathbf{I}}{\vdash (\lambda x:\alpha.\lambda y:\beta.x) : (\alpha\rightarrow\beta\rightarrow\alpha)} \rightarrow \mathbf{I}$$

$\square$

[2] we will see other systems in the next section

Example 4.1.9 The following is a derivation in $\lambda\!\to$ of $\vdash (\lambda x:(\alpha\to\beta\to\gamma).\lambda y:(\alpha\to\beta).\lambda z:\alpha.xz(yz)) : (\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma$:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x,y,z \vdash x : \alpha\to\beta\to\gamma}\,\mathbf{Ax} \quad \dfrac{}{x,y,z \vdash z : \alpha}\,\mathbf{Ax}}{x:\alpha\to\beta\to\gamma, y:\alpha\to\beta, z:\alpha \vdash xz : \beta\to\gamma}\,\to\mathbf{E} \quad \dfrac{\dfrac{}{x,y,z \vdash y : \alpha\to\beta}\,\mathbf{Ax} \quad \dfrac{}{x,y,z \vdash z : \alpha}\,\mathbf{Ax}}{x:\alpha\to\beta\to\gamma, y:\alpha\to\beta, z:\alpha \vdash yz : \beta}\,\to\mathbf{E}}{x:\alpha\to\beta\to\gamma, y:\alpha\to\beta, z:\alpha \vdash xz(yz) : \gamma}\,\to\mathbf{E}}{x:\alpha\to\beta\to\gamma, y:\alpha\to\beta \vdash \lambda z:\alpha.xz(yz) : \alpha\to\gamma}\,\to\mathbf{I}}{x:\alpha\to\beta\to\gamma \vdash (\lambda y:(\alpha\to\beta).\lambda z:\alpha.xz(yz)) : (\alpha\to\beta)\to\alpha\to\gamma}\,\to\mathbf{I}}{\vdash (\lambda x:(\alpha\to\beta\to\gamma).\lambda y:(\alpha\to\beta).\lambda z:\alpha.xz(yz)) : (\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}\,\to\mathbf{I}
$$

□

Notice that, unlike in untyped λ-calculus, these terms are parameterized: for each type $\alpha, \beta, \gamma \in \mathbb{T}$, there are terms $\mathbf{K}_{\alpha,\beta} : \alpha\to\beta$ and $\mathbf{S}_{\alpha,\beta,\gamma} : (\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma$.

Also notice that these types correspond, if we read $\to$ as logical intuitionistic implication and the type variables as propositional symbols, to valid propositions in the intuitionistic propositional logic with single connective $\to$. We will explore this further later on, but at this point we would just like to point out the following: the rules for elimination and introduction of $\to$ correspond exactly, if we view types as propositions, to the rule of *Modus Ponens* and the deduction theorem, respectively. As every term can be generated from these two families of terms just by application and abstraction[3], and by typing these operations become precisely $\to$ elimination and introduction, we can conclude that the type of each typable[4] term is an intuitionistic tautology.

The reciprocal is also true: if we consider a valid intuitionistic proposition, then it can be generated from $(\alpha \Rightarrow (\beta \Rightarrow \alpha))$ and $((\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma)))$ by substitution of propositional symbols and use of *Modus Ponens* and the deduction theorem (this is just the completeness of intuitionistic propositional logic without negation). Therefore, if a proposition A is valid, there is a typed λ-term with type precisely A (replacing $\Rightarrow$ by $\to$ and viewing variables both as propositional symbols and type variables).

We will summarize this result in the following proposition:

Proposition 4.1.10 Let α be a type and let A be the corresponding proposition, when $\to$ is replaced by $\Rightarrow$ and type variables are interpreted as propositional symbols. Then the type α is inhabited iff A is a theorem of intuitionistic proposition logic. □

[3]due to combinatorial completeness of the λ-calculus

[4]As is shown on the next section, not all λ-terms can be assigned a type; that is, the map from $\Lambda_{\mathbb{T}}$ to Λ that forgets the types is not surjective.

A more detailed proof of this result can be found in [3].

Over the next sections we will extend our typed λ-calculus and further develop the relationship with logic.

Some examples illustrating this and other properties of typed λ-calculus will be given in Chapter 5; for simplicity, we will use the notation given here instead of the more general notation for Pure Type Systems that will be introduced in the next section.

4.2 Pure Type Systems

Typed λ-calculus immediately suggests a variety of generalizations. We will briefly explore one of them, motivating the introduction of Pure Type Systems.

In the end of the last section, we saw that there are typed λ-terms $\mathbf{K}_{\alpha,\beta}$ such that $\vdash \mathbf{K}_{\alpha,\beta} : \alpha \to \beta \to \alpha$, for all types $\alpha, \beta \in \mathbb{T}$.

The question naturally arises whether we cannot define a single λ-term $\mathbf{K}$ that combines all of the terms $\mathbf{K}_{\alpha,\beta}$; this term would have as type $\alpha \to \beta \to \alpha$, for every $\alpha, \beta \in \mathbb{T}$. Unfortunately the answer is no, and the reason for this is simple: the syntax we gave for typed λ-calculus only allows us to use variables of a fixed type. We can not construct terms where types are also variables.

Well, what if we generalize our system? This is possible to do, in the following way:

DEFINITION 4.2.1 The system $\lambda 2$ is defined as follows:

- *types*: given a set $\mathbb{V}$ of type variables, the set of types for $\lambda 2$ is defined inductively as follows:

  i. $\mathbb{V} \subseteq \mathbb{T}$;

  ii. if $\alpha, \beta \in \mathbb{T}$, then $(\alpha \to \beta) \in \mathbb{T}$;

  iii. if $\alpha \in \mathbb{V}$ and $\beta \in \mathbb{T}$, then $(\forall \alpha.\beta) \in \mathbb{T}$.

- *alphabet*: the alphabet for $\lambda 2$ is simply the alphabet for $\lambda\!\to$ plus the special symbols $\forall$ and Λ;

- *terms*: the set of terms in $\lambda 2$ is inductively defined by:

  i. if x is a variable, then x is a term;

  ii. if x is a variable, P is a term and $\alpha \in \mathbb{T}$, then $(\lambda x\!:\!\alpha.P)$ is a term;

  iii. if P and Q are terms, then (PQ) is a term;

iv. if P is a term and α is a type variable, then $(P\alpha)$ is a term.

v. if P is a term and α is a type variable, then $(\Lambda\alpha.P)$ is a term.

We will assume the usual conventions for parenthesis and associativity as before; Λ is supposed to be the strongest binding operator.

- *derivations*: the rules for derivation in $\lambda 2$ are the same as in $\lambda\!\to$ plus the two following extra rules:

$$\frac{\Gamma \vdash P : M}{\Gamma \vdash (\Lambda\alpha.P) : (\forall\alpha.M)}\ \forall\mathbf{I} \quad \text{if } \alpha \text{ is not free in } \Gamma$$

$$\frac{\Gamma \vdash P : (\forall\alpha.M)}{\Gamma \vdash PA : M[\alpha := A]}\ \forall\mathbf{E} \quad \text{if } A \in \mathbb{T}$$

As before, we say that M is *of type* α, or that M *has type* α, in context Γ iff $\Gamma \vdash M : \alpha$. □

These new rules, and the terms they type, are interpreted as follows: a term $(\Lambda\alpha.P)$ parameterizes the term P (which is of a type eventually depending on α) in this type variable. Thus, in this system there are, for example, terms $\mathbf{K}$ and $\mathbf{S}$ such that, for all types α, β and γ, we have

$$\vdash \mathbf{K}\alpha\beta : (\alpha \to \beta \to \alpha)$$

and

$$\vdash \mathbf{S}\alpha\beta\gamma : ((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma).$$

We present a derivation of these facts; remember that, as $\lambda\!\to$ is a subsystem of $\lambda 2$, we can assume as proved that $\vdash \mathbf{K}_{\alpha,\beta} : (\alpha \to \beta \to \alpha)$ and $\vdash \mathbf{S}_{\alpha,\beta,\gamma} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$.

Example 4.2.2 The following is a derivation of type for a term $\mathbf{K}$ such that $\vdash \mathbf{K}\alpha\beta : (\alpha \to \beta \to \alpha)$:

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{K}_{\alpha,\beta} : (\alpha \to \beta \to \alpha)}\ \mathbf{Th}}{\vdash \Lambda\beta.\mathbf{K}_{\alpha,\beta} : \forall\beta.(\alpha \to \beta \to \alpha)}\ \forall\ \mathbf{I}}{\vdash \Lambda\alpha.\Lambda\beta.\mathbf{K}_{\alpha,\beta} : \forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha)}\ \forall\ \mathbf{I}$$

□

EXAMPLE 4.2.3 Similarly, we present a derivation of the type of a term $\mathbf{S}$ satisfying $\vdash \mathbf{S}\alpha\beta\gamma : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash \mathbf{S}_{\alpha,\beta,\gamma} : ((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma)}\ \mathbf{Th}}{\vdash \Lambda\gamma.\mathbf{S}_{\alpha,\beta,\gamma} : \forall\gamma.((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma)}\ \forall\,\mathbf{I}}{\vdash \Lambda\beta.\Lambda\gamma.\mathbf{S}_{\alpha,\beta,\gamma} : \forall\beta.\forall\gamma.((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma)}\ \forall\,\mathbf{I}}{\vdash \Lambda\alpha.\Lambda\beta.\Lambda\gamma.\mathbf{S}_{\alpha,\beta,\gamma} : \forall\alpha.\forall\beta.\forall\gamma.((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma)}\ \forall\,\mathbf{I}$$

□

In other words, $\mathbf{K}$ is simply $\Lambda\alpha.\Lambda\beta.\mathbf{K}_{\alpha,\beta}$ and $\mathbf{S}$ is $\Lambda\alpha.\Lambda\beta.\Lambda\gamma.\mathbf{S}_{\alpha,\beta,\gamma}$. Notice how the dependency on types which $\mathbf{K}_{\alpha,\beta}$ and $\mathbf{S}_{\alpha,\beta,\gamma}$ had has been abstracted at term level.

In Chapter 5 we will present more examples of terms that can be typed in $\lambda 2$.

As we did for $\lambda\!\to$, we could now relate $\lambda 2$ to second-order intuitionistic propositional logic; in fact, for these two systems an equivalent of Proposition 4.1.10 can easily be proven. We will not do this, however, because we will now generalize our construction of $\lambda 2$ to arbitrary typing systems which we will be able to relate to various logics in an uniform way.

DEFINITION 4.2.4 The *alphabet* for a Pure Type System contains:

- an infinite sequence $v_1, v_2, \ldots$ of *variables*;
- a set $\mathcal{C}$ of *constants*;
- the special symbols λ, Π;
- punctuation marks (,),:,.; □

Arbitrary variables will be denoted by lowercase latin or greek letters $x, y, z, \ldots, \alpha, \beta, \gamma, \ldots$; the distinction between latin and greek letters is for clarification only.

We will now define what will correspond to terms in a Pure Type System:

DEFINITION 4.2.5 The set of *pseudo-terms* is inductively defined as follows:

i. all elements of $\mathcal{C}$ are pseudo-terms;

ii. if P and Q are pseudo-terms, then so is (PQ);

iii. if P and Q are pseudo-terms and x is a variable, then $(\lambda x.P)$ and $(\Pi x.P)$ are pseudo-terms.

We will denote arbitrary pseudo-terms by capital latin letters $A, B, \ldots$ □

The reason for the adjective "pseudo" is the following: unlike in λ-calculus, we need to be able to construct expressions that are not terms (in the sense that we will not be able to assign them a type) but that may appear in the definition of terms. Another way to look at it is the following: pseudo-terms are *possible* terms to which a type may be assigned by a Pure Type System.

As for the structure of terms, the term (PQ) is interpreted as before as the application of the term P to the term Q. The operators λ and Π are called *abstraction* and *product* operators, respectively; the abstraction operator works just as the λ in $\lambda\!\rightarrow$ or λ and Λ in $\lambda 2$; the product operator corresponds to $\rightarrow$ in $\lambda\!\rightarrow$ and $\forall$ in $\lambda 2$—it provides types for terms generated through abstraction. The meaning of these operators will become clearer after we present the type assignment rules in Definition 4.2.8.

We can define reductions in the set of pseudo-terms just as we did for λ-calculus—binary relations that are congruences with respect to the pseudo-term formation operations.

We are now in a position to make our main definition:

Definition 4.2.6 A *specification* of a Pure Type System over a set of constants $\mathcal{C}$ is a triple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ where:

- $\mathcal{S} \subseteq \mathcal{C}$ is a set, whose elements are called *sorts*;
- $\mathcal{A}$ is a set of statements $c : s$, called *axioms*, such that $c \in \mathcal{C}$ and $s \in \mathcal{S}$;
- $\mathcal{R} \subseteq \mathcal{S}^3$ is a set whose elements are called *rules.* □

Intuitively, the set $\mathcal{C}$ contains the entities with which we will work, namely the basic elements of our terms and types. Among these, some (the elements of $\mathcal{S}$) are special types, in that a term A of type $s \in \mathcal{S}$ can itself be the type of some other term. The axioms are the terms whose type we postulate, and the rules allow us to build new terms and types from existing ones—the so called *product types*; again, this will become clearer after Definition 4.2.8.

Definition 4.2.7 A *Pure Type System* consists of a specification and a reduction in the set of pseudo-terms. □

We will usually omit the reference to the underlying set of constants and to the notion of reduction, assuming that $\mathcal{C} = \mathcal{S}$ and that the reduction in the set of pseudo-terms is the identity unless otherwise specified, as the three constituents of the specification of a Pure Type System are the ones that will be relevant when we consider derivations.

We define statements and contexts for Pure Type Systems just as we did for $\lambda\!\to$. From these, it is possible to define a notion of derivation that is more general than that defined for that system and for $\lambda 2$.

DEFINITION 4.2.8 A statement $A : B$ is said to be *derivable* from a context Γ in a Pure Type System $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ iff $\Gamma \vdash A : B$ can be derived from the following rules:

$$\frac{}{\vdash c{:}s}\ \text{Axiom} \qquad c{:}s \in \mathcal{A}$$

$$\frac{\Gamma \vdash A{:}s}{\Gamma, x{:}A \vdash x{:}A}\ \text{Start} \qquad s \in \mathcal{S},\ x \text{ fresh}$$

$$\frac{\Gamma \vdash A{:}s \quad \Gamma \vdash B{:}C}{\Gamma, x{:}A \vdash B{:}C}\ \text{Weakening} \qquad s \in \mathcal{S},\ x \text{ fresh}$$

$$\frac{\Gamma \vdash F{:}(\Pi x{:}A.B) \quad \Gamma \vdash a{:}A}{\Gamma \vdash Fa{:}B[x := a]}\ \text{Application}$$

$$\frac{\Gamma \vdash A{:}s_1 \quad \Gamma, x{:}A \vdash B{:}s_2}{\Gamma \vdash (\Pi x{:}A.B) : s_3}\ \text{Product} \qquad (s_1, s_2, s_3) \in \mathcal{R}$$

$$\frac{\Gamma, x{:}A \vdash B{:}C \quad \Gamma \vdash (\Pi x{:}A.C) : s}{\Gamma \vdash (\lambda x{:}A.B) : (\Pi x{:}A.C)}\ \text{Abstraction} \qquad s \in \mathcal{S}$$

$$\frac{\Gamma \vdash A{:}B \quad \Gamma \vdash B'{:}s}{\Gamma \vdash A{:}B'}\ \text{Conversion} \qquad s \in \mathcal{S},\ B \text{ reduces to } B'$$

$\square$

Examples of derivations within Pure Type Systems can be found in Chapter 5; here we will limit ourselves to explaining the intended meaning of each of the above rules.

The Axiom Rule is responsible for the intended meaning of the set of axioms: it allows us to derive statements in $\mathcal{A}$ from any context.

The Start Rule is useful for adding hypothesis to a context: if we have already derived $A{:}s$ in context Γ, then A is a valid type and we can assume that there is a variable x of type A.[5]

The Weakening Rule is very similar: it simply says that any statement that can be inferred from Γ can still be inferred if we add hypothesis to Γ

[5] That is, it is consistent to assume the existence of such a variable; whether a term of type A does exist or not is a different question. Notice that anything we derive from now on is dependent on the hypothesis that x is of type A.

(once again, these hypothesis must be of type $x:A$, where A is a valid type in context Γ).

We will consider Application, Product and Abstraction next. As before, the λ operator is used to define functions: if from Γ and the extra hypothesis $x:A$ we can derive $B:C$, then from Γ alone we can derive a function that for any term of type A outputs a result of type C. The difference here is that the type of the resulting term may also depend on the type of x; therefore, the operator Π is introduced with the purpose of permitting abstraction at the level of types. The Abstraction and Application rules control the type assignment for terms generated by these operations, while the Product Rule allows only for the formation of specific types. We will see later on that the expressive power of a system is highly dependent on the operations that are allowed here.

Finally, the Conversion Rule states that if two types B and B' are convertible (in the sense of Proposition 2.2.3) then any term of type B is also of type B'.

We will now briefly show how the systems $\lambda\!\to$ and $\lambda 2$ can be interpreted as Pure Type Systems. We start with $\lambda\!\to$.

Definition 4.2.9 The Pure Type System $\lambda\!\to_{PTS}$ is the Pure Type System with specification $\langle\{*,\Box\},\{*:\Box\},\{(*,*,*)\}\rangle$. $\Box$

Definition 4.2.10 We define a canonical map **Tr** from the set of types of $\lambda\!\to$ into the set of pseudo-terms of $\lambda\!\to_{PTS}$ inductively in the following way:

i. $\mathbf{Tr}(\alpha)=\alpha$, for all $\alpha\in\mathbb{V}$;

ii. $\mathbf{Tr}(\alpha\to\beta)=\Pi x:\mathbf{Tr}(\alpha).\mathbf{Tr}(\beta)$.

This translation allows us to interpret typed λ-terms as pseudo-terms in $\lambda\!\to_{PTS}$—just translate the types using **Tr**. $\Box$

This translation expresses the intended meaning of product types: a term M has type $\Pi x:A.B$ iff Mx has type $B[x:=A]$ whenever x has type A. In the case of $\lambda\!\to_{PTS}$, this is just the function space, as the limited set of available rules does not allow B to depend on x.

This translation is well defined in the following sense:

Lemma 4.2.11 If $\alpha,\beta\in\mathbb{T}$, then $\mathbf{Tr}(\alpha):*,\mathbf{Tr}(\beta):*\vdash_{\lambda\to_{PTS}}\mathbf{Tr}(\alpha\to\beta):*$.

Proof. This is a direct consequence of the Product Rule and the definition of **Tr**. $\Box$

From this lemma it is easy to prove the following result, by induction on the structure of terms:

PROPOSITION 4.2.12 If $\alpha \in \mathbb{T}$, then $\Gamma \vdash \mathbf{Tr}(\alpha) : *$, where Γ is the set of statements of the form $\gamma : *$ such that γ is a type variable in α. $\square$

The following result is now quite simple. The proof given here is not exactly the same as in [3], mainly because our definitions do not coincide exactly, but it is in the same spirit.

PROPOSITION 4.2.13 $\lambda\!\!\to$ is equivalent to $\lambda\!\!\to_{PTS}$.

PROOF. Notice that in the notion of Pure Type System there is no concept of "type variable". Therefore, we will show the above mentioned equivalence in the following sense: $\Gamma \vdash_{\lambda\to} M : \sigma$ iff $\mathbf{Tr}(\Gamma) \cup \Delta \vdash_{\lambda\to_{PTS}} M : \sigma$, where Δ is the set of assertions $\alpha : *$ such that α is a type variable occurring in σ.

We will show the equivalence between these two systems by proving that the rules of one can be derived from the rules of the other.

- We start by showing that the rules of inference in $\lambda\!\!\to$ can be derived in $\lambda\!\!\to_{PTS}$.
  - *Axiom Rule*: we will consider first the case $x : \alpha \vdash_{\lambda\to} x : \alpha$. Consider the following derivation in $\lambda\!\!\to_{PTS}$:

$$\cfrac{\cfrac{\vdash * : \square}{\alpha : * \vdash \alpha : *}\ \text{Start}}{\alpha : *, x : \alpha \vdash x : \alpha}\ \text{Start}$$

    Further elements of $\mathbf{Tr}(\Gamma)$ can now be added to the left side of the last expression by repeated use of the Weakening Rule.
  - $\to$ **I** *Rule*: Suppose that in $\lambda\!\!\to_{PTS}$ we can prove

$$\mathbf{Tr}(\Gamma), x : \mathbf{Tr}(\alpha) \vdash A : \mathbf{Tr}(\beta).$$

    By Proposition 4.2.12 we know that in this case

$$\mathbf{Tr}(\Gamma) \vdash (\Pi x : \mathbf{Tr}(\alpha).\mathbf{Tr}(\beta)) : *;$$

    hence we can conclude from the Abstraction Rule that

$$\mathbf{Tr}(\Gamma) \vdash (\lambda x : \mathbf{Tr}(\alpha).A) : (\Pi x : \mathbf{Tr}(\alpha).\mathbf{Tr}(\beta)).$$

  - $\to$ **E** *Rule*: immediate from the Application Rule.

- We will now show that application of the rules in $\lambda{\to}_{PTS}$ can be imitated by $\lambda{\to}$.
  - *Axiom Rule*: this rule simply states that $\vdash * : \Box$, and as $*$ is not the translation of a typed λ-term there is nothing to prove.
  - *Start Rule*: immediate from the Axiom Rule for $\lambda{\to}$.
  - *Weakening Rule*: Suppose that $\Gamma \vdash B : C$, where Γ is the translation of a set Δ, B is the translation of a typed λ-term P and C is the translation of a type σ, and that A is a type variable. Then we can add at every step in the derivation of $\Delta \vdash_{\lambda\to} M : \sigma$ the extra hypothesis $x : A$ to the context, still producing a valid derivation. Thus, $\Gamma, x : A \vdash B : C$ is still the translation of a valid derivation.
  - *Application Rule*: this is just the $\to$ **E** Rule for $\lambda{\to}$.
  - *Product Rule*: this rule never corresponds to a derivation in $\lambda{\to}$, as it can only produce type assignments to pseudo-terms that correspond to types in that system.
  - *Abstraction Rule*: this rule corresponds to the $\to$ **I** Rule for $\lambda{\to}$ with an extra hypothesis, therefore it is always applicable.
  - *Conversion Rule*: this rule does not add anything to the set of derivations, as we are assuming the identity reduction.

Therefore, $\lambda{\to}$ and $\lambda{\to}_{PTS}$ are equivalent. $\Box$

In view of this result, we will denote both systems simply by $\lambda{\to}$, without any ambiguity.

For $\lambda 2$, we will just present the corresponding Pure Type System and state without proof the corresponding result.

Definition 4.2.14 The Pure Type System $\lambda 2_{PTS}$ is the Pure Type System with specification $\langle\{*, \Box\}, \{* : \Box\}, \{(*, *, *), (\Box, *, *)\}\rangle$. $\Box$

Translation of types is as follows:

Definition 4.2.15 The canonical map **Tr** from the set of types of $\lambda 2$ into the set of pseudo-terms of $\lambda 2_{PTS}$ is defined inductively as follows:

i. $\mathbf{Tr}(\alpha) = \alpha$, for all $\alpha \in \mathbb{V}$;

ii. $\mathbf{Tr}(\alpha \to \beta) = \Pi x : \mathbf{Tr}(\alpha).\mathbf{Tr}(\beta)$;

iii. $\mathbf{Tr}(\forall \alpha.\beta) = \Pi \alpha : *.\mathbf{Tr}(\beta)$[6]. $\Box$

[6] Remember that in this case α must be a type variable.

As we already stated, the following result holds:

Proposition 4.2.16 $\lambda 2$ is equivalent to $\lambda 2_{PTS}$. $\square$

As we did for $\lambda\!\rightarrow$, we will use $\lambda 2$ for both systems, without any ambiguity.

4.3 The Lambda Cube

As we saw before, $\lambda 2$ generalizes $\lambda\!\rightarrow$, by permitting abstraction over type variables as well as over generic variables. As Pure Type Systems, this corresponds simply to the addition of the triple $(\square, *, *)$ to the set of rules of the specification.

Further generalization leads us to the Lambda Cube. The Lambda Cube is a set of eight Pure Type Systems obtained from $\lambda\!\rightarrow$ by adding to the specification all possible combinations of rules (s_1, s_2, s_3) such that $s_2 = s_3$. This system, described in detail in [3], has many interesting properties; we will briefly examine some of them.

Definition 4.3.1 The *Lambda Cube* consists of eight Pure Type Systems with specification $\langle\{*, \square\}, \{*:\square\}, \mathcal{R}\rangle$, where the set of rules for each system is the following:

System	Rules			
$\lambda\!\rightarrow$	$(*, *, *)$			
$\lambda 2$	$(*, *, *)$	$(\square, *, *)$		
$\lambda\underline{\omega}$	$(*, *, *)$		$(\square, \square, \square)$	
$\lambda\omega$	$(*, *, *)$	$(\square, *, *)$	$(\square, \square, \square)$	
λP	$(*, *, *)$			$(*, \square, \square)$
$\lambda P2$	$(*, *, *)$	$(\square, *, *)$		$(*, \square, \square)$
$\lambda P\underline{\omega}$	$(*, *, *)$		$(\square, \square, \square)$	$(*, \square, \square)$
$\lambda P\omega$	$(*, *, *)$	$(\square, *, *)$	$(\square, \square, \square)$	$(*, \square, \square)$

These systems are usually presented in the graphical way depicted in Figure 4.1. $\square$

The arrows denote inclusion; that is, if there is an arrow from λ_1 to λ_2, then $\Gamma \vdash_{\lambda_2} A:B$ whenever $\Gamma \vdash_{\lambda_1} A:B$.

As we already did for $\lambda\!\rightarrow$ and $\lambda 2$, we can define translation maps $[\![\cdot]\!]$ from the set of well formed formulas of an intuitionistic logical system to the set of pseudo-terms of each these systems[7]. We will not do this here; details

[7] This is known as the *propositions-as-types* interpretation of the Lambda Cube.

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda{\rightarrow} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 4.1: The Lambda Cube

can be found in [3]; we will limit ourselves to presenting the corresponding systems and stating the main result.

The eight logical systems involved in this equivalence are:

System		Logic
$\lambda{\rightarrow}$	p	propositional logic
$\lambda 2$	$p2$	second-order propositional logic
$\lambda\underline{\omega}$	$p\underline{\omega}$	weakly higher-order propositional logic
$\lambda\omega$	$p\underline{\omega}$	higher-order propositional logic
λP	P	first-order logic
$\lambda P2$	$P2$	second-order logic
$\lambda P\underline{\omega}$	$P\underline{\omega}$	weakly higher-order logic
$\lambda P\omega$	$P\underline{\omega}$	higher-order logic

These systems, as well as the details of the translation, are described in [3]. They can also be presented in a graphical way as shown in Figure 4.2.

The main result is the following:

THEOREM 4.3.2 Let A be a formula in a logic of the logical cube. Then the formula is valid iff A is inhabited in the corresponding system of the Lambda cube. □

We will give several examples of this in the next section.

$$\begin{tikzcd}
 & p\omega \arrow[rr] & & P\omega \\
p2 \arrow[ur] \arrow[rr] & & P2 \arrow[ur] & \\
 & p\underline{\omega} \arrow[uu] \arrow[rr] & & P\underline{\omega} \arrow[uu] \\
p \arrow[uu] \arrow[ur] \arrow[rr] & & P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 4.2: The cube of logical systems

Chapter 5

Problems in Type Theory

In this section we will give examples of some of the properties of type systems presented in the last section. These examples are chosen from the problems proposed in [2]. We will indicate next to each example the number by which the problem is presented in that paper.

5.1 Typed λ-Calculus

In this section, we will use the simpler notation of section 4.1 instead of the more cumbersome for Pure Type Systems. Also, we will often omit types of variables in the upper branches of proof trees whenever they can easily be understood from the context.

We start by presenting an example of a non-typable λ-term (problem 2).

EXAMPLE 5.1.1 The λ-term $\lambda x.xx$ is not the image of any typed λ-term.

PROOF. The forgetful map from $\Lambda_{\mathbf{T}}$ into Λ only forgets types; thus, if there is a typed λ-term M mapped into $\lambda x.xx$, then M has to be of the form $\lambda x{:}\alpha.xx$.

Consider a proof of $\vdash (\lambda x:\alpha.xx) : \beta$. As $(\lambda x:\alpha.xx)$ can only be typed using rule $\rightarrow \mathbf{I}$, necessarily β is $(\alpha\rightarrow\gamma)$ and $x{:}\alpha \vdash xx{:}\gamma$.

Further reasoning of the same kind leads us to the conclusion that a proof tree for $\vdash (\lambda x{:}\alpha.xx) : \beta$ must have the following structure:

$$\dfrac{\dfrac{x{:}\alpha \vdash x{:}\delta\rightarrow\gamma \quad x{:}\alpha \vdash x{:}\delta}{x{:}\alpha \vdash xx{:}\gamma}\rightarrow \mathbf{E}}{\vdash (\lambda x{:}\alpha.xx) : \alpha\rightarrow\gamma}\rightarrow \mathbf{I}$$

for some type variable δ; this implies that α, δ and $\delta \to \gamma$ coincide. But this is absurd, as no type can be a proper subexpression of itself. Therefore, M is not typable. $\square$

We will now show some examples of how proofs of valid propositions of intuitionistic logic can be represented by typed λ-terms. In the next section we will extend this to other Pure Type Systems.

Throughout this section and the next we will use the natural deduction systems for logics presented in [8]. To stress the parallelism between proofs in these systems and type assignment proofs we will use the symbol $\to$ to denote implication.

EXAMPLE 5.1.2 $(\alpha \to \alpha)$

We start with a very simple example (problem 1.1 of [2]), the same one that we gave to motivate the introduction of types. We start by presenting a proof of this formula in natural deduction.

$$\frac{\alpha^1}{\alpha \to \alpha} \to \mathbf{I}, 1$$

As explained in [8], the numbered formulas are hypothesis that are introduced at some point; some of the rules (in this case, the rule $\to \mathbf{I}$) can eliminate these hypothesis, meaning that the formulas derived later on no longer depend on them. The number next to the rule indicates which hypothesis was cancelled in this step.

The corresponding typed λ-term is $(\lambda x : \alpha . x)$; the proof is simply

$$\frac{\overline{x : \alpha \vdash x : \alpha} \ \mathbf{Ax}}{\vdash \lambda x : \alpha . x : \alpha \to \alpha} \to \mathbf{I}$$

Notice the structural similarity between the two proofs; cancellation of hypothesis 1 (α) in the natural deduction proof corresponds to the binding of the variable x (which represents α) by the λ-operator. Also, the terms on the left side at each node indicate the hypothesis that are open at that node. Thus, from the λ-term alone it is possible to reconstruct the natural deduction proof. $\square$

Just for the sake of the exposition we reproduce the proofs we already gave for $\mathbf{K}_{\alpha,\beta}$ and $\mathbf{S}_{\alpha,\beta,\gamma}$, together with the natural deduction proofs for the corresponding propositions (problems 1.2 and 1.3 of [2]).

EXAMPLE 5.1.3 $(\alpha \to (\beta \to \alpha))$

Proof in natural deduction:

$$\dfrac{\dfrac{\alpha^1}{\beta\to\alpha}\ \to\mathbf{I},2}{\alpha\to(\beta\to\alpha)}\ \to\mathbf{I},1$$

Corresponding derivation in $\lambda{\to}$:

$$\dfrac{\dfrac{\dfrac{}{x:\alpha, y:\beta\vdash x:\alpha}\ \mathbf{Ax}}{x:\alpha\vdash(\lambda y:\beta.x):(\beta\to\alpha)}\ \to\mathbf{I}}{\vdash(\lambda x:\alpha.\lambda y:\beta.x):(\alpha\to(\beta\to\alpha))}\ \to\mathbf{I}$$

□

EXAMPLE 5.1.4 $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$

Proof in natural deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha^1\quad(\alpha\to\beta)^2}{\beta}\ \to\mathbf{E}\qquad\dfrac{\alpha^1\quad(\alpha\to(\beta\to\gamma))^3}{\beta\to\gamma}\ \to\mathbf{E}}{\gamma}\ \to\mathbf{E}}{\alpha\to\gamma}\ \to\mathbf{I},1}{(\alpha\to\beta)\to(\alpha\to\gamma)}\ \to\mathbf{I},2}{(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))}\ \to\mathbf{I},3$$

Corresponding derivation in $\lambda{\to}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x,y,z\vdash y:\alpha\to\beta}\ \mathbf{Ax}\quad\dfrac{}{x,y,z\vdash z:\alpha}\ \mathbf{Ax}}{x:\alpha\to\beta\to\gamma, y:\alpha\to\beta, z:\alpha\vdash yz:\beta}\ \to\mathbf{E}\qquad\dfrac{\dfrac{}{x,y,z\vdash x:\alpha\to(\beta\to\gamma)}\ \mathbf{Ax}\quad\dfrac{}{x,y,z\vdash z:\alpha}\ \mathbf{Ax}}{x:\alpha\to(\beta\to\gamma), y:\alpha\to\beta, z:\alpha\vdash xz:\beta\to\gamma}\ \to\mathbf{E}}{x:\alpha\to(\beta\to\gamma), y:\alpha\to\beta, z:\alpha\vdash xz(yz):\gamma}\ \to\mathbf{E}}{x:\alpha\to(\beta\to\gamma), y:\alpha\to\beta\vdash\lambda z:\alpha.xz(yz):\alpha\to\gamma}\ \to\mathbf{I}}{x:\alpha\to(\beta\to\gamma)\vdash(\lambda y:(\alpha\to\beta).\lambda z:\alpha.xz(yz)):(\alpha\to\beta)\to(\alpha\to\gamma)}\ \to\mathbf{I}}{\vdash(\lambda x:(\alpha\to(\beta\to\gamma)).\lambda y:(\alpha\to\beta).\lambda z:\alpha.xz(yz)):(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))}\ \to\mathbf{I}$$

□

Finally, we will present an example of a valid tautology (problem 1.4 of [2]) and try to give some insight on the construction of the corresponding λ-term, from which we hope to reinforce the idea that this process might be simpler than directly prove the formula in natural deduction. Henceforth, we will use the conventions of associativity for $\to$ in the type assignment proofs.

EXAMPLE 5.1.5 $(\alpha \to (\alpha \to \beta)) \to (\alpha \to \beta)$

Due to the definition of $\to$, a term of type $(\alpha \to (\alpha \to \beta)) \to (\alpha \to \beta)$ must be a function with arguments $x:(\alpha\to(\alpha\to\beta))$ and $y:\alpha$ which will give an output of type β. As x is in this case itself a function that on receiving two arguments of type α returns a value of type β, it is easy to conclude that the simplest way to produce a term of type β is to evaluate x when both arguments are set to y; this produces the term xyy. Thus, the λ-term we are looking for is $\lambda x\!:\!(\alpha\to\alpha\to\beta)\lambda y\!:\!\alpha.xyy$.

We start by checking the type of this term[1]:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$x, y \vdash y\!:\!\alpha$}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$x, y \vdash x\!:\!(\alpha\to\alpha\to\beta)$}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$x, y \vdash y\!:\!\alpha$}
\RightLabel{$\to\mathbf{E}$}
\BinaryInfC{$x\!:\!(\alpha\to\alpha\to\beta), y\!:\!\alpha \vdash xy : (\alpha\to\beta)$}
\RightLabel{$\to\mathbf{E}$}
\BinaryInfC{$x\!:\!(\alpha\to\alpha\to\beta), y\!:\!\alpha \vdash xyy : \beta$}
\RightLabel{$\to\mathbf{I}$}
\UnaryInfC{$x\!:\!(\alpha\to\alpha\to\beta) \vdash (\lambda y\!:\!\alpha.xyy) : (\alpha\to\beta)$}
\RightLabel{$\to\mathbf{I}$}
\UnaryInfC{$\vdash (\lambda x\!:\!\alpha\to\alpha\to\beta.\lambda y\!:\!\alpha.xyy) : (\alpha\to\alpha\to\beta)\to\alpha\to\beta$}
\end{prooftree}
$$

Finally, we can prove the original formula using natural deduction. We already know the structure of the proof tree, as well as the hypothesis to be used (they are simply the types of x and y in the above term) and where they should be cancelled (this is just where x and y are bounded by λ-abstraction).

Thus, we can build the following proof:

$$
\begin{prooftree}
\AxiomC{α^1}
\AxiomC{α^1}
\AxiomC{$(\alpha\to(\alpha\to\beta))^2$}
\RightLabel{$\to\mathbf{E}$}
\BinaryInfC{$\alpha\to\beta$}
\RightLabel{$\to\mathbf{E}$}
\BinaryInfC{β}
\RightLabel{$\to\mathbf{I}, 1$}
\UnaryInfC{$\alpha\to\beta$}
\RightLabel{$\to\mathbf{I}, 2$}
\UnaryInfC{$(\alpha\to(\alpha\to\beta))\to(\alpha\to\beta)$}
\end{prooftree}
$$

□

We will now introduce negation. In intuitionistic logic, $(\neg\alpha)$ can be defined as abbreviation of $(\alpha\to\bot)$, where $\bot$ is the *falsum*, a formula that is always false. Unfortunately, we do not have type constants, so we can not reproduce this construction in $\lambda\!\to$. However, we can define an analog of it—namely, we can choose a fresh[2] type variable γ and define $(\sim\alpha) = (\alpha\to\gamma)$.

We show two examples of proofs using this new connective (problem 3 of [2]).

Example 5.1.6 $(\alpha\to\beta)\to(\sim\beta\to\sim\alpha)$

When we expand this formula using the definition of $\sim$, we obtain $(\alpha\to\beta)\to((\beta\to\gamma)\to(\alpha\to\gamma))$. As before, it is easy to see that a λ-term of this type must be a function that receives arguments $x\!:\!(\alpha\to\beta)$, $y\!:\!(\beta\to\gamma)$ and

[1] Notice that the construction of this tree is purely mechanical, due to the deterministic nature of the rules involved.

[2] That is, a variable not occurring anywhere in the formula we want to negate.

$z\!:\!\alpha$ and produces an output of type γ. Well, the obvious choice is applying x to z, obtaining $(xz)\!:\!\beta$, and feeding this term as input to y. We obtain the following proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x,y,z\vdash z\!:\!\alpha}\,\mathbf{Ax}\quad \dfrac{}{x,y,z\vdash x\!:\!(\alpha\!\rightarrow\!\beta)}\,\mathbf{Ax}}{x\!:\!(\alpha\!\rightarrow\!\beta),y\!:\!(\beta\!\rightarrow\!\gamma),z\!:\!\alpha\vdash xz\!:\!\beta}\rightarrow\mathbf{E}\quad \dfrac{}{x,y,z\vdash y\!:\!(\beta\!\rightarrow\!\gamma)}\,\mathbf{Ax}}{x\!:\!(\alpha\!\rightarrow\!\beta),y\!:\!(\beta\!\rightarrow\!\gamma),z\!:\!\alpha\vdash y(xz)\,:\,\gamma}\rightarrow\mathbf{E}}{x\!:\!(\alpha\!\rightarrow\!\beta),y\!:\!(\beta\!\rightarrow\!\gamma)\vdash(\lambda z\!:\!\alpha.y(xz))\,:\,(\alpha\!\rightarrow\!\gamma)}\rightarrow\mathbf{I}}{x\!:\!(\alpha\!\rightarrow\!\beta)\vdash(\lambda y\!:\!(\beta\!\rightarrow\!\gamma)\lambda z\!:\!\alpha.y(xz))\,:\,(\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma)}\rightarrow\mathbf{I}}{\vdash(\lambda x\!:\!(\alpha\!\rightarrow\!\beta)\lambda y\!:\!(\beta\!\rightarrow\!\gamma)\lambda z\!:\!\alpha.y(xz))\,:\,(\alpha\!\rightarrow\!\beta)\!\rightarrow\!((\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma))}\rightarrow\mathbf{I}$$

From this term it is easy to generate the proof in natural deduction for the original formula:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha^1\quad(\alpha\!\rightarrow\!\beta)^2}{\beta}\rightarrow\mathbf{E}\quad(\beta\!\rightarrow\!\gamma)^3}{\gamma}\rightarrow\mathbf{E}}{\alpha\!\rightarrow\!\gamma}\rightarrow\mathbf{I},1}{(\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma)}\rightarrow\mathbf{I},3}{(\alpha\!\rightarrow\!\beta)\!\rightarrow\!((\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma))}\rightarrow\mathbf{I},2$$

□

Before we present the second example, we need to make a remark. In intuitionistic logic with falsum, there is an inference rule stating that from $\perp$ anything is derivable. As we do not have that rule in $\lambda\!\rightarrow$, we will have to assume as an extra hypothesis the existence of a term **exfalso** of type $(\gamma\!\rightarrow\!\alpha)$.

Example 5.1.7 $\sim\sim(\sim\sim\alpha\rightarrow\alpha)$

This is a tricky one. We need to find a λ-term of type $(((((\alpha\rightarrow\gamma)\rightarrow\gamma)\rightarrow\alpha)\rightarrow\gamma)\rightarrow\gamma)$; this must be a function that on an input x of type $((((\alpha\!\rightarrow\!\gamma)\!\rightarrow\!\gamma)\!\rightarrow\!\alpha)\!\rightarrow\!\gamma)$ returns an output of type γ.

The only way of producing such an output is by applying x to an object A of type $(((\alpha\!\rightarrow\!\gamma)\!\rightarrow\!\gamma)\!\rightarrow\!\alpha)$. That is, A must be a function that receives as argument any y of type $((\alpha\!\rightarrow\!\gamma)\!\rightarrow\!\gamma)$ and returns a value of type α.

Now using x and y we must build a term B of type α; one possible way is by applying **exfalso** to another term C of type γ. This can be done through finding a term D of type $(\alpha\!\rightarrow\!\gamma)$ and applying y to it.

Such a D must be a function that receives as argument z of type α and returns a term of type γ. Now we can again look at our original x and apply it to a function that receives an argument of type $((\alpha\to\gamma)\to\gamma)$ and returns z.

Carefully following the above line of reasoning we can arrive at the following term:

$$\lambda x:((((\alpha\to\gamma)\to\gamma)\to\alpha)\to\gamma).x(\lambda y:((\alpha\to\gamma)\to\gamma).\mathbf{exfalso}(y(\lambda z:\alpha.x(\lambda w:((\alpha\to\gamma)\to\gamma).z))))$$

The proof that this term is indeed of type $(((((\alpha\to\gamma)\to\gamma)\to\alpha)\to\gamma)\to\gamma)$ is presented in Figure 5.1 (justifications for axioms are omitted) . To make reading easier, we present here the types of the different variables that appear in the proof and omit them in the deduction tree.

Variable	**Type**
x	$((((\alpha\to\gamma)\to\gamma)\to\alpha)\to\gamma)$
y	$((\alpha\to\gamma)\to\gamma)$
z	α
w	$((\alpha\to\gamma)\to\gamma)$

Finally, we can obtain the derivation for the original formula as we show in Figure 5.2.

□

5.2 Propositions as Types

In this section, we will present some more examples of the propositions-as-types interpretation of the systems of the Lambda Cube. We will show how pseudo-terms codify proofs of formulas in the different systems, and compare the proofs in natural deduction with the structure of the pseudo-terms involved. In most cases we will not detail the proof that the term has the required type, giving an overall justification that it is indeed the case.

We start by giving some proofs in second-order propositional logic. By Theorem 4.3.2, this is equivalent to giving typed terms in $\lambda 2$; we will use the simpler notation we introduced for this system instead of the general one when we view it as a Pure Type System, that is, we will write $(\forall\alpha.P)$ instead of $(\Pi\alpha:*.P)$.

Example 5.2.1 Second-order intuitionistic propositional logic comes with a natural candidate for falsum, which is $(\forall\alpha.\alpha)$; in fact, this proposition has the usual properties of the intuitionistic $\bot$, in particular $(\forall\beta.(\forall\alpha.\alpha)\to\beta)$. This is quite obvious in both systems; the relevant term of $\lambda 2$ is $(\lambda\beta:*.\lambda x:(\forall\alpha.\alpha).x\beta)$ (problem 9 of [2]).

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$\mathbf{exfalso}, x \vdash x$}
\AxiomC{}
\UnaryInfC{$\mathbf{exfalso}, x, y \vdash \mathbf{exfalso}$}
\AxiomC{}
\UnaryInfC{$\mathbf{exfalso}, x, y \vdash y$}
\AxiomC{}
\UnaryInfC{$\mathbf{exfalso}, x, y, z \vdash x$}
\AxiomC{}
\UnaryInfC{$\mathbf{exfalso}, x, y, z, w \vdash z$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$\mathbf{exfalso}, x, y, z \vdash (\lambda w.z) : (((\alpha \rightarrow \gamma) \rightarrow \gamma) \rightarrow \alpha)$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$\mathbf{exfalso}, x, y, z \vdash x(\lambda w.z) : \gamma$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$\mathbf{exfalso}, x, y \vdash \lambda z.x(\lambda w.z) : (\alpha \rightarrow \gamma)$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$\mathbf{exfalso}, x, y \vdash y(\lambda z.x(\lambda w.z)) : \gamma$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$\mathbf{exfalso}, x, y \vdash \mathbf{exfalso}(y(\lambda z.x(\lambda w.z))) : \alpha$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$\mathbf{exfalso}, x \vdash (\lambda y.\mathbf{exfalso}(y(\lambda z.x(\lambda w.z)))) : (((\alpha \rightarrow \gamma) \rightarrow \gamma) \rightarrow \alpha)$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$\mathbf{exfalso}, x \vdash x(\lambda y.\mathbf{exfalso}(y(\lambda z.x(\lambda w.z)))) : \gamma$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$\mathbf{exfalso} \vdash \lambda x.x(\lambda y.\mathbf{exfalso}(y(\lambda z.x(\lambda w.z)))) : (((((\alpha \rightarrow \gamma) \rightarrow \gamma) \rightarrow \alpha) \rightarrow \gamma) \rightarrow \gamma)$}
\end{prooftree}
$$

Figure 5.1: Derivation of Type Assignment

$$\dfrac{\alpha^3}{(((\alpha\to\perp)\to\perp)\to\alpha)}\ \to\mathbf{I},4$$

$$\dfrac{(((\alpha\to\perp)\to\perp)\to\alpha) \qquad ((((\alpha\to\perp)\to\perp)\to\alpha)\to\perp)^1}{\dfrac{\perp}{(\alpha\to\perp)}\ \to\mathbf{I},3}\ \to\mathbf{E}$$

$$\dfrac{(\alpha\to\perp) \qquad ((\alpha\to\perp)\to\perp)^2}{\dfrac{\dfrac{\perp}{\alpha}\ \perp}{(((\alpha\to\perp)\to\perp)\to\alpha)}\ \to\mathbf{I},2}\ \to\mathbf{E}$$

$$\dfrac{(((\alpha\to\perp)\to\perp)\to\alpha) \qquad ((((\alpha\to\perp)\to\perp)\to\alpha)\to\perp)^1}{\dfrac{\perp}{(((((\alpha\to\perp)\to\perp)\to\alpha)\to\perp)\to\perp)}\ \to\mathbf{I},1}\ \to\mathbf{E}$$

Figure 5.2: Natural Deduction Derivation

We present the proof of this fact in several steps. We start by showing that $(\forall\beta.(\forall\alpha.\alpha)\to\beta)$ is a valid type:

$$\dfrac{\dfrac{}{\vdash *:\Box}\,\text{Ax} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash *:\Box}\,\text{Ax} \quad \dfrac{}{\vdash *:\Box}\,\text{Ax}}{\beta:*\vdash *:\Box}\,\text{W} \quad \dfrac{\dfrac{\dfrac{}{\vdash *:\Box}\,\text{Ax} \quad \dfrac{}{\vdash *:\Box}\,\text{Ax}}{\beta:*\vdash *:\Box}\,\text{W}}{\beta:*,\alpha:*\vdash\alpha:*}\,\text{St}}{\beta:*\vdash(\forall\alpha.\alpha):*}\,\text{Prod} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash *:\Box}\,\text{Ax} \quad \dfrac{\dfrac{}{\vdash *:\Box}\,\text{Ax}}{\alpha:*\vdash\alpha:*}\,\text{St}}{\vdash(\forall\alpha.\alpha):*}\,\text{Prod} \quad \dfrac{}{\vdash *:\Box}\,\text{Ax}}{x:(\forall\alpha.\alpha)\vdash *:\Box}\,\text{W}}{\beta:*,x:(\forall\alpha.\alpha)\vdash\beta:*}\,\text{St}}{\beta:*\vdash(\forall\alpha.\alpha)\to\beta:*}\,\text{Prod}}{\vdash(\forall\beta.(\forall\alpha.\alpha)\to\beta):*}\,\text{Prod}$$

Using as lemmas the various formulas that occur in the previous tree, we can now prove that our term has the intended type:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\beta:*\vdash(\forall\alpha.\alpha):*}\,\text{L}}{\beta:*,x:(\forall\alpha.\alpha)\vdash x:(\forall\alpha.\alpha)}\,\text{St} \quad \dfrac{}{\beta:*,x:(\forall\alpha.\alpha)\vdash\beta:*}\,\text{L}}{\beta:*,x:(\forall\alpha.\alpha)\vdash x\beta:\beta}\,\text{App} \quad \dfrac{}{\beta:*\vdash(\forall\alpha.\alpha)\to\beta:*}\,\text{L}}{\beta:*\vdash\lambda x:(\forall\alpha.\alpha).x\beta:(\forall\alpha.\alpha)\to\beta}\,\text{Abs} \quad \dfrac{}{\vdash(\forall\beta.(\forall\alpha.\alpha)\to\beta):*}\,\text{L}}{\vdash(\lambda\beta:*.\lambda x:(\forall\alpha.\alpha).x\beta):(\forall\beta.(\forall\alpha.\alpha)\to\beta)}\,\text{Abs}$$

From this proof it is simple to generate the corresponding proof in natural deduction. Notice that, as we are treating $\lambda 2$ as a Pure Type System, there is a part of the derivation which is ignored—namely, the part corresponding to the proof that the required type is indeed a type. This corresponds to checking that the corresponding formula is a well formed formula, which is always guaranteed in proof trees for natural deduction systems. Therefore, our proof is:

$$\dfrac{\dfrac{\dfrac{\forall\alpha.\alpha}{\beta}\,\forall\mathbf{E}}{(\forall\alpha.\alpha)\to\beta}\to\mathbf{I},1}{\forall\beta.((\forall\alpha.\alpha)\to\beta)}\,\forall\mathbf{I}$$

$\Box$

From now on, we will write $\bot^2$ for the type $(\forall\alpha.\alpha)$, and $\neg\alpha$ for $\bot^2\to\alpha$; **exfalso** will denote the (canonical) inhabitant of type $(\forall\beta.\bot^2\to\beta)$.

At this stage, the process of proving that a given term has a certain type should be clear; in fact, this problem is decidable[3] and easily implemented in a computer algorithm. Therefore, as things begin to get more complicated, we will hereafter present terms with their type without showing the corresponding proof tree.

[3] This is quite easy to prove, by structural induction.

In second-order proposition logic we can generalize Examples 5.1.2 and the like by quantifying over α, β and other intervening type variables; this corresponds, as we showed before, to generating a new term in $\lambda 2$ that receives those types as arguments. When we get to negation, however, things become more interesting: we no longer need **exfalso** as a hypothesis, as in the previous example we have shown that there is an inhabitant of type $(\forall\beta.(\perp^2 \to \beta))$. For curiosity's sake we present a term of type $\forall\alpha.\neg\neg(\neg\neg\alpha\to\alpha)$ (problem 10 of [2]). Notice that this term is easy to obtain from the term in Example 5.1.7.

Example 5.2.2 The term

$$\lambda\alpha{:}*.\lambda x{:}(\neg(\neg\neg\alpha\to\alpha)).x(\lambda y{:}\neg\neg\alpha.\mathbf{exfalso}\alpha(y(\lambda z{:}\alpha.x(\lambda w{:}\neg\neg\alpha.z))))$$

has type $\forall\alpha.\neg\neg(\neg\neg\alpha\to\alpha)$. $\square$

We will now move over to first-order predicate logic. According to Theorem 4.3.2, we must now change our frame of reference to the system λP.

The main difference between this system and $\lambda\to$ is the introduction of the rule $(*, \square, \square)$. This rule allows us to define functions that produce output on *types*, which we could not do in $\lambda\to$; notice also that this system generalizes $\lambda\to$ in quite a different way than $\lambda 2$ did: in $\lambda 2$, we can define functions that act on *types* and return *terms*; in λP we have functions acting on *terms* and returning *types*.

The motivation behind this generalization is quite easy to explain. We want to have *parameterized types*, that is, types depending on terms. Thus, we might have a collection A of terms, to each of which we can assign a type. This can be represented by a function f from A into $*$, which will have as type $A\to *$.

Example 5.2.3 In λP we can prove that every antisymmetric relation is irreflexive (problem 18 of [2]).

We will start by representing this proposition in first-order predicate logic. We need a set A and a relation $R \subseteq A \times A$; also we will suppose that R is antisymmetric, that is, $\forall_{x,y\in A} R(x,y) \to \neg R(y,x)$. Then we want to prove that R is irreflexive, that is, that $\forall_{x\in A}\neg R(x,x)$.

This translates quite immediately into λP in the following way: just read $\forall_{x\in A}$ as $\Pi x{:}A$; negation is defined as for $\lambda\to$.[4]

As for A, it is just a type; how do we represent R? Well, it is just a function that assigns to every pair of elements of A a type—that is, $R{:}A\to A\to *$. Intuitively, for $x,y{:}A$, the type Rxy will be inhabited iff $R(x,y)$.

[4]Notice that $\perp^2$ is not a valid type in λP.

Define $\Gamma = A{:}*, R{:}A \to A \to *, \mathbf{antisym}{:}\Pi x, y{:}A.(Rxy \to \sim Ryx)$.

Given these representations, what we want to prove is that there is a term M such that

$$\Gamma \vdash M : (\Pi x : A.(\sim Rxx)).$$

We start by noticing that, by use of the Application Rule,

$$\Gamma, x : A \vdash \mathbf{antisym}xx : (Rxx \to \sim Rxx) \tag{5.1}$$

Remember that $\sim Rxx$ is by definition $(Rxx \to \gamma)$, for some fresh γ. We are trying to find an inhabitant of this type, given x; this corresponds to defining a function that receives an input of type Rxx and produces an output of type γ. This suggests that we add an extra hypothesis $y{:}Rxx$. It is now easy to see that from Equation 5.1 we can conclude, using Weakening and Application, that

$$\Gamma, x : A, y : Rxx \vdash \mathbf{antisym}xxyy : \gamma \tag{5.2}$$

Now, using Abstraction twice, we conclude that

$$\Gamma \vdash (\lambda x : A \lambda y : Rxx.\mathbf{antisym}xxyy) : (\Pi x : A.(\sim Rxx)) \tag{5.3}$$

From the derivation of type assignment for this term it is easy to produce the corresponding derivation in natural deduction for the corresponding first-order formula. □

To conclude, we will present a lengthy example showing how algebraic reasoning can be incorporated in λP and $\lambda P2$. This example corresponds to problems 19 and 26 of [2].

We start with an algebraic concept.

Definition 5.2.4 An *abelian group* is a structure $\langle G, + \rangle$ where $+ : G \to G \to G$ is a commutative and associative operation with 0 element such that every element of G has an inverse. □

As is usual, we denote the inverse of x by $(-x)$.

Proposition 5.2.5 For every $x \in G$, $(-(-x)) = x$.

Proof. By definition, the inverse of x satisfies

$$x + (-x) = 0 \tag{5.4}$$

Adding $(-(-x))$ to both sides of this equation, we obtain

$$(x + (-x)) + (-(-x)) = (-(-x)) \tag{5.5}$$

By associativity, we conclude that

$$x + ((-x) + (-(-x))) = (-(-x)) \tag{5.6}$$

But $(-(-x))$ is the inverse of $(-x)$, so $(-x) + (-(-x)) = 0$. Substituting this in the last equation, we get

$$x + 0 = (-(-x)) \tag{5.7}$$

and, by definition of zero,

$$x = (-(-x)) \tag{5.8}$$

which by symmetry is precisely

$$(-(-x)) = x \tag{5.9}$$

as we wanted to prove. □

We will now formalize this proof using λP. We start with a type $G : *$, that will represent our group.

Equality is a binary relation in G that is reflexive, symmetric and transitive. This can be represented by the following context Γ_1:

$$\begin{array}{rcl} \Gamma_1 & = & \mathbf{eq} : G \to G \to *, \\ & & \mathbf{refl} : \Pi x : G.\mathbf{eq}xx, \\ & & \mathbf{sym} : \Pi x, y : G.\mathbf{eq}xy \to \mathbf{eq}yx, \\ & & \mathbf{trans} : \Pi x, y, z : G.\mathbf{eq}xy \to \mathbf{eq}yz \to \mathbf{eq}xz. \end{array}$$

For example, if $\mathbf{eq}xy$ is inhabited by A, then $\mathbf{sym}xyA$ is an inhabitant of $\mathbf{eq}yx$.

Next, we need to define the operations in G. We will represent $+$ by a term $+ : G \to G \to G$—that is, if x and y are of type G, then so is $(+xy)$.

The properties of this operation are presented in context Γ_2:

$$\begin{array}{rcl} \Gamma_2 & = & + : G \to G \to G, \\ & & 0 : G, \\ & & - : G \to G, \\ & & \mathbf{comm} : \Pi x, y : G.\mathbf{eq}(+xy)(+yx), \\ & & \mathbf{assoc} : \Pi x, y, z : G.\mathbf{eq}(+(+xy)z)(+x(+yz)), \\ & & \mathbf{inv} : \Pi x : G.\mathbf{eq}(+x(-x))0, \\ & & \mathbf{neutral} : \Pi x : G.\mathbf{eq}(+x0)x, \end{array}$$

We will write $\overline{x}$ for $(-x)$ and $\overline{\overline{x}}$ for $(-(-x))$, so as to make the notation lighter.

Finally, addition is a congruence with respect to equality, that is, if two terms are equal then adding them to the same element of G produces two equal terms. We specify this in another context:

$$\Gamma_3 = \mathbf{congr}\!:\!\Pi x, x'\!:\!G.\mathbf{eq}xx' \to \Pi y\!:\!G.\mathbf{eq}(+xy)(+x'y).$$

Defining $\Gamma = G\!:\!*, \Gamma_1, \Gamma_2, \Gamma_3$, we will now find an M such that $\Gamma, x\!:\!G \vdash M\!:\!\mathbf{eq}\overline{\overline{x}}x$.

Proposition 5.2.6 There is a term M such that $\Gamma, x\!:\!G \vdash M\!:\!\mathbf{eq}\overline{\overline{x}}x$.

Proof. We will just repeat the above algebraic proof, presenting terms that will have each of the preceding equations as types. All statements are understood to be derivable in context $\Gamma, x\!:\!G$.

Equation 5.4 is easy to produce; we just take **inv** and apply it to x.

$$\mathbf{inv}x\!:\!\mathbf{eq}(+x\overline{x})0$$

We now want to add $\overline{\overline{x}}$ to both sides of equation 5.4. This can be done in two steps. First we apply **congr** to $+x\overline{x}$ and 0, obtaining a term

$$\mathbf{congr}(+x\overline{x})0 : \mathbf{eq}(+x\overline{x})0 \to \Pi y\!:\!G.\mathbf{eq}(+(+x\overline{x})y)(+0y).$$

We apply this to $\mathbf{inv}x$ and $\overline{\overline{x}}$, obtaining

$$\mathbf{congr}(+x\overline{x})0(\mathbf{inv}x)\overline{\overline{x}} : \mathbf{eq}(+(+x\overline{x})\overline{\overline{x}})(+0\overline{\overline{x}}).$$

We will denote this term symply by A.

In order to have exactly equation 5.5 we want to have symply $\overline{\overline{x}}$ as the last subterm of the last expression. This is not hard to do; we start by applying **neutral** to $\overline{\overline{x}}$, obtaining

$$\mathbf{neutral}\overline{\overline{x}} : \mathbf{eq}(+\overline{\overline{x}}0)\overline{\overline{x}};$$

comm applied to 0 and $\overline{\overline{x}}$ yields a term

$$\mathbf{comm}0\overline{\overline{x}} : \mathbf{eq}(+0\overline{\overline{x}})(+\overline{\overline{x}}0).$$

Now we just have to combine these three terms using **trans**. First we get

$$\mathbf{trans}(+(+x\overline{x})\overline{\overline{x}})(+0\overline{\overline{x}})(+\overline{\overline{x}}0)A(\mathbf{comm}0\overline{\overline{x}}) : \mathbf{eq}(+(+x\overline{x})\overline{\overline{x}})(+\overline{\overline{x}}0),$$

which we will denote by B, and finally

$$\mathbf{trans}(+(+x\overline{x})\overline{\overline{x}})(+\overline{\overline{x}}0)\overline{\overline{x}}B(\mathbf{neutral}\overline{\overline{x}}) : \mathbf{eq}(+(+x\overline{x})\overline{\overline{x}})\overline{\overline{x}}.$$

In the sequence, we will denote this term by C. Notice that C proves equation 5.5.

We move on to equation 5.6. This is just associativity and transitivity applied to the previous equation. First, we apply **assoc** to x, $\overline{x}$ and $\overline{\overline{x}}$, obtaining

$$\mathbf{assoc}x\overline{x}\overline{\overline{x}} : \mathbf{eq}(+(+x\overline{x})\overline{\overline{x}})(+x(+\overline{x}\overline{\overline{x}})).$$

Now we need to apply symmetry to the last expression and, by transitivity, we obtain the desired result. Define D to be the term

$$\mathbf{sym}(+(+x\overline{x})\overline{\overline{x}})(+x(+\overline{x}\overline{\overline{x}}))(\mathbf{assoc}x\overline{x}\overline{\overline{x}}) : \mathbf{eq}(+(+x\overline{x})\overline{\overline{x}})(+x(+\overline{x}\overline{\overline{x}})).$$

We now apply **trans** to the arguments $(+(+x\overline{x})\overline{\overline{x}})$, $(+x(+\overline{x}\overline{\overline{x}}))$ and $\overline{\overline{x}}$ and then to C and D; the term we get is

$$\mathbf{trans}(+(+x\overline{x})\overline{\overline{x}})(+x(+\overline{x}\overline{\overline{x}}))\overline{\overline{x}}CD : \mathbf{eq}(+x(+\overline{x}\overline{\overline{x}}))\overline{\overline{x}},$$

therefore proving equation 5.6. We will refer to this term as E.

We now want to show that the lefthandside of 5.6 is equal to $x+0$. This is done using the definition of inverse. We start with

$$\mathbf{inv}\overline{x} : \mathbf{eq}(+\overline{x}\overline{\overline{x}})0.$$

We apply congruence, obtaining

$$\mathbf{congr}(+\overline{x}\overline{\overline{x}})0(\mathbf{inv}x)x : \mathbf{eq}(+(+\overline{x}\overline{\overline{x}})x)(+0x),$$

which we will denote by F.

Using transitivity with F and

$$\mathbf{comm}0x : \mathbf{eq}(+0x)(+x0)$$

allows us to obtain

$$\mathbf{trans}(+(+\overline{x}\overline{\overline{x}})x)(+0x)(+x0)F(\mathbf{comm}0x) : \mathbf{eq}(+(+\overline{x}\overline{\overline{x}})x)(+x0),$$

which we will refer to as G.

Applying symmetry we obtain

$$\mathbf{sym}(+(+\overline{x}\overline{\overline{x}})x)(+x0)G : \mathbf{eq}(+x0)(+(+\overline{x}\overline{\overline{x}})x);$$

naming this term H, commutativity and transitivity allow us to find

$$\mathbf{trans}(+x0)(+(+\overline{x}\overline{\overline{x}})x)(+x(+\overline{x}\overline{\overline{x}}))H(\mathbf{comm}(+\overline{x}\overline{\overline{x}})x) : \mathbf{eq}(+x0)(+x(+\overline{x}\overline{\overline{x}})).$$

This auxiliary term, which we will call J, can be combined with E using transitivity, producing

$$\mathbf{trans}(+x0)(+x(+\overline{x}\overline{\overline{x}}))\overline{\overline{x}}JE : \mathbf{eq}(+x0)\overline{\overline{x}}.$$

This term, which we will denote as K, proves equation 5.7.

Now it is easy. We have

$$\mathbf{neutral}x : \mathbf{eq}(+x0)x;$$

applying symmetry we get

$$\mathbf{sym}(+x0)x(\mathbf{neutral}x) : \mathbf{eq}x(+x0).$$

By transitivity with K we get

$$\mathbf{trans}x(+x0)\overline{\overline{x}}K(\mathbf{sym}(+x0)x(\mathbf{neutral}x)) : \mathbf{eq}x\overline{\overline{x}},$$

which we will name L.

This is equation 5.8. Applying symmetry yields our final result:

$$\mathbf{sym}x\overline{\overline{x}}L : \mathbf{eq}\overline{\overline{x}}x.$$

M is just the term $\lambda x\!:\!G.(\mathbf{sym}x\overline{\overline{x}}L)$.

Type-checking M is far from being trivial, but it is a purely mechanical task. Therefore, we will only present the full expression for M as a curiosity in Figure 5.2.

$\square$

For the next property we need to make another definition. We use the following abbreviation, where $n \in \mathbb{N}$:

$$nx \overset{\text{def}}{=} \underbrace{x + \ldots + x}_{n \text{ times}}$$

DEFINITION 5.2.7 An element x of an abelian group is said to *have torsion* iff there is a natural number n such that $nx = 0$. $\square$

It is obvious that, in particular, if $2x = 0$, then x has torsion. We will show this using the context Γ above defined; however, we will use a stronger equality than **eq**.

$$
\begin{array}{l}
\lambda x\!:\!G. \\
\quad (\mathbf{sym} x \overline{\overline{x}} \\
\quad\quad (\mathbf{trans} x (+x0) \overline{\overline{x}} \\
\quad\quad\quad (\mathbf{trans} (+x0)(+x(+\overline{x}\overline{\overline{x}})) \\
\quad\quad\quad\quad (\mathbf{trans} (+x0)(+(+\overline{x}\overline{\overline{x}})x)(+x(+\overline{x}\overline{\overline{x}})) \\
\quad\quad\quad\quad\quad (\mathbf{sym} (+(+\overline{x}\overline{\overline{x}})x)(+x0) \\
\quad\quad\quad\quad\quad\quad (\mathbf{trans} (+(+\overline{x}\overline{\overline{x}})x)(+0x)(+x0) \\
\quad\quad\quad\quad\quad\quad\quad (\mathbf{congr} (+\overline{x}\overline{\overline{x}})0(\mathbf{inv} x)x) \\
\quad\quad\quad\quad\quad\quad\quad (\mathbf{comm} 0x))) \\
\quad\quad\quad\quad\quad (\mathbf{comm} (+\overline{x}\overline{\overline{x}})x)) \\
\quad\quad\quad\quad (\mathbf{trans} (+(+x\overline{x})\overline{\overline{x}})(+x(+\overline{x}\overline{\overline{x}}))\overline{\overline{x}} \\
\quad\quad\quad\quad\quad (\mathbf{trans} (+(+x\overline{x})\overline{\overline{x}})(+\overline{\overline{x}}0)\overline{\overline{x}} \\
\quad\quad\quad\quad\quad\quad (\mathbf{trans} (+(+x\overline{x})\overline{\overline{x}})(+0\overline{\overline{x}})(+\overline{\overline{x}}0) \\
\quad\quad\quad\quad\quad\quad\quad (\mathbf{congr} (+x\overline{x})0(\mathbf{inv} x)\overline{\overline{x}}) \\
\quad\quad\quad\quad\quad\quad\quad (\mathbf{comm} 0\overline{\overline{x}})) \\
\quad\quad\quad\quad\quad\quad (\mathbf{neutral} \overline{\overline{x}})) \\
\quad\quad\quad\quad\quad (\mathbf{sym} (+(+x\overline{x})\overline{\overline{x}})(+x(+\overline{x}\overline{\overline{x}})) \\
\quad\quad\quad\quad\quad\quad (\mathbf{assoc} x\overline{x}\overline{\overline{x}})))) \\
\quad\quad\quad (\mathbf{sym} (+x0)x \\
\quad\quad\quad\quad (\mathbf{neutral} x))))
\end{array}
$$

Figure 5.3: Inhabitant of $\Pi x\!:\!G.(\mathbf{eq}\overline{\overline{x}}x)$

DEFINITION 5.2.8 The *Leibniz equality* on type $A{:}*$ is defined by

$$\mathbf{eq}_L xy \stackrel{\text{def}}{=} \Pi P{:}A\to *.Px \to Py,$$

in context $x, y{:}A$. □

We will not prove, but $\mathbf{eq}_L$ is an equivalence relation (problem 25 of [2]), therefore has all the usual properties of equality. In fact, $\mathbf{eq}_L$ is an *extensional equality*: two terms x and y of type A are identified by $\mathbf{eq}_L$ iff there is no type-assignment function from A that distinguishes them.

PROPOSITION 5.2.9 There is a term T such that:

i. $\Gamma \vdash T{:}G\to *$;

ii. for $x{:}G$, Tx is inhabited iff x has torsion.

PROOF. Following the suggestion in [2], we start by defining a term $Q{:}G\to G\to *$ such that Qxy is inhabited iff y is in the smallest subset of G containing x and closed under addition. As before, all statements are understood to be derivable in context $\Gamma, x{:}G$ or $\Gamma, y{:}G$.

We can represent the above sentence in second-order predicate logic by the following sentence:

$$\forall P.[(P(x) \wedge \forall_{z\in G}[P(z)\to P(x+z)])\to P(y)],$$

that is, y is in all sets[5] containing x and closed under addition of x. This is precisely the set we wanted to characterize, so we just have to translate this proposition as a type.

As we are working in second order, we can define conjunction by abbreviation as

$$\alpha \wedge \beta \stackrel{\text{def}}{=} (\forall\gamma(\alpha\to\beta\to\gamma)\to\gamma).$$

P becomes a term of type $G\to *$, and we get

$$Qxy = \Pi P{:}(G\to *).[\Pi\gamma{:}*.(Px\to \Pi z{:}G.(Pz\to P(+xz))\to\gamma)\to\gamma]\to Py.$$

Observe that $Qxy : *$ is inhabited iff we can, given $P{:}G\to *$, an inhabitant of Px and a function from Pz into $P(+xz)$, find an inhabitant of Py.

Define now $T = \lambda x{:}G.Qx0$. Then $T{:}G\to *$ and Tx is inhabited iff $Qx0$ also is, which happens iff 0 is in all the sets containing x and closed under addition. □

[5]Remember that subsets of G are equivalent to predicates with domain G.

PROPOSITION 5.2.10 There is a term X such that

$$\Gamma \vdash X : (\Pi x{:}G.\mathbf{eq}_L(+xx)0 \to Tx).$$

PROOF. We will start by assuming

$$x : G$$

and

$$\omega : \mathbf{eq}_L(+xx)0.$$

Using x and ω we want to find a term of type Tx.

Developing $Tx = Qx0$, we get

$$\Pi P{:}(G \to *).[\Pi\gamma{:}*.(Px \to \Pi z{:}G.(Pz \to P(+xz)) \to \gamma) \to \gamma] \to P0.$$

An inhabitant of Tx will then have to receive arguments $P : (G \to *)$ and $\alpha : (\Pi\gamma : *.(Px \to \Pi z : G.(Pz \to P(+xz)) \to \gamma) \to \gamma)$ and produce output of type $P0$.

Suppose then that

$$P : (G \to *)$$

and

$$\alpha : (\Pi\gamma{:}*.(Px \to \Pi z{:}G.(Pz \to P(+xz)) \to \gamma) \to \gamma).$$

In this context, we can show that $P0{:}*$, hence

$$\alpha(P0) : (Px \to \Pi z{:}G.(Pz \to P(+xz)) \to P0) \to P0.$$

In order to find an inhabitant of $P0$, we can try to find a term of type $(Px \to \Pi z : G.(Pz \to P(+xz)) \to P0)$ and apply $\alpha(P0)$ to it. This term will be of the form

$$\lambda A{:}Px\lambda B{:}(\Pi z{:}G.(Pz \to P(+xz))).C$$

with $C{:}P0$. So let us now add to the context

$$A : Px$$

and

$$B : (\Pi z{:}G.(Pz \to P(+xz))).$$

As we have by hypothesis $x{:}G$, we get

$$Bx : Px \to P(+xx).$$

We just supposed that $A{:}Px$, so we can apply Bx to A obtaining

$$BxA : P(+xx).$$

By hypothesis $\omega : \mathbf{eq}_L(+xx)0$, that is,

$$\omega : \Pi P{:}(G \to *).(P(+xx) \to P0).$$

In particular,

$$\omega P : (P(+xx) \to P0)$$

and

$$\omega P(BxA) : P0.$$

Then we can define D as $(\lambda A{:}Px\lambda B{:}(\Pi z{:}G.(Pz \to P(+xz))).\omega P(BxA))$ and we have

$$D : (Px \to \Pi z{:}G.(Pz \to P(+xz)) \to P0);$$

therefore,

$$\alpha(P0)D : P0.$$

Applying abstraction, we can conclude that X is the term (types are omited)

$$\lambda x.\lambda\omega.\lambda P.\lambda\alpha.[\alpha(P0)(\lambda A.\lambda B.(\omega P(BxA)))]$$

and that

$$\Gamma \vdash X : (\Pi x{:}G.\mathbf{eq}_L(+xx)0 \to Tx).$$

□

5.3 Computability in Pure Type Systems

In this last section, we will show how Pure Type Systems also work as a model of computation. These examples show another aspect of these systems in which we have not focused much until now.

We will start by introducing a numeral system for untyped λ-calculus; then, we will proceed to generalize it to typed λ-calculus and progressively to other type systems. Also we will show how to define functions on these numerals; computational completeness of λ-calculus is proved in Chapter 11 of [1][6], so we will not dwell on it, preferring to give some specific examples.

We start with some notation.

DEFINITION 5.3.1 Let F, X be λ-terms and $n \in \mathbb{N}$. The term $F^n X$ is defined inductively as follows:

[6] Although the proof given there is not for the numeral system we shall use, but rather for the one introduced in Chapter 3, in Chapter 6 the author explains why the proof he gives simultaneously covers both systems.

i. F^0X is X;

ii. $F^{n+1}X$ is $F(F^nX)$. □

Definition 5.3.2 The *Church numerals* in Λ are defined as follows: if $n \in \mathbb{N}$ then $c_n = \lambda f \lambda x. f^n x$. □

We will not prove it, but these numerals share the properties that were earlier discussed to be important; namely, $c_n = c_m$ iff $n = m$.

We could define some terms in Λ representing diverse functions acting in the natural numbers; however, as we can always get terms in Λ from typed λ-terms, we will immediately define analogues to Church numerals in $\lambda\!\rightarrow$ and work with them. The examples we will give in this system are very easy to translate into Λ—just drop the types in the corresponding terms.

Definition 5.3.3 Let $\alpha \in \mathbb{T}$ be a type variable in $\lambda\!\rightarrow$. The *type for Church numerals over* α in $\lambda\!\rightarrow$ is $\mathrm{Nat}^\alpha \stackrel{\text{def}}{=} (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. □

It is easy to see why Nat^α is defined in this way: the numeral c_n represents the operator that, given a function f and an argument x, iterates f n times over x. Supposing x to be of type α, then f must be of type $(\alpha \rightarrow \alpha)$ and we will obtain a term of type $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$.

Definition 5.3.4 Let $\alpha \in \mathbb{T}$ be a type variable in $\lambda\!\rightarrow$. The *Church numerals* in $\lambda\!\rightarrow$ are defined as follows: if $n \in \mathbb{N}$, then $c_n^\alpha = \lambda f : (\alpha \rightarrow \alpha) \lambda x : \alpha . f^n x$. □

This numeral system shares all the good properties of the numeral system for combinatory algebras introduced in Chapter 3; namely, $c_m^\alpha = c_n^\alpha$ iff $m = n$, and all numerals are solvable by $(\lambda x : \alpha . x)$. Furthermore, all numerals are in β-normal form.

Parameterized addition and multiplication (on α) are easy to define over this number system.

Proposition 5.3.5 For every type variable α, there are terms $\oplus^\alpha$ and $\otimes^\alpha$ such that:

i. $\vdash \oplus^\alpha, \otimes^\alpha : \mathrm{Nat}^\alpha \rightarrow \mathrm{Nat}^\alpha$;

ii. for all $n, m \in \mathbb{N}$, $\oplus^\alpha c_n^\alpha c_m^\alpha \rightarrow_\beta^* c_{n+m}^\alpha$;

iii. for all $n, m \in \mathbb{N}$, $\otimes^\alpha c_n^\alpha c_m^\alpha \rightarrow_\beta^* c_{n\times m}^\alpha$;

Proof. Define

$$\oplus^\alpha \stackrel{\text{def}}{=} \lambda n, m\!:\!\mathrm{Nat}^\alpha \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda x\!:\!\alpha.(nf(mfx))$$

and

$$\otimes^\alpha \stackrel{\text{def}}{=} \lambda n, m\!:\!\mathrm{Nat}^\alpha \lambda f\!:\!(\alpha\!\to\!\alpha).n(mf),$$

where $\lambda n, m\!:\!\mathrm{Nat}^\alpha.P$ stands for $\lambda n\!:\!\mathrm{Nat}^\alpha.\lambda m\!:\!\mathrm{Nat}^\alpha.P$.

We begin by proving that $\oplus^\alpha$ has the required type. The proof is shown in Figure 5.4.

Proving that $\oplus^\alpha \mathrm{c}_n^\alpha \mathrm{c}_m^\alpha \to_\beta^* \mathrm{c}_{n+m}^\alpha$ is quite direct:

$$\begin{array}{ll}
\multicolumn{2}{l}{\oplus^\alpha \mathrm{c}_n^\alpha c_m^\alpha \to_\beta^*} \\
\to_\beta^* & \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda x\!:\!\alpha.\mathrm{c}_n^\alpha f(\mathrm{c}_m^\alpha fx) \\
\stackrel{\text{def}}{=} & \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda x\!:\!\alpha.(\lambda g\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.g^n y)f((\lambda h\!:\!(\alpha\!\to\!\alpha)\lambda z\!:\!\alpha.h^m z)fx) \\
\to_\beta^* & \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda x\!:\!\alpha.((\lambda g\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.g^n y)f(f^m x) \\
\to_\beta^* & \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda x\!:\!\alpha.f^n(f^m x) \\
\stackrel{\text{def}}{=} & \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda x\!:\!\alpha.f^{n+m}x \\
\stackrel{\text{def}}{=} & c_{n+m}^\alpha
\end{array}$$

As for $\otimes^\alpha$, proving that it too has type $\mathrm{Nat}^\alpha \to \mathrm{Nat}^\alpha \to \mathrm{Nat}^\alpha$ is also quite simple:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n,m,f \vdash n : \mathrm{Nat}^\alpha}\,\mathbf{Ax} \quad \dfrac{\dfrac{}{n,m,f \vdash m : \mathrm{Nat}^\alpha}\,\mathbf{Ax} \quad \dfrac{}{n,m,f \vdash f : (\alpha\!\to\!\alpha)}\,\mathbf{Ax}}{n,m,f \vdash mf : (\alpha\!\to\!\alpha)}\,\to\mathbf{E}}{n,m\!:\!\mathrm{Nat}^\alpha, f\!:\!(\alpha\!\to\!\alpha) \vdash n(mf) : \alpha\!\to\!\alpha}\,\to\mathbf{E}}{n,m\!:\!\mathrm{Nat}^\alpha \vdash \lambda f\!:\!(\alpha\!\to\!\alpha).n(mf) : \mathrm{Nat}^\alpha}\,\to\mathbf{I}}{n\!:\!\mathrm{Nat}^\alpha \vdash \lambda m\!:\!\mathrm{Nat}^\alpha\lambda f\!:\!(\alpha\!\to\!\alpha).n(mf) : \mathrm{Nat}^\alpha\!\to\!\mathrm{Nat}^\alpha}\,\to\mathbf{I}}{\vdash \lambda n,m\!:\!\mathrm{Nat}^\alpha\lambda f\!:\!(\alpha\!\to\!\alpha).n(mf) : \mathrm{Nat}^\alpha\!\to\!\mathrm{Nat}^\alpha\!\to\!\mathrm{Nat}^\alpha}\,\to\mathbf{I}$$

The proof that $\otimes^\alpha \mathrm{c}_n^\alpha \mathrm{c}_m^\alpha \to_\beta^* \mathrm{c}_{n\times m}^\alpha$ is by induction on n. First we simplify $\otimes^\alpha \mathrm{c}_n^\alpha \mathrm{c}_m^\alpha$:

$$\begin{array}{ll}
\multicolumn{2}{l}{\otimes^\alpha \mathrm{c}_n^\alpha c_m^\alpha \to_\beta^*} \\
\to_\beta^* & \lambda f\!:\!(\alpha\!\to\!\alpha).\mathrm{c}_n^\alpha(\mathrm{c}_m^\alpha f) \\
\stackrel{\text{def}}{=} & \lambda f\!:\!(\alpha\!\to\!\alpha).(\lambda g\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.g^n y)((\lambda h\!:\!(\alpha\!\to\!\alpha)\lambda z\!:\!\alpha.h^m z)f) \\
\to_\beta & \lambda f\!:\!(\alpha\!\to\!\alpha).(\lambda g\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.g^n y)(\lambda z\!:\!\alpha.f^m z) \\
\to_\beta & \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.(\lambda z\!:\!\alpha.f^m z)^n y
\end{array}$$

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$n, m, f, x \vdash n$}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$n, m, f, x \vdash f$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$n, m : \mathrm{Nat}^\alpha, f : (\alpha \rightarrow \alpha), x : \alpha \vdash nf : (\alpha \rightarrow \alpha)$}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$n, m, f, x \vdash m$}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$n, m, f, x \vdash f$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$n, m, f, x \vdash mf : (\alpha \rightarrow \alpha)$}
\AxiomC{}
\RightLabel{$\mathbf{Ax}$}
\UnaryInfC{$n, m, f, x \vdash x$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$n, m : \mathrm{Nat}^\alpha, f : (\alpha \rightarrow \alpha), x : \alpha \vdash mfx : \alpha$}
\RightLabel{$\rightarrow \mathbf{E}$}
\BinaryInfC{$n, m : \mathrm{Nat}^\alpha, f : (\alpha \rightarrow \alpha), x : \alpha \vdash (nf(mfx)) : \alpha$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$n, m : \mathrm{Nat}^\alpha, f : (\alpha \rightarrow \alpha) \vdash \lambda x : \alpha.(nf(mfx)) : \alpha \rightarrow \alpha$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$n, m : \mathrm{Nat}^\alpha \vdash \lambda f : (\alpha \rightarrow \alpha) \lambda x : \alpha.(nf(mfx)) : \mathrm{Nat}^\alpha$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$n : \mathrm{Nat}^\alpha \vdash \lambda m : \mathrm{Nat}^\alpha \lambda f : (\alpha \rightarrow \alpha) \lambda x : \alpha.(nf(mfx)) : \mathrm{Nat}^\alpha \rightarrow \mathrm{Nat}^\alpha$}
\RightLabel{$\rightarrow \mathbf{I}$}
\UnaryInfC{$\vdash \lambda n, m : \mathrm{Nat}^\alpha \lambda f : (\alpha \rightarrow \alpha) \lambda x : \alpha.(nf(mfx)) : \mathrm{Nat}^\alpha \rightarrow \mathrm{Nat}^\alpha \rightarrow \mathrm{Nat}^\alpha$}
\end{prooftree}
$$

Figure 5.4: Type checking for $\oplus^\alpha$

i. if $n = 0$, then the last term is simply $\lambda f : (\alpha \to \alpha)\lambda y : \alpha.y$, which by definition is c_0^α;

ii. for the induction step, we have that

$$\begin{aligned}
&\lambda f\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.(\lambda z\!:\!\alpha.f^m z)^{n+1} y \\
&\stackrel{\text{def}}{=} \quad \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.(\lambda z\!:\!\alpha.f^m z)((\lambda z\!:\!\alpha.f^m z)^n y) \\
&\to_\beta^* \quad \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.(\lambda z\!:\!\alpha.f^m z)((\lambda z\!:\!\alpha.f^{nm} z) y) \\
&\to_\beta \quad \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.(\lambda z\!:\!\alpha.f^m z)(f^{nm} y) \\
&\to_\beta \quad \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.f^m (f^{nm} y) \\
&\stackrel{\text{def}}{=} \quad \lambda f\!:\!(\alpha\!\to\!\alpha)\lambda y\!:\!\alpha.f^{m(n+1)} y \\
&\stackrel{\text{def}}{=} \quad \mathrm{c}_{m(n+1)}^\alpha
\end{aligned}$$

which ends our proof. □

As an example, we will now present a term representing the function $f : \mathbb{N}\!\to\!\mathbb{N}$ such that $f(n) = n^3 + 3n^2 + 1$ (problem 4 of [2]).

EXAMPLE 5.3.6 For each type variable α the term F^α defined by

$$\lambda n\!:\!\mathrm{Nat}^\alpha.\ \oplus^\alpha (\oplus^\alpha(\otimes^\alpha(\otimes^\alpha nn)n)(\otimes^\alpha(\otimes^\alpha c_3^\alpha n)n))c_1^\alpha$$

is such that $\vdash F^\alpha\!:\!(\mathrm{Nat}^\alpha\!\to\!\mathrm{Nat}^\alpha)$ and $F^\alpha c_n^\alpha \to_\beta^* c_{n^3+3n^2+1}^\alpha$. □

At this stage of affairs we will skip the proof of the properties we stated for F^α, as they are direct consequences of the analogous properties for $\oplus^\alpha$ and $\otimes^\alpha$.

Generalizing numerals to other systems of the Lambda Cube proves to be an interesting task. In λP there is not much that we can do; $\lambda\underline{\omega}$, however, turns out to be quite interesting.

When we defined numerals in $\lambda\!\to$, we did so for a type variable α. Looking at this construction from the point of view of Pure Type Systems, this means that all the properties we can prove regarding numerals can only be proven for a context $\alpha\!:\!*$; furthermore, each type variable gives birth to an independent numeral system, and it is not even clear what interaction between numerals defined over different type variables should mean.

In $\lambda\underline{\omega}$, the construction is much more natural. This is the weakest system in which we can build functions from types into types; as a consequence, it is the first system where we can consider the type Nat^*—that is, the type of the numerals over $*$.

It is easy to see that, due to the existence of the product rule $\langle \Box, \Box, \Box \rangle$, in this system we can build the types $(* \to *)$ and $(* \to *) \to (* \to *)$; it seems quite natural, then, to define the numeral C_n as being $\lambda f : (* \to *) \lambda x : *. f^n x$. Notice that we can now define terms $\oplus$, $\otimes$ and F that act on these numerals in the same way that $\oplus^\alpha$, $\otimes^\alpha$ and F^α above did; the difference is that properties about them can be proven from the empty context (problem 20 in [2]).

Obviously, in all systems containing $\lambda\underline{\omega}$ the same construction is possible.

As for $\lambda 2$, things also get interesting, though in a different way. Here, we cannot define a "natural" numeral system as in $\lambda\underline{\omega}$, but we can also define numerals independently of the type variable: we just use abstraction and define c_n as being $\lambda\alpha : *. \mathrm{c}_n^\alpha$. The constructions given for $\lambda\!\to$ have to be somewhat adapted, but we can still define terms $\oplus^2$, $\otimes^2$ and F^2 with properties analogous to their correspondent terms in $\lambda\!\to$.

Besides, in $\lambda 2$ we can canonically represent data types, therefore reproducing more complex kinds of computation than that done just in the natural numbers. The details of this representation can be found in [3].

In particular, we can define an operator that executes the recursion (problem 14.2 of [2]); that is, there is a term R such that

$$\vdash R : (\Pi\alpha : *. [\alpha \to (\mathrm{Nat}^2 \to \alpha \to \alpha) \to \mathrm{Nat}^2 \to \alpha])$$

and R satisfies

$$\begin{array}{rcl} R\alpha ab\mathrm{c}_0 & \to_\beta^* & a \\ R\alpha ab\mathrm{c}_{n+1} & \to_\beta^* & b\mathrm{c}_n(R\alpha ab\mathrm{c}_n) \end{array}$$

Chapter 6

Conclusions

In this work we introduced λ-calculus as a model of computation and developed it in two directions, obtaining two other important models.

We defined combinatory algebras in Chapter 3, which generalize λ-calculus insofar as terms that are identified in Λ need not be the same in every combinatory algebra, and showed how the constructions in [1] could be generalized to arbitrary combinatory algebras. We then proved the corresponding generalizations of known theorems of λ-calculus, namely showing that any combinatory algebra can be viewed as a model of computation where the numerals are well defined and every partial recursive function (defined accordingly to Kleene) can be represented in an uniform way.

Obviously, the possibility of representing computability in an arbitrary combinatory algebra has some advantages over just doing it in λ-calculus; namely, terms in Λ are subject to very strict formation rules, even if we allow for a non-empty context. In an arbitrary combinatory algebra we can simply postulate the existence of a new element and define the way it acts (via the operation of the algebra) on other elements, thus adding expressive power to the system. In this way we can, for example, introduce the concept of *oracle* in a natural way: if $\mathcal{D}$ is a combinatory algebra, an element $\mathbf{M} \in \mathcal{D}$ represents the oracle $\varphi : \mathbb{N}^k \to \mathbb{N}$ iff, for every $n_1, \ldots, n_k \in \mathbb{N}$,

$$\mathbf{M} \ulcorner n_1 \urcorner \ldots \ulcorner n_k \urcorner \to \ulcorner \varphi(n_1, \ldots, n_k) \urcorner.$$

Other important (non-computable) functions can be introduced in a similar way, and results in relative computability can be thus obtained.

On the other side, we tried to restrict λ-calculus in order to allow for the use of different data types. The motivation for this comes, for example, from programming languages where variables can be assigned types and we want a function to be applied only to objects of a given type. This can be done in a simple and intuitive way through typed λ-calculus, as we showed in 4.1, and in a more general setting through the use of Pure Type Systems. Besides the computational aspects of the question, we explored a little of the intrinsic relationship that the systems of the Lambda Cube share with different known systems of logic, namely the result usually known as the Propositions-as-Types interpretation (Theorem 4.3.2).

Pure Type Systems can be further strenghtened by adding the possibility of recursive definitions and proofs; we didn't pursue this study here, but it is done in [4] in some detail.

Properties of these systems are illustrated in Chapter 5. We based ourselves on the list of problems given in [2], which we previously solved, and chose therefrom a selection broad enough to exemplify these properties and explore some of the main differences between the systems. Some of the solutions to these problems are already sketched in [3], but we developed them, sometimes arriving at altogether different answers.

Index

Bibliography

[1] Barendregt, H. P., *The Lambda Calculus*, Elsevier Science Publishers B. V., 1984

[2] Barendregt, H. P., *Problems in Type Theory*

[3] Barendregt, H. P., *Lambda Calculi with Types*, in Abramsky, S., Gabbay, D. M. and Maibaum, T. S. E. (eds.), *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press, pp.117-309.

[4] Barendregt, H. P. and Geuvers, H., *Proof-Assistants using Dependent Type Systems*, in Robinson, A. and Voronkov, A. (eds.), *Handbook of Automated Reasoning*, Elsevier Science Publishers B. V., 2001

[5] Engeler, E., *Foundations of Mathematics*, Springer-Verlag, 1993

[6] Odifreddi, P. G., *Classical Recursion Theory*, Vol. I, Elsevier Science Publishers B. V., 1999

[7] Smullyan, R., *To Mock a Mockingbird*, Oxford University Press, 1985

[8] Troelstra, A. and Schwichtenberg, H., *Basic Proof Theory*, 2nd edition, Cambridge University Press, 2000.