# Continuations and Martin-Löf's Type Theory

A thesis presented in partial fulfilment of the requirements of the degree of

Doctor of Philosophy
in
Computer Science

at Massey University, Albany, New Zealand.

Neil Leslie
February 29, 2000

# Abstract

We explain how to program with continuations in Martin-Löf's theory of types (M-LTT).

M-LTT is a theory designed to formalize constructive mathematics. By appealing to the Curry-Howard 'propositions as types' analogy, and to the Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic we can treat M-LTT as a framework for the specification and derivation of correct functional programs. However, programming in M-LTT has two weaknesses:

- we are limited in the functions that we can naturally express;
- the functions that we do write naturally are often inefficient.

Programming with continuations allows us partially to address these problems. The continuation-passing programming style is known to offer a number of advantages to the functional programmer. We can also observe a relationship between continuation passing and type lifting in categorial grammar.

We present computation rules which allow us to use continuations with inductively-defined types, and with types not presented inductively. We justify the new elimination rules using the usual proof-theoretic semantics. We show that the new rules preserve the consistency of the theory.

We show how to use well-orderings to encode continuation-passing operators for inductively defined types.

# Acknowledgements

An earlier version of some of the material in Chapter 6 appeared as [70].

I would like to thank:

- Peter Kay of Massey University's Albany campus, and Steve Reeves of Waikato University, for providing invaluable support and guidance;
- Ross Renner, and the School of Mathematical and Computing Sciences at Victoria University of Wellington, for indulging me with time and money to visit Waikato University to talk with Steve;
- Mhairi for caring (again) that I should finish a thesis;
- Keir and Ailidh for not caring about theses at all.

I would also like to acknowledge the debt which I owe to Aaron T. Beck.

# Contents

# List of Rules

# List of Programs

# List of Figures

# Introduction

This thesis shows how we can extend Martin-Löf's Type Theory to enable ourselves to program with continuations in a natural way.

In Part I we give an an introduction to M-LTT, and explain its utility. Chapter 1 presents the framework of M-LTT, paying particular attention to the principles by which we explain and justify the rules. In Chapter 2 we build up a basic framework of useful types. M-LTT is a theory designed to formalize constructive mathematics. By appealing to the Curry-Howard 'propositions as types' analogy, and to the Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic we can treat M-LTT as a framework for the specification and derivation of correct functional programs. However, programming in M-LTT has two weaknesses:

- we are limited in the functions that we can naturally express;
- the functions that we do write naturally are often inefficient.

In Part II we introduce continuations and discuss their uses. We see that programming with continuations allows us partially to address the issues of expressiveness and efficiency of functional programming. Chapter 4 points to an analogy between type-lifting in categorial grammar and continuation passing.

In Part III we explain how to add rules for computing with continuations in M-LTT. We do this by presenting a new form of elimination rule. We are careful to justify these rules in the same way that the usual elimination rules are justified. We give examples for some types, and we compare the new rules with the usual ones. Chapter 5 deals with types which are not inductively defined; Chapter 6 deals with inductively defined types which are common in programming. In Chapter 7 we show how we can use a non-canonical constant for the well-orderings to implement the continuation-passing, non-canonical constants for the inductively defined types in the same way we can implement the structurally recursive non-canonical constants for inductively defined types using the usual non-canonical constant for the well-orderings.

In Chapter 8 we present some small examples.

Finally, in Part IV we draw some conclusions and make suggestions for further work.

## Typesetting

The thesis was typeset in LaTeX2e using Wenzel Matiaske's thesis document class.

Fonts were set using Walter Schmidt's mathppl and Sebastian Rahtz' helvet packages.

Proof figures were produced using Paul Taylor's prooftree macros.

Some of the floating objects were produced using Anselm Lingau's float package.

Some of the figures were rotated using Sebastian Rahtz and Leonor Barroca's rotating package.

Frank Mittelbach's varioref package was used for some of the references.

The bibliography style was generated using Patrick W. Daly's makebst program.

*The LaTeX Companion*[1] proved invaluable, and provided the odd moment of

[1] Goosens, M., F. Mittelbach, and A. Samarin. *The LaTeX Companion.* Addison-Wesley, Reading, Massachusetts, USA, 1994.

humour.[2]

## Typesetting large rules

A number of rather large inference rules are used in this thesis. Typesetting these presents some difficulties. There are a number of ways in which one may present inference rules. My own preference is to present rules after the fashion of [77] and [110] (and, indeed, [42] and [95]), with the premisses arranged side-by-side. I find this presentation to be the most readable. Readability comes at the expense of size, and rules presented in this way can become too wide to fit onto a page. Such over-wide rules have been rotated and placed at the end of the chapter in which they appear. This, unfortunately, reduces readability. An alternative is to follow the practice of [88], where the premisses are arranged vertically. This is more compact, and avoids the problem of over-wide rules. However, I believe that, in general, rules presented in this way are less readable. A third alternative is to follow the presentation of [4], which, again, uses vertically stacked premisses, and which I find less readable still.

[2] For example, on page *v* 'LaTeX is not difficult to learn.'

# Part I

# Martin-Löf's Type Theory

# 1. Syntax, judgement and inference

In this chapter and in Chapter 2 we introduce and describe M-LTT. The rules of M-LTT are presented as a system of natural deduction inference rules which relate judgements of various forms. Before we explain the judgement forms and the inference rules we must discuss the syntax of expressions, so we begin in §1.1 by explaining the theory of aritied expressions. Then we explain the judgement forms in §1.2. In §1.3 we explain the syntax of the proof rules. We then proceed in §1.4 to explain how we give a proof-theoretic justification of the inference rules, after the fashion of Dummett. This approach to the justification of the rules is greatly different in spirit for that of, for example [101]. We do not explain our rules indirectly with reference to some external 'formal semantics', but directly and informally. It is important that we understand how the rules of M-LTT are justified because M-LTT is an *extensible* theory. We are allowed to add new types, and new ways to form types. The new material that we add to the theory is just as 'primitive' or basic as material that anyone else has added. This is to be contrasted with other systems, for example various higher-order logics [11] or Girard's System **F** [43, 45, 48], where we are given a fixed collection of 'primitive' types and we proceed to define other types in terms of these ones. We postpone the presentation of particular types until Chapter 2.

In Part III we introduce elimination rules of an unusual form, and we wish to emphasise that these can be justified, directly, in exactly the same way as the usual elimination rules are. Our motivation for this is that we make use of computation rules of an unusual form as we wish to provide ourselves with a natural way to write more efficient programs.

Martin-Löf states in [79] that M-LTT was originally

> 'intended to be a full scale system for formalizing intuitionistic mathematics as developed, for example, in the book by Bishop 1967'.[1]

For computer scientists the crucial aspect of the theory is that it is constructive:[2] that is, the theory has an *algorithmic* content. Once we have recognised this algorithmic content we can use the theory for programming and program development. It should be stressed that a belief in the utility of M-LTT in computer science does *not* depend on our believing that classical mathematics is flawed, nor on our holding a revisionist approach to mathematics in general, merely on a recognition that constructive mathematics is the appropriate mathematics for computing. To quote Michael Dummett:

> 'the important question for us is whether constructive mathematics is adequate for applications.'

Having said this, however, we view M-LTT as an *intuitionistic* theory in a stronger sense than merely being a theory to do constructive mathematics in. Much of the work that we do in this thesis relies on an understanding of how the laws of M-LTT can be justified. Because we view M-LTT as a *foundational* theory, and because we are intuitionists and do not believe in an infinite regress of formal explanations, we expect the explanation of the theory to be *informal*. We follow Brouwer here and admit the unformalizability of 'basic' notions [14]. While the explanation that we use to justify the rules of M-LTT must, at the end of the day, be informal, it must also be very careful. It is our intention to achieve Kreisel's 'informal rigour' [66] in our explanations. Taking care to justify the theory in a constructive fashion is worthwhile because, again quoting Michael Dummett, we believe that:

[1] Bishop 1967 is [12].

[2] We will generally treat 'constructive' and 'intuitionistic' as synonyms: the distinctions made in, for example, [13] are not of immediate relevance to us.

> 'the resulting versions of the theories indeed prove more apt for their applications.'

## 1.1 Rules of syntax: aritied expressions

Before we start to deal with the semantic aspects of the theory we need to build up an unambiguous theory of syntax. We do this using the theory of aritied expressions, originally suggested by [38]. We identify:

- abstraction;
- application;
- combination; and
- selection;

as the ways in which we form mathematical expressions. These concepts are all at the level of *syntax*. In particular we must be careful not to confuse the purely syntactic abstraction and application with $\lambda$-abstraction and function application (see § 2.1.6 on page 40).

We are motivated to do all this because we wish to rule out 'expressions' like $\mathsf{sin}(\mathsf{sin})$ and $3(+, 4)$ as being utterly nonsensical, and because we need to have a decidable equality on expressions.

Arities themselves are as follows:

- $\mathrm{o}$ is an arity;
- if $\alpha$ and $\beta$ are arities then $\alpha \mathrel{\ominus\!\!\!\rightarrow} \beta$ is an arity;[3]
- if $\alpha_1, \ldots, \alpha_n$ are arities then $\alpha_1 \otimes \ldots \otimes \alpha_n$ is an arity.

[3] The symbol $\mathrel{\ominus\!\!\!\rightarrow}$ has been chosen avoid confusion with the half-a-dozen other arrows we use and to provide visual analogy with $\otimes$.

The intended interpretation is:

- o is the arity of simple, saturated expressions;
- $\alpha \ominus\!\!\rightarrow \beta$ is the arity of an abstraction;
- $\alpha_1 \otimes \ldots \otimes \alpha_n$ is the arity of a combination.

The arities give us a very simple set of rules for how bits of syntax may be plugged together. In this respect they have a similarity to a simple type system. The syntax that we use for arities themselves reflects this similarity.[4]

Arities are associated with expressions as follows:

- variables and constants are given to us with an arity;
- if $\alpha$ is the arity of the variable $x$ and $\beta$ the arity of the expression $b$ then $\alpha \ominus\!\!\rightarrow \beta$ is the arity of the abstraction $(x)b$;
- if $\alpha$ is the arity of the expression $a$ and $\alpha \ominus\!\!\rightarrow \beta$ is the arity of the expression $f$ then $\beta$ is the arity of the application $f(a)$;
- if $\alpha_1, \ldots, \alpha_n$ are the arities of the expressions $a_1, \ldots, a_n$ respectively then $\alpha_1 \otimes \ldots \otimes \alpha_n$ is the arity of the combination $(a_1, \ldots, a_n)$.
- if $\alpha_1 \otimes \ldots \otimes \alpha_n$ is the arity of the expression $a$ then $\alpha_i$, where $i$ is between 1 and $n$, is the arity of the selection $a.i$, the $i$th component of $a$.

[4] An alternative syntax for arities is:

- () is the arity of simple, saturated expressions;
- if $\alpha$ and $\beta$ are arities then $\alpha\beta$ is an arity, the arity of an abstraction;
- if $\alpha_1, \ldots, \alpha_n$ are arities then $(\alpha_1, \ldots, \alpha_n)$ is an arity, the arity of a combination.

This syntax, although less readable in some ways, has the advantage that it gives a visual representation of the 'shape' of the 'holes' to be filled to form a saturated expression.

The rules dealing with arities restrict the expressions that we can form: since sin has arity $\circ \ominus\!\!\!\longrightarrow \circ$, sin(sin) is ruled out; however arities do not prevent us from forming *meaningless* expressions. For example, natrec (see § 2.4.1 on page 58) has arity $\circ \otimes (\circ \otimes \circ \ominus\!\!\!\longrightarrow \circ) \otimes \circ \ominus\!\!\!\longrightarrow \circ$, and nil (see § 2.4.2 on page 60) has arity $\circ$. Hence, if $x$ and $y$ both have arity $\circ$, then:

$$\mathsf{natrec}(\mathsf{nil}, (x, y)\mathsf{nil}, \mathsf{nil})$$

is an expression of arity $\circ$. This expression is, however, devoid of *meaning*. It is only when we have built the theory of types that we can discuss the meanings of expressions.

## Definitions

We can make abbreviatory definitions of the form:

$$\mathsf{definiendum} =_{\mathsf{def}} \mathsf{definiens}$$

The definiendum is a new constant. The definiens is an expression which must not contain free variables, and in which the definiendum must not occur.

The definiendum has the same arity as the definiens.

Such definitions are merely abbreviatory and do not allow us to write functions.

## Syntactic equality between expressions

We have a notion of equality between syntactic expressions, which we denote by $\equiv$. For two expressions to be syntactically equal they must be of the same arity.

Syntactic equality is reflexive, symmetric and transitive. Formally:

- if $a$ is an expression then $a \equiv a$

- if $a \equiv b$ then $b \equiv a$
- if $a \equiv b$ and $b \equiv c$ then $a \equiv c$

A definiendum and its definiens are syntactically equal. Formally:

- if $a =_{\mathsf{def}} b$ then $a \equiv b$.

We distinguish between $=_{\mathsf{def}}$ and $\equiv$, whereas, for example, [75, 88] do not. There is an asymmetry between the definiendum and the definiens, and the action of defining is different from the observation of equality. Thus the concepts denoted by $=_{\mathsf{def}}$ and $\equiv$ are different.

Combinations made from syntactically equal parts are themselves syntactically equal. Formally:

- if $a_1 \equiv b_1 : \alpha_1, \ldots, a_n \equiv b_n : \alpha_n$ then $(a_1, \ldots, a_n) \equiv (b_1, \ldots, b_n) : \alpha_1 \otimes \ldots \otimes \alpha_n$

Taking a combination apart and putting it back together again in the same way does not change it. Formally:

- if $a : \alpha_1 \otimes \ldots \otimes \alpha_n$ then $(a.1, \ldots, a.n) \equiv a : \alpha_1 \otimes \ldots \otimes \alpha_n$

The same selections from syntactically equal combinations are themselves syntactically equal. Formally:

- if $a \equiv b : \alpha_1 \otimes \ldots \otimes \alpha_n$ then $a.i \equiv b.i : \alpha_i$

Combining an expression with others and then selecting it does not change the expression. Formally:

- if $a : \alpha$ then $(b_1, \ldots, b_l, a, b_m, \ldots, b_n).(l+1) \equiv a : \alpha$

The application of syntactically equal expressions to syntactically equal expressions are syntactically equal. Formally:

- if $f \equiv g : \alpha \ominus\!\!\!\rightarrow \beta$ and $a \equiv b : \alpha$ then $f(a) \equiv g(b) : \beta$

For applications and abstractions we have rules corresponding to $\alpha$-congruence, $\beta$- and $\eta$-reduction and the $\xi$ rule of the $\lambda$-calculus [6]. If $a$ and $b$ are expressions and $x$ a variable then $b[x := a]$ is the expression formed by replacing occurrences of $x$ in $b$ with $a$, taking care not to capture free variables occurring in $a$.

The names of bound variables are not important. Formally:

($\alpha$) if $x : \alpha$ and $y : \alpha$ are variables, and $b : \beta$ then $(x)b \equiv (y)(b[x := y]) : \alpha \ominus\!\!\!\rightarrow \beta$

Abstractions over syntactically equal terms are syntactically equal. Formally:

($\xi$) if $x : \alpha$ and $a \equiv b : \beta$ then $(x)a \equiv (x)b : \alpha \ominus\!\!\!\rightarrow \beta$

Abstraction of an application to the same variable is syntactically equal to the applied expression. Formally:

($\eta$) if $x : \alpha$ and $f : \alpha \ominus\!\!\!\rightarrow \beta$ then $(x)(f(x)) \equiv f : \alpha \ominus\!\!\!\rightarrow \beta$, $x$ not free in $f$

Application of an abstraction is substitution. Formally:

($\beta$) if $x : \alpha$, $a : \alpha$ and $b : \beta$ then $((x)b)(a) \equiv b[x := a] : \beta$

The relation $\equiv$ is decidable. This is a crucial property: if we were not able to decide syntactic equality between expressions we should not, for example, even be able to decide whether an inference rule can be applied to a given expression.

### Practical syntax

Having developed the theory of aritied expressions with great care we now proceed to use a slightly more relaxed concrete syntax to enhance readability.

We will often write definitions as:

$$c(x_1, \ldots, x_n) =_{\mathrm{def}} d$$

rather than:

$$c =_{\mathrm{def}} (x_1, \ldots, x_n)d$$

We will often use the $\eta$-rule to write:

$$f(x_1, \ldots, x_n)$$

rather than:

$$(y)f(x_1, \ldots, x_n, y)$$

In particular, the non-canonical constant associated with a type $T$ (see § 1.4.4 on page 25) will typically take an argument of type $T$ as its final argument. The non-canonical constant associated with the type of natural number is natrec (see § 2.4.1 on page 58) so we may write:

$$\mathsf{natrec}(d, e)$$

rather than:

$$(n)\mathsf{natrec}(d, e, n)$$

Sometimes, of course, it it clearer to write an unsaturated expression out explicitly as an abstraction, so sometimes we will do this.

We will also freely write:

$$(x_1, \ldots, x_n)f$$

rather than:

$$(x_1) \ldots (x_n)f$$

We will adopt the 'usual' rules for precedence and associativity of operators to omit brackets, a convention we have already adopted in this section.

## 1.2 Forms of judgement

The inference rules of M-LTT relate *judgements*. This is also the case with the inference rules of logic, although in logic we are used to using only one judgement form. In classical logic we use the judgement that a proposition $P$ is true. In constructive logic, because the notion of *proof* is prior to that of *truth*, we use the judgement that a proposition $P$ has a proof. Usually when we present logical rules we leave the judgement involved implicit (see § 2.3.2 on page 48).

In M-LTT we deal very carefully with the notion of judgement and consider four judgement forms:

- being a type;
- being an object of a type;
- being equal types;
- being equal objects of a type.

We explain these in turn.

### 1.2.1 Judgement of being a type

The first judgement form that we consider is the judgement of being a type, which we write:

$$A \text{ type}$$

We justify the judgement $A$ type by explaining what the *canonical* objects of the type are, and when two canonical objects are equal. A canonical object is also called a *value*. Informally a canonical object is one which we can see directly to be an object of the type. We will return to this in §1.4.4.

### 1.2.2 Judgement of being an object of a type

The second judgement form that we consider is the judgement of being an object of a type, which we write:

$$a : A$$

We justify the judgement $a : A$ by showing that $a$ can be computed to a value of type $A$.

### 1.2.3 Judgement of two types being equal

The third judgement form that we consider is the judgement of two types being equal types, which we write:

$$A = B$$

We justify the judgement $A = B$ by showing that values of type $A$ are values of type $B$, and equal values of type $A$ are equal values of type $B$, and *vice versa*.

### 1.2.4 Judgement of two objects being equal

The fourth judgement form that we consider is the judgement of two objects being equal objects of a type, which we write:

$$a = b : A$$

We justify the judgement $a = b : A$ by showing that $a$ and $b$ can be computed to the same value of type $A$.

### 1.2.5 Hypothetical judgements

So far we have described only categorical judgements. We can introduce hypothetical judgements, that is judgements which are dependent on other judgements. Suppose $J_1$ is a judgement. Then:

$$\begin{array}{c}[J_1]\\ \vdots\\ J_2\end{array}$$

is a hypothetical judgement, which tells us that we can make the judgement $J_2$ on condition that we can make the judgement $J_1$. A typical hypothetical judgement is that one type is a family over another type:

$$\begin{array}{c}[x : A]\\ \vdots\\ C(x)\ \mathsf{type}\end{array}$$

This is the judgement that $C(x)$ is a type, on condition that $x$ is an object of type $A$. A hypothetical judgement allows us to introduce a variable, and defines its scope. We can form hypothetical judgements with more than one hypothesis, and with 'hypothetical hypotheses'. One of the premisses to the $\Sigma$ elimination rule (Rule 2.12 on page 37) is:

$$\begin{array}{c}\left[\begin{array}{c}x : A\\ y(w) : B(w)[w : A]\end{array}\right]\\ \vdots\\ d(x,y) : C(\mathsf{pair}(x,y))\end{array}$$

This is the judgement that $d(x,y)$ is an object of type $C(\mathsf{pair}(x,y))$, given:

- $x$ is an object of type $A$, and
- $y(w)$ is an object of type $B(w)$, given
  - $w$ is an object of type $A$.

### 1.2.6 Other interpretations of the judgement forms

We have given the typical[5] interpretation of the judgement forms. Using the Brouwer-Heyting-Kolmogorov (BHK) interpretation of intuitionistic logic [64] and the Curry-Howard propositions-as-types analogy [51] we can also give two other important interpretations:

- a logical interpretation;
- a computational interpretation.

In the logical interpretation we read:

- $A$ type as '$A$ is a proposition';
- $a : A$ as '$a$ is a proof witness of the proposition $A$';
- $A = B$ as '$A$ and $B$ are equal propositions';
- $a = b : A$ as '$a$ and $b$ are equal proof witnesses of the proposition $A$'.

In the computational interpretation we read:

- $A$ type as '$A$ is a specification';
- $a : A$ as '$a$ is a program which meets the specification $A$';
- $A = B$ as '$A$ and $B$ are equal specifications';
- $a = b : A$ as '$a$ and $b$ are equal programs which meet the specification $A$'.

In §2.3.2 we explain how we can represent the connectives of intuitionistic logic directly in M-LTT.

Given these three readings of the judgements we see that in M-LTT, *in one system*, we have:

[5] In the sense 'pertaining or relating to a type or types', rather than 'distinctive, characteristic'.

- a logic;
- a specification language;
- a programming language.

This is the key to why M-LTT is of especial interest to computer scientists, a theme we return to in §2.3.4.

## 1.3 Syntax of proof rules

Our inference rules relate judgements. We will present rules as informal rule schemas like:

$$\dfrac{\begin{array}{c}\left[\begin{array}{c} H_{(1,1)}[H_{(1,1,1)}, \ldots, H_{(1,1,m)}] \\ \ldots \\ H_{(1,p)}[H_{(1,p,1)}, \ldots, H_{(1,p,q)}] \end{array}\right] \\ \vdots \\ P_1 \end{array} \qquad \ldots \qquad \begin{array}{c}\left[\begin{array}{c} H_{(n,1)}[H_{(n,1,1)}, \ldots, H_{(n,1,s)}] \\ \ldots \\ H_{(n,r)}[H_{(n,r,1)}, \ldots, H_{(n,r,t)}] \end{array}\right] \\ \vdots \\ P_n \end{array}}{C}\ \text{Rule name}$$

Rule 1.1: An inference rule schema

The judgements above the inference line are the *premisses*. The rule allows us to discharge the hypotheses inside square brackets. When the rule schema is instantiated in a particular proof the *conclusion* $C$ will remain dependent on any undischarged hypotheses. The scope of names within the schema generally goes from top to bottom, and left to right, except that in hypothetical hypotheses it works right to left. Thankfully only a few rules, such as Rule 2.64 on page 66 and Rule 2.86 on page 75, require this complexity. The name of the rule may be omitted, as this can often reduce clutter in proofs.

It is also common practice to reduce clutter in rules by omitting judgements of the form $A$ type in rules, especially when the conclusion of the rule is one of the 'higher' judgements, and the omitted judgement can be inferred easily. This practice can be criticised for being sloppy, and can be defended because it allows us to make the important content of the rule clearer.

## 1.4 Justification of the rules

In this section we shall explain how the rules of M-LTT are themselves justified. The justification that we use is a proof-theoretic one, and was originally applied to the rules of logic. So we approach the task of justifying the rules of M-LTT *via* the task of justifying the rules of logic. Because we use fewer judgement forms in logic the explanation of the justification of the logical rules is also simpler.

The rules of M-LTT are generally presented in these classes:

- formation rules;
- introduction rules;
- rules for the introduction of equal objects;
- computation rules;
- elimination rules;
- rules for the equality of expressions formed using non-canonical constants.

In contrast, when we present natural deduction rules for logic we only present:

- introduction rules;
- elimination rules.

With the exception of the computation rules, the extra rules in M-LTT reflect the extra judgement forms that we make use of in M-LTT. The computation rules are the key to the justification of the elimination rules. To understand why we present the rules in these forms, and how they are justified we must look at the proof theoretic justification of the rules of deduction, especially following the work of Dummett [26, 27, 28, 32, 33], particularly [30]; and Prawitz [96]. Dummett's work is motivated by the desire to understand what a theory of meaning is, and hence to use this understanding to justify the rules of deduction. Dummett's work is not primarily aimed at constructive type theory, but it provides the appropriate basis for us to justify the rules of type theory. Dummett's ideas are implicit in the usual justifications given in the literature (for example, [77, 78, 88, 110]). We shall make the connection much more explicit as, in Chapters 5, 6 and 7, we wish to introduce unconventional computation and elimination rules. We will treat these rules as having the same status as the conventional rules. Thus we need to be very clear about how the conventional rules were justified.

### 1.4.1 Using 'meaning is use'

We focus on the introduction, computation and elimination rules, because the formation rules are not a source of confusion, and the rules for the introduction of equal objects and for the equality of expressions formed using non-canonical constants are analogous to the introduction and elimination rules, respectively. We start from Wittgenstein's dictum: 'meaning is use' [119].[6] Dummett writes [28]:

> 'The meaning of a mathematical statement determines and is exhaustively determined by its *use*. The meaning of such a statement cannot be, or contain as an ingredient, anything which is not manifest in the use made of it, lying solely in the

[6] Actually: 'the meaning of a word is its use in the language.' §43 of Part I of [119].

> mind of the individual who apprehends that meaning: if two individuals agree completely about the use to be made of the statement, then they agree about its meaning. The reason is that the meaning of a statement consists solely in its rôle as an instrument of communication between individuals, just as the powers of a chess-piece consist solely in its rôle in the game according to the rules.'[7]

Dummett continues to point out that knowledge of meaning cannot, if it is to be well-founded, ultimately consist of the ability to state or verbalise meaning, and that:

> 'a grasp of the meaning of a mathematical statement must, in general, consist of a capacity to use that statement in a certain way, or to respond in a certain way to its use by others.'

One consequence of adopting the slogan 'meaning is use', might seem to be that use is unchallengeable, that is, *however* we choose to use a mathematical statement is, and must be, acceptable. Dummett argues that such an attitude can only be supported if we assume a holistic[8] approach. Under such a holistic view it is not legitimate to ask for the meaning of an individual statement on its own: meaning of individual statements is only understandable in terms of meaning of the language as a whole.

The two main arguments for rejecting holism rely on the need for language to be a vehicle of communication, and on the need for a language to be learnable. If the meaning of an utterance depended not only on the utterance itself and its parts but on the whole language then it would never be possible to be sure that communication of the utterance had been effective, unless it could be shown that the whole language had been communicated (and understood). If individual statements were not understandable in themselves it would not be possible to learn a language incrementally:

[7] Emphasis in the original.

[8] Dummett's notion of holism is discussed in rather more detail in [104].

there would always be the possibility that, as more of the language was learned, the meanings of previously learned parts would have to be altered.

### Holism and classical logic

Adopting the slogan 'meaning is use', and rejecting holism leads us swiftly away from classical logic. As an example of holism in classical logic consider a proof of Peirce's Law in Gentzen's system **NK** [42]. Peirce's Law is:

$$((P \supset Q) \supset P) \supset P$$

Peirce's Law involves only implications. Gentzen's **NK** is his **NJ** augmented with a rule for double negation or excluded middle. Suppose we choose the latter extension.

The rule of excluded middle is:

$$\dfrac{}{P \vee \neg P}\;\mathrm{ExMid}$$

Rule 1.2: *Tertium Non Datur*

We can derive Peirce's Law in **NK**:

$$\dfrac{\dfrac{\dfrac{}{P \vee \neg P}\;\mathrm{ExMid} \qquad [P]^2 \qquad \dfrac{[(P \supset Q) \supset P]^1 \qquad \dfrac{\dfrac{\dfrac{[\neg P]^2 \quad [P]^3}{\bot}\;\neg\,\mathrm{E}}{Q}\;\bot\,\mathrm{E}}{P \supset Q}\;\supset \mathrm{I}^3}{P}\;\supset \mathrm{E}}{P}\;\vee\,\mathrm{E}^2}{((P \supset Q) \supset P) \supset P}\;\supset \mathrm{I}^1$$

Rule 1.3: An **NK** proof of Peirce's Law

Since **NK** is **NJ** plus the excluded middle rule, and there is no **NJ** proof of Peirce's Law, any **NK** proof of Peirce's Law must use excluded middle. Thus it appears that an understanding of the rules involving implication is *not* sufficient to understand the classical concept of implication: we also have to understand rules involving negation and disjunction. Perhaps this is only because we made a foolish choice in expressing **NK** as we did. Perhaps we should extend **NJ** with a rule corresponding to Peirce's Law:

$$\begin{array}{c}[P \supset Q]\\ \vdots \\ \dfrac{P}{P}\,\mathrm{PL}\end{array}$$

Rule 1.4: Peirce's Law as an inference rule

Now we find that we can prove $P \vee \neg P$, by making use of Peirce's Law:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[P]^2}{P \vee \neg P}\vee\mathrm{I\,l} \qquad [P \vee \neg P \supset \bot]^1}{\bot}\supset\mathrm{E}}{\neg P}\neg\,\mathrm{I}^2}{P \vee \neg P}\vee\mathrm{I\,r}}{P \vee \neg P}\mathrm{PL}^1$$

Rule 1.5: A proof of $P \vee \neg P$

Thus, taking **NK** to be **NJ** plus Rule 1.4, rather than **NJ** plus Rule 1.5 has merely moved the problem about. Now we cannot understand disjunction without understanding implication also. This example gives us some evidence that a degree of holism is required to give a 'meaning is use' explanation of the connectives of classical logic. As we have seen the meanings of the connectives appear to depend on rules in which they themselves do not figure, and this would not appear to simply be an artefact of one particular presentation of the rules of classical logic. Consequently, believing that 'meaning is use', and rejecting holism leads us away from classical logic.

### 1.4.2 Adopting a molecular approach

Having rejected holism we seek instead a molecular theory where the meaning of a connective is explained using *part* of the use of the connective and focus on Gentzen's assertion (in §5.13 of [42]) that:

> 'The introductions represent, as it were, the 'definitions' of the symbols concerned, and the eliminations are no more, in the final analysis, than the consequences of these definitions.'

### 1.4.3 The challenge of tonk

It is well known that we must be careful about 'the final analysis'. We must avoid the absurdities of Prior's tonk [97]. The introduction rules for tonk are:

$$\frac{A}{A \,\mathsf{tonk}\, B}\ \mathsf{tonk}\ \text{Intro. L}$$

Rule 1.6: tonk introduction on the left

and:

$$\frac{B}{A \,\mathsf{tonk}\, B}\ \mathsf{tonk}\ \text{Intro. R}$$

Rule 1.7: tonk introduction on the right

These rules are, in themselves, unobjectionable as they are just the same as the usual rules for $\vee$ introduction. The elimination rules for tonk are:

$$\frac{A \ \mathsf{tonk}\ B}{A}\ \mathsf{tonk}\ \text{Elim. L}$$

Rule 1.8: tonk elimination on the left

and:

$$\frac{A \ \mathsf{tonk}\ B}{B}\ \mathsf{tonk}\ \text{Elim. R}$$

Rule 1.9: tonk elimination on the right

These are also, in themselves, quite unobjectionable. They are just the same as the usual rules we find in elementary logic texts for & elimination. The problem comes when we combine tonk introduction and elimination. We get this figure:

$$\dfrac{\dfrac{A}{A \ \mathsf{tonk}\ B}\ \text{Tonk Intro}}{B}\ \text{Tonk Elim}$$

Rule 1.10: A proof of $B$ from $A$

This is clearly unacceptable. The clue to how we can prevent ourselves from writing such absurdities comes from the observation that this proof is not a normal proof as it involves the introduction of a connective immediately followed by its elimination. Normalising Rule 1.10 would give us:

$$\begin{array}{c} A \\ \vdots \\ B \end{array}$$

Rule 1.11: Normalising Rule 1.10

Clearly, the connective tonk clearly fails to enjoy 'harmony' between the introduction and elimination rules. We know that proof normalisation corresponds to program evaluation, that is to computation (see, for example, [48]). So it is reasonable to expect that the computation rules are the key to harmony. The next sections of this chapter seek to clarify this.

### 1.4.4 Clarifying our notion of rules

In order to proceed we need to clarify some of our notions about what the forms of the rules are. In particular we need to clarify what we mean when we describe a rule as an introduction rule or as an elimination rule. Once again, for simplicity we will illustrate this section with rules of logic (although we keep the typical interpretation in mind too).

First we suppose that we have:

- a supply of propositional (or type) variables, e.g. $P, Q, R, S \ldots$
- a supply of constants, e.g. $\bot, \neg, \vee, \&, \supset$, List, $\Pi, \Sigma \ldots$

Propositional (or type) formulas are either variables or expressions whose principal connective is one of the constants. Variables are *prime* formulas; other formulas are *non-prime*.

Our first definition of an introduction rule is that it is a rule whose conclusion is the judgement that we have a proof of a *non-prime* formula.

Our first definition of an elimination rule is that it is a rule which has as a premiss the judgement that we have a proof of a *non-prime* formula.

Under these definitions one rule may be *both* an introduction rule *and* an elimination rule:

$$\frac{A \,\&\, B}{\neg(\neg A \vee \neg B)}\,\mathrm{DM}$$

Rule 1.12: A De Morgan law as an inference rule

Furthermore a rule may be both an introduction rule and an elimination rule for the *same* connective:

$$\frac{A \mathbin{\&} B}{B \mathbin{\&} A}$$

Rule 1.13: Symmetry of &

and:

$$\frac{\neg B \qquad A \supset B}{\neg A}$$

Rule 1.14: *Modus Tollendo Tollens*

Rules 1.12, 1.13 and 1.14, while introduction rules by our definition above, are not candidates for self-justification. We need to have a more restrictive notion of what sorts of introduction rule we can use to define the meanings of the connectives.

### 1.4.5 Self-justifying introduction rules

To impose these restrictions we follow the line of reasoning given by Dummett in Chapter 11 of [30], which asserts:

> 'The minimal demand we should make on an introduction rule intended to be self-justifying is that its form be such as to guarantee that, in any application of it, the conclusion will be of higher logical complexity than any of the premisses and than any discharged hypothesis. We may call this the "complexity condition".'

Dummett describes a rule as 'single-ended' for a given connective if it is an introduction rule, but not an elimination rule, or if it is an elimination rule, but not an introduction rule.

We can further demand that an introduction rule intended to be self-justifying be single-ended, as any rule which is not single-ended can be decomposed into rules which are single-ended.

So, by this point, we have explained that we can take single-ended introduction rules which meet the complexity condition to be self-justifying. We will follow common practice and describe introduction rules of this special form as introduction rules *simpliciter*. In the typical interpretation the introduction rules introduce *values* or *canonical* objects of a type; in the logical interpretation the introduction rules introduce *immediate* or *canonical* proof witnesses. We have not yet overcome the challenge of tonk, as we must still explain *how* we produce elimination rules. Our strong hint has been that the computation rules are involved, and this is what we explain next.

### 1.4.6 Computation rules

In M-LTT the computation rules tell us how we may simplify expressions formed using *non-canonical* constants. The form of the computation rules for a particular non-canonical constant relies on our grasp of what form of computation is appropriate:

- for the enumerated types (see §2.2) the appropriate form of computation is case-analysis;
- for the inductively defined types (see § 2.4 on page 57) the appropriate form of computation is usually taken to be structural recursion;
- for the well-orderings the appropriate form of computation is usually taken to be well-founded recursion.

The form of the computation rules is:

$$\frac{a_1 \longrightarrow b_1 \quad \ldots \quad a_q(\ldots) \longrightarrow b_q}{f(a_1, \ldots, a_n) \longrightarrow b_q}\ f\ \text{Comp.}$$

Rule 1.15: The form of a computation rule

This rule tells us that the expression $f(a_1, \ldots, a_n)$ can be computed to $b_q$, if $a_1$ can be computed to $b_1$, and so on. The expressions to be computed may depend on the results of previous computations. Suppose some type $T$ has $n$ introduction rules i.e. there are $n$ different forms that the canonical objects of type $T$ can take. The non-canonical constant associated with the type $T$ will take $n + 1$ arguments. Typically, there will be $n$ computation rules, each one corresponding to one of the introduction rules. For example, the $+$ types (§ 2.1.1 on page 33) have two introduction rules, and there are two computation rules for the non-canonical constant. The empty type (§ 2.2.2 on page 43) is one exception to this rule.

The computation rules describe *lazy* evaluation (call-by-need) [92]. We are not compelled to use lazy evaluation. However, if we used eager evaluation then we would produce elimination rules with premisses stronger than are really needed.

### 1.4.7 Elimination rule

Now we can view the elimination rule for a type $T$ as a rule which takes one premiss of the form $t : T$ and has as a conclusion a judgement about the type of an expression formed using the non-canonical constant associated with $T$. The other premisses of the elimination rule simply tell us what judgements we need to be able to make in order to justify the conclusion. Typically, some of them are judgements about the well-formedness of some type which appears in the rule, most often about the type of the conclusion. Other premisses give judgements which allow us to utilise the computation rules for the non-canonical constant.

Suppose we have a computation rule like:

$$\frac{a \longrightarrow \mathsf{constr}(c) \qquad b(c) \longrightarrow b'}{\mathsf{ncc}_A(b, \ldots, a) \longrightarrow b'}\ \mathsf{ncc}_A \text{ Comp.}$$

Rule 1.16: A computation rule

where:

- $\mathsf{constr}$ is one of the constructors for the type $A$;
- $\mathsf{ncc}_A$ is the non-canonical constant associated with the type $A$.

Further, suppose that one of the introduction rules for $A$ is

$$\frac{c : C}{\mathsf{constr}(c) : A}\ A \text{ Intro.}$$

Rule 1.17: An example of an introduction rule

One of the premisses of the elimination rule is the judgement $a : A$. This means that $a$ evaluates to a canonical object of type $A$. Each introduction rules tells us what one of the canonical forms is. One of the canonical forms for type $A$ is $\mathsf{constr}(c)$ where $c$ is of type $C$. The computation rule tells us how we can use one of the auxiliary arguments to $\mathsf{ncc}_A$ to compute with $c$. The judgement:

$$\begin{array}{c} [y : C] \\ \vdots \\ b(y) : B(\mathsf{constr}(y)) \end{array}$$

is then enough to ensure that $\mathsf{ncc}_A(b, \ldots, \mathsf{constr}(c)) : B(\mathsf{constr}(c))$.

We perform the same analysis for each of the introduction and computation rules to justify an elimination rule like:[9]

[9] As suggested in §1.3 we would typically suppress the judgement $A$ type.

$$\dfrac{A\ \mathsf{type} \qquad a : A \qquad \begin{matrix}[x : A]\\ \vdots \\ B(x)\ \mathsf{type}\end{matrix} \qquad \ldots \qquad \begin{matrix}[y : C]\\ \vdots \\ b(y) : B(\mathsf{constr}(y))\end{matrix}}{\mathsf{ncc}_A(a, \ldots, b) : B(a)}$$

Rule 1.18: A proof rule

In this way we can see the elimination rule as a consequence of the introduction rule and the computation rules. Further illustrations will be provided in the next chapter where we present some useful types.

If we consider alternative computation rules then we will derive different elimination rules. This is what we will, in fact, do in Part III.

Now, at last, we have a principled reason for rejecting tonk without throwing our logical baby away with the bath water: the tonk elimination rules are clearly not elimination rules which we can derive from the two tonk introduction rules given, and from our understanding of how to make use of a canonical proof witness of $A$ tonk $B$.

### 1.4.8 Summary of the relationship between the rules

The relationship between the introduction, computation and elimination rules can be summarised:

- Introduction rules of a special form are taken to be self-justifying. They introduce *canonical* objects of the type involved.
- Computation rules tell us how to compute with the non-canonical constant associated with a type. We formulate these by considering the introduction rules in conjunction with our intuitive grasp of computation.
- The elimination rule will have as a premiss a judgement that we have some object of the type in question. The elimination rule is essentially a consequence of the introduction and computation rules. The

elimination rule can also be thought of as a typing rule for expressions formed with a non-canonical constant.

## 1.5 Chapter Summary

In this chapter we have introduced a number of key notions:

- the theory of aritied expressions;
- the judgement forms used in M-LTT;
- the logical and computational readings of the judgement forms;
- the syntax of the proof rules, and of the computation rules;
- the proof-theoretic justification of the rules.

# 2. Some useful types, and applications

In Chapter 1 we introduced the framework of M-LTT. In this chapter we will present some types. The types can be organised into a number of groups:

- the types which have a logical interpretation:
  - disjoint union of two types: $+$,
  - disjoint union of a family of types: $\Sigma$,
  - Cartesian product of a family of types: $\Pi$,
  - the empty type,
- the enumerated types;
- some inductively defined types common in programming:
  - the natural numbers,
  - polymorphic lists,
  - binary trees,
- well-order types: $W$;
- equality types: $\mathsf{EQ}$, $\mathsf{ID}$.

This organisation is not completely consistent. For example, the empty type is an enumerated type and has a logical interpretation, and the $\Pi$ types have a significant functional interpretation in addition to their logical interpretation.

In this chapter we will also illustrate some uses for these types.

## 2.1 'Logical' types

In this section we will present the rules for dealing with the $+$, $\Sigma$, $\Pi$, and empty types. These types, *via* the Curry-Howard propositions as types analogy [51] and the Brouwer-Heyting-Kolmogorov (BHK) [64] interpretation of intuitionistic logic, allow us to represent constructive logic *directly* in type theory. We outline this representation in §2.3.2.

### 2.1.1 Disjoint union of two types: $+$

If $A$ and $B$ are types we can form their disjoint union. So, the formation rule for the $+$ types is:

$$\frac{A \text{ type} \qquad B \text{ type}}{A + B \text{ type}} + \text{Form}$$

Rule 2.1: + formation

The introduction rules are:

$$\frac{a : A}{\mathsf{inl}(a) : A + B} + \text{Intro L}$$

Rule 2.2: + introduction on the left

$$\frac{b : B}{\mathsf{inr}(b) : A + B} + \text{Intro R}$$

Rule 2.3: + introduction on the right

It is implicit in both these rules that $A + B$ is well-formed, that is that $A$ and $B$ are both types. For clarity the well-formedness judgements are often omitted.

Rules 2.2 and 2.3 are single-ended introduction rules which meet the complexity condition, and thus we can take them to be self justifying. Next we construct computation rules for a non-canonical constant, and then we construct an elimination rule.

The non-canonical constant associated with the + types is called when, and the rules for its evaluation are as follows:

$$\frac{f \longrightarrow \mathsf{inl}(l) \qquad d(l) \longrightarrow d'}{\mathsf{when}(d, e, f) \longrightarrow d'}\ \mathsf{when}\ \text{Comp}$$

Rule 2.4: when computation 1

and:

$$\frac{f \longrightarrow \mathsf{inr}(r) \qquad e(r) \longrightarrow e'}{\mathsf{when}(d, e, f) \longrightarrow e'}\ \mathsf{when}\ \text{Comp}$$

Rule 2.5: when computation 2

The computation rules formalise our understanding of how we can compute with an expression which can itself be reduced to one of the forms $\mathsf{inr}(a)$ or $\mathsf{inl}(b)$. As we stated in §1.4.6 the computation rules describe lazy computation. In when computation one of the auxiliary arguments will not be evaluated. Notice also that the computation rules tell us nothing about how to compute $\mathsf{when}(d, e, f)$ if $f$ evaluates to anything other than $\mathsf{inr}(a)$ or $\mathsf{inl}(b)$.

The elimination rule for + is:

$$\frac{f : A + B \qquad \begin{array}{c}[z : A + B] \\ \vdots \\ C(z)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[x : A] \\ \vdots \\ d(x) : C(\mathsf{inl}(x))\end{array} \qquad \begin{array}{c}[y : B] \\ \vdots \\ e(y) : C(\mathsf{inr}(y))\end{array}}{\mathsf{when}(d, e, f) : C(f)}\ + \text{E}$$

Rule 2.6: + elimination

The conclusion of this rule is a judgement about the type of $\mathsf{when}(d, e, f)$. The first premiss:

$$f : A + B$$

tells us that $f$ evaluates to a canonical object of type $A + B$: either the left injection of an object of type $A$, or the right injection of an object of type $B$. The third and fourth premisses:

$$\begin{array}{c} [x : A] \\ \vdots \\ d(x) : C(\mathsf{inl}(x)) \end{array}$$

and:

$$\begin{array}{c} [y : B] \\ \vdots \\ e(y) : C(\mathsf{inr}(y)) \end{array}$$

ensure that $d(x)$ and $e(y)$ evaluate to values in a suitable type. A suitable type, in this instance, is a type which is a family over $A + B$. This explains the second premiss, and thus gives us the type of the expression $\mathsf{when}(d, e, f)$.

There are two rules for the equality of expressions formed with when:

$$\frac{a : A \qquad \begin{array}{c} [z : A + B] \\ \vdots \\ C(z)\ \mathsf{type} \end{array} \qquad \begin{array}{c} [x : A] \\ \vdots \\ d(x) : C(\mathsf{inl}(x)) \end{array}}{\mathsf{when}(d, e, \mathsf{inl}(a)) = d(a) : C(\mathsf{inl}(a))}$$

Rule 2.7: Equality of expressions involving when 1

and:

$$\frac{b : B \qquad \begin{array}{c}[z : A+B]\\ \vdots \\ C(z)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : B]\\ \vdots \\ e(y) : C(\mathsf{inr}(y))\end{array}}{\mathsf{when}(d, e, \mathsf{inr}(b)) = e(b) : C(\mathsf{inr}(b))}$$

Rule 2.8: Equality of expressions involving when 2

These rules are justified by considering the computation rules for when. Because evaluation is lazy we do not need to have a premiss involving $e$ in Rule 2.7, nor one involving $d$ in Rule 2.8.

We will not give such a detailed justification of the subsequent rules, nor present the rules for the equality of expressions formed using non-canonical constants, as these all, generally, fall into the same pattern.

### 2.1.2 Disjoint union of a family of types: Σ types

If $B$ is a family of types over $A$ we can form $\Sigma(A, B)$:

$$\frac{A\ \mathsf{type} \qquad \begin{array}{c}[x : A]\\ \vdots \\ B(x)\ \mathsf{type}\end{array}}{\Sigma(A, B)\ \mathsf{type}}\ \Sigma\ \text{Form}$$

Rule 2.9: Σ formation

The values in $\Sigma(A, B)$ are pairs:

$$\frac{a : A \qquad b(a) : B(a)}{\mathsf{pair}(a, b) : \Sigma(A, B)}\ \Sigma\ \text{I}$$

Rule 2.10: Σ introduction

The non-canonical constant associated with the Σ types is called split and its computation rule is:

$$\dfrac{c \longrightarrow \mathsf{pair}(a,b) \qquad d(a,b) \longrightarrow d'}{\mathsf{split}(d,c) \longrightarrow d'}\ \mathsf{split}\ \text{Comp}$$

Rule 2.11: split computation

The elimination rule is:

$$\dfrac{c : \Sigma(A,B) \qquad \begin{array}{c}[z : \Sigma(A,B)] \\ \vdots \\ C(z)\ \mathsf{type}\end{array} \qquad \begin{array}{c}\left[\begin{array}{c} x : A \\ y(w) : B(w)[w : A]\end{array}\right] \\ \vdots \\ d(x,y) : C(\mathsf{pair}(x,y))\end{array}}{\mathsf{split}(d,c) : C(c)}\ \Sigma\ \text{Elim}$$

Rule 2.12: Σ elimination

We justify this rule with the following argument. The premiss $c : \Sigma(A,B)$ tells us that $c$ evaluates to the form $\mathsf{pair}(a,b)$. The second premiss tells us that $C$ is a family of types over $\Sigma(A,B)$. The third premiss:

$$\begin{array}{c}\left[\begin{array}{c} x : A \\ y(w) : B(w)[w : A]\end{array}\right] \\ \vdots \\ d(x,y) : C(\mathsf{pair}(x,y))\end{array}$$

is a judgement which allows us to make use of the computation rule to make a judgement about the type of $\mathsf{split}(c,d)$. There is only one introduction rule for Σ, so we only require one such premiss.

We can use split to define fst:

$$\mathsf{fst}(p) =_{\mathsf{def}} \mathsf{split}((x,y)x, p)$$

and snd:

$$\mathsf{snd}(p) =_{\mathsf{def}} \mathsf{split}((x,y)y, p)$$

### 2.1.3 Non-dependent $\Sigma$

If $B$ is not a family of types over $A$, i.e. we can make the judgement $B$ type then we make the definition:

$$A \times B =_{\mathsf{def}} \Sigma(A, (x)B)$$

### 2.1.4 Cartesian product of a family of types: $\Pi$ types

If $B$ is a family of types over $A$ we can form $\Pi(A, B)$:

$$\frac{A\ \mathsf{type} \qquad \begin{array}{c}[x : A] \\ \vdots \\ B(x)\ \mathsf{type}\end{array}}{\Pi(A, B)\ \mathsf{type}}\ \Pi\ \text{Form.}$$

Rule 2.13: $\Pi$ formation

$$\frac{\begin{array}{c}[x : A] \\ \vdots \\ b(x) : B(x)\end{array}}{\lambda(b) : \Pi(A, B)}\ \Pi\ \text{Intro.}$$

Rule 2.14: $\Pi$ introduction

Again we have left implicit the judgements that $A$ is a type and $B$ a family of types over $A$. We follow the same pattern that we have used before to produce the associated computation rule and elimination rule.

$$\frac{f \longrightarrow \lambda(b) \qquad d(b) \longrightarrow d'}{\mathsf{funsplit}(d, f) \longrightarrow d'}\ \mathsf{funsplit}\ \text{Comp.}$$

Rule 2.15: funsplit computation

$$\dfrac{f : \Pi(A,B) \qquad \begin{matrix}[z : \Pi(A,B)]\\ \vdots \\ C(z)\,\mathsf{type}\end{matrix} \qquad \begin{matrix}[y(x) : B(x)[x : A]]\\ \vdots \\ a(y) : C(\lambda(y))\end{matrix}}{\mathsf{funsplit}(a,f) : C(f)}\ \Pi\text{ Elim.}$$

Rule 2.16: Π elimination

We may choose to use apply rather than the funsplit as the non-canonical constant, as this is more familiar to computer scientists.

We can make to following definition:

$$\mathsf{apply}(f,x) =_{\mathsf{def}} \mathsf{funsplit}((y)(y(x)),f)$$

We evaluate $\mathsf{apply}(f,a)$ as:

$$\dfrac{\dfrac{f \longrightarrow \lambda(b) \qquad \dfrac{b(a) \longrightarrow c}{(y)(y(a))b \longrightarrow c}\equiv,\beta}{\mathsf{funsplit}((y)(y(a)),f) \longrightarrow c}\ \mathsf{funsplit}\text{ Comp.}}{\mathsf{apply}(f,a) \longrightarrow c}=_{\mathsf{def}}$$

Rule 2.17: Evaluating $\mathsf{apply}(f,a)$

The use of syntactic equality is not, strictly, a step of computation. We have written it as if it were for clarity. We will also do this when we use syntactic equality in inference rules.

We can write this rule as apply computation:

$$\dfrac{f \longrightarrow \lambda(b) \qquad b(a) \longrightarrow c}{\mathsf{apply}(f,a) \longrightarrow c}\ \mathsf{apply}\text{ Comp.}$$

Rule 2.18: apply computation

From this we obtain an alternative Π elimination rule:

$$\frac{f : \Pi(A, B) \qquad a : A}{\mathsf{apply}(f, a) : B(a)}\ \Pi\ \text{Elim.}$$

Rule 2.19: Alternative Π elimination

### 2.1.5 Non-dependent Π

If $B$ is not a family of types over $A$, i.e. we can make the judgement $B$ type then we make the definition:

$$A \rightarrow B =_{\mathsf{def}} \Pi(A, (x)B)$$

### 2.1.6 The two abstractions

We have now seen two sorts of abstraction:

- the syntactic abstraction introduced in §1.1;
- $\lambda$ abstraction.

To grasp the difference between them consider the three expressions

- $\mathsf{succ}$
- $(x)(\mathsf{succ}(x))$
- $\lambda(\mathsf{succ})$

where succ is a constructor of the type Nat as described in §2.4.1 below. $(x)(\mathsf{succ}(x))$ and succ are *syntactically* equal. As succ is an unsaturated expression of arity $\mathsf{o} \ominus\!\!\!\rightarrow \mathsf{o}$ it does not make sense to ask what value succ has. On the other hand, $\lambda(\mathsf{succ})$ is *not* the same as succ. Syntactically, it is the application of $\lambda : (\mathsf{o} \ominus\!\!\!\rightarrow \mathsf{o}) \ominus\!\!\!\rightarrow \mathsf{o}$, to $\mathsf{succ} : \mathsf{o} \ominus\!\!\!\rightarrow \mathsf{o}$. So $\lambda(\mathsf{succ})$ is a saturated expression. We can show $\lambda(\mathsf{succ}) : \mathsf{Nat} \rightarrow \mathsf{Nat}$:

$$\cfrac{\cfrac{[x : \mathsf{Nat}]}{\mathsf{succ}(x) : \mathsf{Nat}}\ \mathsf{Nat}\ \mathrm{I}}{\lambda(\mathsf{succ}) : \mathsf{Nat} \rightarrow \mathsf{Nat}}\ \rightarrow \mathrm{I}$$

Rule 2.20: $\lambda(\mathsf{succ})$ is a function

Thus in M-LTT we make a distinction between succ the constructor and $\lambda(\mathsf{succ})$ the function.

## 2.2 Finite sets, or enumerated types

We can form enumerated types. The formation rule for an enumerated type with $n$ objects is:

$$\frac{}{\{x_1, \ldots, x_n\}\mathsf{type}}$$

Rule 2.21: Enumerated type formation

For each enumerated type there are $n$ introduction rules, where $i$ runs between 1 and $n$:

$$\frac{}{x_i : \{x_1, \ldots, x_n\}}$$

Rule 2.22: Enumerated type introduction

We will have $n$ rules for the introduction of equal elements of this form, where $i$ runs between 1 and $n$ :

$$\frac{}{x_i = x_i : \{x_1, \ldots, x_n\}}$$

Rule 2.23: Introduction of equal enumerated types

The general form of the non-canonical constant for the enumeration types is a case expression. In general $\mathsf{case}_{\{x_1,\ldots,x_n\}}$ will take $n+1$ arguments and will have $n$ computation rules like this, again where $i$ runs between 1 and $n$:

$$\frac{a \longrightarrow x_i \qquad\qquad d_i \longrightarrow d'}{\mathsf{case}_{\{x_1,\ldots,x_n\}}(d_1,\ldots,d_n,a) \longrightarrow d'}\ \mathsf{case}_{\{x_1,\ldots,x_n\}}\ \text{comp}$$

Rule 2.24: $\mathsf{case}_{\{x_1,\ldots,x_n\}}$ computation

So, each enumeration type will have an elimination rule of this form, with $n$ premisses of the form $d_i : C(x_i)$, where $i$ runs between 1 and $n$:

$$\frac{a : \{x_1,\ldots,x_n\} \qquad \begin{array}{c}[z : \{x_1,\ldots,x_n\}]\\ \vdots \\ C(z)\ \mathsf{type}\end{array} \qquad d_1 : C(x_1) \quad \ldots \quad d_n : C(x_n)}{\mathsf{case}_{\{x_1,\ldots,x_n\}}(d_1,\ldots,d_n,a) : C(a)}$$

Rule 2.25: Enumerated type elimination

And there will be $n$ rules for the equality of expressions formed with the non-canonical constant:

$$\frac{a : \{x_1,\ldots,x_n\} \qquad \begin{array}{c}[z : \{x_1,\ldots,x_n\}]\\ \vdots \\ C(z)\ \mathsf{type}\end{array} \qquad d_i : C(x_i)}{\mathsf{case}_{\{x_1,\ldots,x_n\}}(d_1,\ldots,d_n,a) = d_i : C(a)}\ \text{Eq Enum elim}$$

Rule 2.26: Equality of expressions involving $\mathsf{case}_{\{x_1,\ldots,x_n\}}$

### 2.2.1 Booleans

The type of Booleans, Bool, can be defined:

$$\mathsf{Bool} \quad =_{\mathsf{def}} \quad \{\mathsf{true}, \mathsf{false}\}$$

$$\text{if } b \text{ then } d \text{ else } e \quad =_{\text{def}} \quad \text{case}_{\text{Bool}}(d, e, b)$$

### 2.2.2 The empty type

The empty type is a special case of an enumeration type. The literature is curiously confused at this point. It is very tempting simply to generalise Rules 2.21...2.26 with the case when $n$ is 0. This is what Martin-Löf (page 65 of [77]) and Nordström *et al* (page 47 of [88]) do, and what Thompson invites us to do for ourselves (Exercise 4.20 on page 100 of [110]). Unfortunately this leads us to write a rule down like:

$$\frac{}{x_i : \{x_1, \ldots, x_0\}}$$

Rule 2.27: A putative empty type introduction rule

where $i$ is in the range $1 \ldots 0$. It is certainly not at all clear what this means. Here we might make an analogy with the following explanation of exponentiation: we evaluate $2^n$ by writing down $n$ twos and multiplying them all together. This explanation is inadequate because it fails to tell us how to evaluate $2^0$: writing down no twos and multiplying them all together seems a difficult task, and certainly does not explain why $2^0$ is 1. It is possible to argue that the sequence $1 \ldots 0$ is the empty sequence, but this merely begs the question. Why should we need to refer to the empty *sequence* to explain the empty *type*? We shall therefore deal with the empty type directly.

The formation rule is simple:

$$\frac{}{\{\}\ \text{type}}\ \text{Empty form}$$

Rule 2.28: {} formation

The empty type obviously has no introduction rules: it is empty. There can be no canonical object of the empty type.

We now have the task of explaining the form of the computation rules for $\mathsf{case}_{\{\}}$ and justifying the elimination rule for $\{\}$.

The elimination rule will have at least:

$$a : \{\}$$

and:

$$\begin{array}{c}[z : \{\}]\\ \vdots \\ C(z)\mathsf{type}\end{array}$$

as premisses, and

$$\mathsf{case}_{\{\}}(a) : C(a)$$

as a conclusion.

We expect that there will be no other premisses, as there are no introduction rules. Recall that the **meaning** of a typing judgement $a : A$ is that evaluating $a$ will give us a canonical element of $A$. So if we are to justify a rule with the judgement $\mathsf{case}_{\{\}}(a) : C(a)$ as a conclusion we are forced to explain how to evaluate $\mathsf{case}_{\{\}}(a)$, that is we must give a computation rule for $\mathsf{case}_{\{\}}$. This is another point at which the literature is confused.

On page 19 of [4] Backhouse *et al* state:

> 'Since there are no introduction rules there are no computation rules. Any attempt to evaluate $\emptyset - \mathsf{elim}(r)$ may thus be considered as a divergent computation.'

Where $\emptyset - \mathsf{elim}$ is $\mathsf{case}_{\{\}}$. Backhouse *et al* use the expression 'computation rule' to refer to the (type) rules for the equality of expressions involving non-canonical constants (for example Rules 2.45 and 2.46).

On page 66 of [77] Martin-Löf states:

> '...the set of instructions for executing a program of the form $R_0(c)$ is vacuous. It is similar to the programming statement `abort` introduced by Dijkstra[1].'

$R_0(c)$ is $\mathsf{case}_{\{\}}(c)$. It is implicit in Martin-Löf's presentation that there *is* a rule for the equality of expressions involving $\mathsf{case}_{\{\}}$.

Following exactly the same pattern as we have followed for all the other types we produce a computation rule for $\mathsf{case}_{\{\}}$ as:

$$\dfrac{d \longrightarrow d'}{\mathsf{case}_{\{\}}(a) \longrightarrow d'}\ \mathsf{case}_{\{\}}\ \text{comp}$$

Rule 2.29: $\mathsf{case}_{\{\}}$ computation

So we get this elimination rule:

$$\dfrac{a : \{\} \qquad \begin{array}{c}[z : \{\}] \\ \vdots \\ C(z)\ \mathsf{type}\end{array}}{\mathsf{case}_{\{\}}(a) : C(a)}\ \{\}\ \text{elim}$$

Rule 2.30: $\{\}$ elimination

Informally: if we have an object of the empty type then we can construct an object in any type.

We can also justify this rule for the equality of expressions involving $\mathsf{case}_{\{\}}$:

$$\dfrac{a : \{\} \qquad \begin{array}{c}[x : \{\}] \\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[z : \{\}] \\ \vdots \\ y(z) : C(z)\end{array}}{\mathsf{case}_{\{\}}(a) = y(a) : C(a)}\ \{\}\ \text{elim eq}$$

Rule 2.31: Equality of expressions involving $\mathsf{case}_{\{\}}$

[1] In [21]. (Martin-Löf's footnote.)

Informally: if we have a object of the empty type then we can construct an object equal to any object in any type. This rule is theone which Backhouse *et al* reject.

We have spent a fair amount of time discussing the empty type and its rules. The empty type would seem to be a simple type, and explaining its rules should be easy. The confusion in the literature seems to belie this. We might find this worrying as we have asserted that the explanation of rules, and the development of the theory is simple and straightforward: perhaps our semantics are not as simple as we would like to make out.

## 2.3 Applications

We have introduced a fair amount of material so far. It is worthwhile to pause here and sketch out some applications of what we have built up.

### 2.3.1 Defining $+_{df}$

We can use the $\Sigma$ and enumerated types to define the disjoint union of two types:

$$A +_{df} B =_{\mathrm{def}} \Sigma(\{0,1\}, \mathsf{case}_{\{0,1\}}(A,B))$$

We can show:

$$\cfrac{\cfrac{\cfrac{}{\{0,1\}\ \mathsf{type}}\ \text{Enum Form} \qquad \cfrac{[x:\{0,1\}] \qquad A\ \mathsf{type} \qquad B\ \mathsf{type}}{\mathsf{case}_{\{0,1\}}(A,B,x)\ \mathsf{type}}\ \{0,1\}\ \text{Elim.}^*}{\Sigma(\{0,1\}, \mathsf{case}_{\{0.1\}}(A,B))\ \mathsf{type}}\ \Sigma\ \text{Form}}{A +_{df} B\ \mathsf{type}}\ \equiv$$

Rule 2.32: Derivation of $+_{df}$ formation

We have used some sleight of hand here. The inference labelled '$\{0,1\}$ Elim.*' is *not* an instance of the $\{0,1\}$ elimination rule as the conclusion is a judgement of the form $T$ type, rather than of the form $t : T$. We will justify this sleight of hand when we discuss universes in §2.7.

We can present Rule 2.32 as the following derived rule:

$$\frac{A\ \mathsf{type} \qquad B\ \mathsf{type}}{A +_{df} B\ \mathsf{type}}\ +_{df}\ \mathrm{Form}$$

Rule 2.33: $+_{df}$ formation

which is exactly Rule 2.1.

We can show, again with a little sleight of hand:

$$\cfrac{\cfrac{\cfrac{}{0 : \{0,1\}}\,\{0,1\}\,\mathrm{I} \qquad \cfrac{a : A}{a : \mathsf{case}_{\{0,1\}}(A, B, 0)}}{\mathsf{pair}(0,(x)a) : \Sigma(\{0,1\}, \mathsf{case}_{\{0,1\}}(A, B))}\,\Sigma\,\mathrm{I}}{\mathsf{pair}(0,(x)a) : A +_{df} B}\,=_{\mathrm{def}}$$

Rule 2.34: $+_{df}$ introduction

We can construct a similar proof for $\mathsf{pair}(1,(x)b)$, and hence can define $\mathsf{inl}$ and $\mathsf{inr}$ as:

$$\begin{aligned} \mathsf{inl}(a) &=_{\mathsf{def}} \mathsf{pair}(0,(x)a) \\ \mathsf{inr}(b) &=_{\mathsf{def}} \mathsf{pair}(1,(x)b) \end{aligned}$$

Finally, we can define $\mathsf{when}$ as:

$$\mathsf{when}(d, e) =_{\mathsf{def}} \mathsf{split}((p,q)\mathsf{case}_{\{0,1\}}(d(q(p)), e(q(p)), p))$$

and we can then derive the elimination rule for $+_{df}$.

The fact that we *can* define $+_{df}$ in terms of $\Sigma$ and $\{0,1\}$ raises the issue of whether we *should*. We return to this theme at the start of Chapter 5.

### 2.3.2 Representing the logical constants

In this section we outline how we can use the Σ, Π,+, and empty types, *via* the Curry-Howard propositions as types analogy [51] and the Brouwer-Heyting-Kolmogorov (BHK) [64] interpretation of intuitionistic logic, to represent constructive logic directly in type theory.

#### Disjunction

Recall that in the BHK-interpretation of intuitionistic logic a (direct) proof of $A \vee B$ consists of either a proof of $A$, or a proof of $B$, along with an indication of which it is. A (canonical) object of type $A + B$ is either an object of type $A$, or an object of type $B$, together with a tag which tells us which it is. Hence we see that we can make the following definition:

$$A \vee B =_{\mathsf{def}} A + B$$

We now define two judgements: $A$ prop is a synonym for $A$ type, and $A$ true is an abbreviation for $a : A$. Now we can re-write Rules 2.2 and 2.3 as:

$$\frac{A \text{ true}}{A \vee B \text{ true}} \vee \text{ Intro L}$$

Rule 2.35: ∨ introduction on the left

and:

$$\frac{B \text{ true}}{A \vee B \text{ true}} \vee \text{ Intro R}$$

Rule 2.36: ∨ introduction on the right

If we suppose that the well-formedness of $C$ does not depend on a proof of $A \vee B$, then Rule 2.6 becomes:

$$\dfrac{A \vee B\ \mathsf{true} \qquad C\ \mathsf{prop} \qquad \begin{array}{c}[A\ \mathsf{true}]\\ \vdots \\ C\ \mathsf{true}\end{array} \qquad \begin{array}{c}[B\ \mathsf{true}]\\ \vdots \\ C\ \mathsf{true}\end{array}}{C\ \mathsf{true}}\ \vee\ \mathrm{Elim}$$

Rule 2.37: ∨ elimination

If we were further to suppose that all the propositions mentioned in our rules were well-formed then we could leave implicit the judgement true, and recover exactly the usual rules for disjunction.

If we were to use the definition of $+_{df}$ given in §2.3.1 we would recover exactly the explanation of disjunction given in [10].

### Universal quantification

Recall that in the BHK-interpretation of intuitionistic logic a (direct) proof of $(\forall x : A)B(x)$ consists of a method to construct a proof of $B(x)$ for an arbitrary object of type $A$. A (canonical) object of type $\Pi(A, B)$ is a function which, given a arbitrary $x$ of type $A$ constructs an object of type $B(x)$. Hence we see that we can make the following definition:

$$(\forall x : A)B(x) =_{\mathsf{def}} \Pi(A, B)$$

We can recover the usual logical rules for universal quantification by performing the same steps we did for disjunction *mutatis mutandis*.

### Existential quantification

Recall that in the BHK-interpretation of intuitionistic logic a (direct) proof of $(\exists x : A)B(x)$ consists of an object $a$ of type $A$, and a proof of $B(a)$. A (canonical) object of type $\Sigma(A, B)$ is an object of type $A$, paired with an object of $B(a)$. Hence we see that we can make the following definition:

$$(\exists x : A)B(x) =_{\mathsf{def}} \Sigma(A, B)$$

We can recover the usual logical rules for existential quantification by performing the same steps we did for disjunction *mutatis mutandis*.

### Conjunction

Recall that in the BHK-interpretation of intuitionistic logic a (direct) proof of $A \,\&\, B$ consists of a proof of $A$, and a proof of $B$.

A (canonical) object of $A \times B$ is an object of type $A$, paired with an object of type $B$. Hence we see that, in this case, we can make the definition:

$$A \,\&\, B =_{\mathrm{def}} A \times B$$

When we try to recover the usual logical rules we get this rule as an elimination rule:

$$\dfrac{A \,\&\, B \qquad \begin{array}{c}[A, B]\\ \vdots\\ C\end{array}}{C'}\ \&\ \text{Elim}$$

Rule 2.38: & elimination

Choosing one of the conjuncts for $C$ allows us to derive the pair of rules familiar from elementary logic textbooks.

### Implication

Recall that in the BHK-interpretation of intuitionistic logic a (direct) proof of $A \supset B$ consists of a method to construct a proof of $B$, given an arbitrary proof of $A$.

A (canonical) object of $A \to B$ is a function which, given a arbitrary $x$ of type $A$ constructs an object of type $B$. Hence we see that in this case we can make the following definition:

$$A \supset B =_{\text{def}} A \to B$$

We can recover the usual logical rules for implication by performing the same steps we did for disjunction *mutatis mutandis*. The rule *Modus Ponens* is obtained from Rule 2.19.

### Falsum

Recall that in the BHK-interpretation of intuitionistic logic the is no (direct) proof of $\bot$.

There is no (canonical) object of the empty type, so we see than we can make the following definition:

$$\bot =_{\text{def}} \{\}$$

We can recover the usual logical rules for absurdity by performing the same steps we did for disjunction *mutatis mutandis*. Rule 2.30 becomes *Ex Falsum Quodlibet*.

### Negation

We follow the usual practice and define:

$$\neg A =_{\text{def}} A \supset \bot$$

Hence:

$$\neg A \equiv A \to \{\}$$

### Bi-implication

We also define:

$$A \leftrightarrow B =_{\text{def}} (A \supset B) \,\&\, (B \supset A)$$

### Summary of the logical types

In this section we have shown that we can formally define the logical constants *directly* in M-LTT. This simplicity of these definitions should not come as a surprise as we have always been working with an informal correspondence. A proposition is something of which we would recognise a (direct) proof, and a type is something of which we would recognise a (canonical) object.

## 2.3.3 Dealing with donkey sentences

Now we look at a very different use of M-LTT.

Formalising sentences in natural language is the bane of the lives of both beginning logic students and researchers in computational linguistics. Geach's 'donkey sentences' help us to understand why there are problems for both the naïve and the sophisticated. Consider the following sentence (Example 22 from [41]):

> 'If Smith owns a donkey then he beats it.'[2]

The 'If... then... ' form indicates to us to formalise this sentence with an implication. We expect that the implicans will be an existentially quantified statement like:

$$(\exists d)(\mathsf{donkey}(d)\ \&\ \mathsf{owns}(\mathsf{Smith}, d))$$

The implicate will them tell us that Smith beats the donkey. So we expect to see something like:

$$\mathsf{beats}(\mathsf{Smith}, X)$$

[2] In the later literature we find out that Smith's first name is Pedro.

where $X$ is the donkey. We come up against a problem here: if we are working in a classical logic we have lost the donkey underneath the existential quantifier. Essentially, the donkey is no longer in scope. We might propose to get the donkey back into scope with a formalisation like:

$$(\exists d)((\mathsf{donkey}(d)\ \&\ \mathsf{owns}(\mathsf{Smith}, d)) \supset \mathsf{beats}(\mathsf{Smith}, d))$$

Unfortunately this looks more like a formalisation of:

> 'There is a donkey, which if Smith owned, he would beat.'

This is an example of a group of problems related to anaphora resolution. A number of solutions have been proposed to overcoming problems like this, essentially based on model-theoretic semantics with some intensional flavour in the tradition of Montague [23, 80]. The best known are Kamp's Discourse Representation Theory (DRT) [59, 60], and Barwise and Perry's Situation Semantics [8, 9]. There are also approaches using dynamic logic [53], such as [34].[3] DRT, for example, solves the problem above by introducing discourse referents and discourse representation structures (DRS's), with their rather complicated scoping rules. Most of the rules relating to DRS's are justified by reference to their utility in handling some phenomenon of (some particular) natural language. On one hand this means that there is empirical evidence to support them; on the other they often appear *ad hoc*.

Instead of adding intensions as an afterthought we follow the analysis of donkey sentences given by [106] and [98] and treat the existential quantifier as a $\Sigma$ type. The implicans is then:

$$\Sigma(\mathsf{donkey}, (d)\mathsf{owns}(\mathsf{Smith}, d))$$

[3] There is surely some irony here, in that many computer scientists advocate the use of declarative programming languages on precisely the grounds that state-based languages are too hard to reason about.

Notice that if $a$ is an inhabitant of this type then $\mathsf{fst}(a)$ is precisely the donkey which we had previously lost. The implicate can then be $\mathsf{beats}(\mathsf{Smith}, \mathsf{fst}(a))$, and so the whole sentence is represented by:

$$\Pi(\Sigma(\mathsf{donkey}, (d)\mathsf{owns}(\mathsf{Smith}, d)), (a)(\mathsf{beats}(\mathsf{Smith}, \mathsf{fst}(a))))$$

This analysis was particularly smooth and simple. There was nothing *ad hoc* or especially designed to handle anaphora resolution in M-LTT to begin with. We have not had to add *any* extra notions to the ones that we have already to model the phenomenon we are interested in. If nothing else Ockham's razor is on our side. Our ability to handle donkey sentences is a testament to the value of the simple and careful way in which M-LTT has been built. Of course, as anyone with any experience of computational linguistics might point out, one example, no matter how elegant does not prove very much at all. Ranta's monograph [98], however, provides many more examples and shows that constructive type theory can have useful applications in computational linguistics and natural language understanding.

### 2.3.4 Program development

We are now also at the stage where we can start to explain how M-LTT can be used for program development. We suppose that we work in a goal-directed fashion, and start from a specification. A specification for a program describes the relationship between the input of the program and the output. For example, we might have the following informal specifications:

1. take two numbers number, say $x$ and $n$, and return a number $y$, such that $y^n = x$;

2. take a sequence of numbers and return a sequence consisting of the same numbers arranged in ascending order;

3. take a string and a description of a grammar and state whether or not the string is generated by the grammar;

4. take a program and input for it and state whether computation of the program will terminate on the input.

The first two specifications have an obvious formalisation like:

$$\Pi(\mathsf{In}, (i)\Sigma(\mathsf{Out}, (o)P(i,o)))$$

where In and Out are the types of the input and output, respectively, and $P(i,o)$ expresses the desired relationship.

A proof of such a proposition can then be identified with a program which meets the specification. We have extended the 'propositions-as-types' analogy with 'propositions-as-specifications' and 'proofs-as-programs'. We therefore have in M-LTT a *single* framework in which we can:

- express specifications;
- express programs;
- show that programs meet their specifications.

The second pair of specifications have a slightly different form from the first pair. The obvious formalisation of 3 is:

$$\Pi(\mathsf{String}\ \&\ \mathsf{Grammar}, (i)(G(i) + \neg G(i)))$$

If we were to treat this as a proposition of *classical* logic then the proof is trivial, but lacks any computational content. It is clear that this makes classical logic less well-suited to reasoning about programs, at least if we wish to retain the simple 'proofs-as-programs' interpretation. The brevity of the classical proof remains attractive, and there is a body of work on the relationship between classical and intuitionistic logic and how to extract constructive content from classical proofs (see, for example, §3 of Chapter

2 of [114], and [39, 85, 86]). Continuation-passing-style transformations, of the form described in Chapter 3 for much of the basis for this work.

M-LTT gives us a balanced, elegant and integrated framework for program development. In other frameworks we typically expect to use three different languages: a specification language, a programming language and a logical language: each more-or-less-well suited to its individual task and each typically ill-suited to co-exist with the others. As with our ability to handle donkey sentences we see great advantages arise from clarity and simplicity.

In M-LTT it is clear that specifications are *not* programs, and ought not to be thought of as such.[4] The informal specifications that we presented above were chosen partly to illustrate that, informally, this is the case, too. It is clear that there is no *algorithmic* content to specification 1: what part of this tells us how to compute an $n$th root? And we would have to stretch the notion of 'program' beyond breaking point to accept specification 4 as a program.

Our 'proofs-as-programs' interpretation has one important corollary: in M-LTT typability implies termination. Since the judgement:

$$p : S$$

tells us that $p$ has a *value* of type $S$ it tells us that computation of $p$ *terminates*. This is a very desirable property, but the price to be paid for it is to limit the class of programs that we can write. The work in Chapter 6 is an attempt to address part of this problem.

There are elements of the 'proofs-as-programs' paradigm which one might consider undesirable. In particular, an object of a $\Sigma$ type is a pair consisting of (under this interpretation) a program and the evidence that it is correct. While we are looking for the correctness proof it is of great interest, but, as soon as we have it we are only really interested in the program.

[4] [55] comes to the same conclusion, although *via* a different route.

We have not yet introduced enough types to do much in the way of programming. In the following section we shall introduce inductively defined types which the computer scientist will recognize as typical of the inductive data types found in modern (functional) programming languages.

## 2.4 Inductively defined types

We look at three inductively defined types:

- the natural numbers;
- polymorphic lists;
- binary trees.

For each of these types the non-canonical constant that we present in this section embodies structural recursion. Structural recursion on the natural numbers is called primitive recursion, after [63]. The use of these different names by mathematicians and computer scientists may, perhaps, be explained by observing that mathematicians often only work *directly* with one inductively defined data type: the natural numbers; whereas computer scientists typically expect to work *directly* with many different inductively defined data types: the natural numbers, lists, trees, syntactic terms and so on.

Structural recursion is a weak form of recursion. It is well-known that there are functions which are not definable using structural recursion: an Ackermann function[5] being the usual example given. As we shall see, defining programs using structural recursion has one crucial advantage: termination essentially comes for free.

From one point of view we have a very pleasant situation: we are satisfied with the rules that we have, and that we can extend the theory securely.

[5] Ackermann introduced this function in [1], and a number of variants of it have appeared since. See remark 1.2.8 of [47].

We also have the property that if a program is well-typed then its evaluation terminates with a value of the type. From another point of view the situation is not so good: the functions which we construct using structural recursion are often not very efficient, and we know that there are techniques from functional programming which allow us to write more efficient programs. For example, quicksort is not expressed using structural recursion over lists. As computer scientists, we *should* be concerned with efficiency and expressiveness.

There are two obvious ways to address this problem: indirect and direct approaches. In the indirect approach we represent the type we are interested in in another, where recursion over the new type corresponds to a different recursion over the type we are interested in. Examples, of well-founded recursion, can be found in [69, 87, 89]. The indirect approach can be rather hard work. The other approach is to use directly a different form of computation. There is no good reason why structural recursion has to be the *only* recursion that we use. If we do choose a different form of recursion then we have the burden of producing and justifying a new elimination rule.

### 2.4.1 Natural numbers

The formation rule for the natural numbers is:

$$\frac{}{\mathsf{Nat\ type}}\ \text{NatFormation}$$

Rule 2.39: Natural number formation

The introduction rules for the natural numbers are:

$$\frac{}{\mathsf{zero} : \mathsf{Nat}}\ \text{Nat Introduction 1}$$

Rule 2.40: Natural number introduction 1

$$\frac{n : \mathsf{Nat}}{\mathsf{succ}(n) : \mathsf{Nat}}\ \text{Nat Introduction 2}$$

Rule 2.41: Natural number introduction 2

These rules tell us that zero is a canonical object (a *value*) of type Nat, and that if $n$ is an object of type Nat (not necessarily a canonical object), then succ($n$) is a canonical object of type Nat.

The structural recursion operator for the natural numbers is usually called natrec, and the rules for its behaviour are:

$$\frac{n \longrightarrow \mathsf{zero} \qquad d \longrightarrow d'}{\mathsf{natrec}(d, e, n) \longrightarrow d'}\ \mathsf{natrec}\,\text{Comp. 1}$$

Rule 2.42: natrec computation 1

$$\frac{n \longrightarrow \mathsf{succ}(m) \qquad e(m, \mathsf{natrec}(d, e, m)) \longrightarrow e'}{\mathsf{natrec}(d, e, n) \longrightarrow e'}\ \mathsf{natrec}\,\text{Comp. 2}$$

Rule 2.43: natrec computation 2

Evaluation is about finding a value: in both rules above $n$ is reduced to a *canonical* object of the type we have in mind. These rules tell us nothing about what to do in case $n$ cannot be so evaluated. Evaluation *means* reducing to canonical form. It is from this reduction that we can infer the types required in the elimination rules.

The elimination rule tells us in what circumstances we can make a judgement about a computation involving a natural number. The elimination rule, which is justified by a consideration of the introduction and the computation rules, is:

$$\dfrac{n:\mathsf{Nat} \qquad \begin{array}{c}[x:\mathsf{Nat}]\\ \vdots\\ C(x)\ \mathsf{type}\end{array} \qquad d:C(\mathsf{zero}) \qquad \begin{array}{c}\begin{bmatrix} p:\mathsf{Nat}\\ q:C(p)\end{bmatrix}\\ \vdots \\ e(p,q):C(\mathsf{succ}(p))\end{array}}{\mathsf{natrec}(d,e,n):C(n)}\ \text{Nat Elim.}$$

Rule 2.44: Natural number elimination

And we also have rules like:

$$\dfrac{C(\mathsf{zero})\ \mathsf{type} \qquad d:C(\mathsf{zero})}{\mathsf{natrec}(d,e,\mathsf{zero})=d:C(\mathsf{zero})}\ \text{Natrec} =$$

Rule 2.45: Equality of expressions involving natrec 1

$$\dfrac{n:\mathsf{Nat} \qquad \begin{array}{c}[x:\mathsf{Nat}]\\ \vdots\\ C(x)\ \mathsf{type}\end{array} \qquad d:C(\mathsf{zero}) \qquad \begin{array}{c}\begin{bmatrix} p:\mathsf{Nat}\\ q:C(p)\end{bmatrix}\\ \vdots \\ e(p,q):C(\mathsf{succ}(p))\end{array}}{\mathsf{natrec}(d,e,\mathsf{succ}(n))=e(n,\mathsf{natrec}(d,e,n)):C(\mathsf{succ}(n))}\ \text{Natrec} =$$

Rule 2.46: Equality of expressions involving natrec 2

In Rule 2.45 we need only know that $C(\mathsf{zero})$ is a type, whereas in Rule 2.46 we need to know that $C$ is a family of types over Nat.

### 2.4.2 Polymorphic lists

The list formation rule is:

$$\dfrac{A\ \mathsf{type}}{\mathsf{List}(A)\ \mathsf{type}}\ \text{List Form}$$

Rule 2.47: List formation

Lists are very like natural numbers:

$$\frac{A \ \mathsf{type}}{\mathsf{nil} : \mathsf{List}(A)} \text{ List Intro. 1}$$

Rule 2.48: List introduction 1

For clarity we retain the judgement $A$ type in this rule.

$$\frac{a : A \qquad as : \mathsf{List}(A)}{\mathsf{cons}(a, as) : \mathsf{List}(A)} \text{ List Intro. 2}$$

Rule 2.49: List introduction 2

The judgement $A$ type need not be explicitly stated in this rule as $A$ must be a type before the judgement $a : A$ makes sense.

These rules tell us that nil is a canonical object (a *value*) of type List($A$) (so long as $A$ is a type), and that if $a$ is an object of type $A$ and $as$ an object of type List($A$) ($a$ and $as$ need not be canonical objects), then $\mathsf{cons}(a, as)$ is a canonical object of type List($A$).

The structural recursion operator for polymorphic lists is usually called listrec, and the rules for its behaviour are:

$$\frac{l \longrightarrow \mathsf{nil} \qquad d \longrightarrow d'}{\mathsf{listrec}(d, e, l) \longrightarrow d'} \text{ listrec Comp.}$$

Rule 2.50: listrec computation 1

$$\frac{l \longrightarrow \mathsf{cons}(a, as) \qquad e(a, as, \mathsf{listrec}(d, e, as)) \longrightarrow \epsilon'}{\mathsf{listrec}(d. e, l) \longrightarrow e'} \text{ listrec Comp.}$$

Rule 2.51: listrec computation 2

The elimination rule tells us in what circumstances we can make a judgement about a computation involving a list. The elimination rule, which is justified by a consideration of the introduction and the computation rules, is:

$$\dfrac{l : \mathsf{List}(A) \qquad \begin{array}{c}[x : \mathsf{List}(A)]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad d : C(\mathsf{nil}) \qquad \begin{array}{c}\begin{bmatrix} a : A \\ as : \mathsf{List}(A) \\ q : C(as)\end{bmatrix}\\ \vdots \\ e(a, as, q) : C(\mathsf{cons}(a, as))\end{array}}{\mathsf{listrec}(d, e, l) : C(l)}\ \text{List Elim}$$

Rule 2.52: List elimination

The rules for listrec equality are:

$$\dfrac{C(\mathsf{nil})\ \mathsf{type} \quad d : C(\mathsf{nil})}{\mathsf{listrec}(d, e, \mathsf{nil}) = d : C(\mathsf{nil})}\ \mathsf{listrec} = 1$$

Rule 2.53: Equality of expressions involving listrec 1

and:

$$\dfrac{h : A \quad t : \mathsf{List}(A) \qquad \begin{array}{c}[x : \mathsf{List}(A)]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad d : C(\mathsf{nil}) \qquad \begin{array}{c}\begin{bmatrix} a : A \\ as : \mathsf{List}(A) \\ q : C(as)\end{bmatrix}\\ \vdots \\ e(a, as, q) : C(\mathsf{cons}(a, as))\end{array}}{\mathsf{listrec}(d, e, \mathsf{cons}(h, t)) = e(a, as, \mathsf{listrec}(d, e, t)) : C(\mathsf{cons}(h, t))}$$

Rule 2.54: Equality of expressions involving listrec 2

### 2.4.3 Binary Trees

Binary trees follow exactly the same pattern as lists.

The formation rule for binary trees is:

$$\frac{A\ \mathsf{type}}{\mathsf{BinTree}(A)\ \mathsf{type}}\ \text{BinTree Form}$$

Rule 2.55: Binary tree formation

A tree is either a leaf or a node with a value and left and right subtrees, so the introduction rules are:

$$\frac{A\ \mathsf{type}}{\mathsf{leaf} : \mathsf{BinTree}(A)}\ \text{BinTree Intro}$$

Rule 2.56: Binary tree introduction 1

and:

$$\frac{a : A \qquad \mathsf{left} : \mathsf{BinTree}(A) \qquad \mathsf{right} : \mathsf{BinTree}(A)}{\mathsf{node}(a, \mathsf{left}, \mathsf{right}) : \mathsf{BinTree}(A)}\ \text{BinTree Intro}$$

Rule 2.57: Binary tree introduction 2

The computation rules for treerec are:

$$\frac{t \longrightarrow \mathsf{leaf} \qquad d \longrightarrow d'}{\mathsf{treerec}(d, e, t) \longrightarrow d'}\ \mathsf{treerec}\,\text{Comp}$$

Rule 2.58: treerec computation 1

$$\dfrac{t \longrightarrow \mathsf{node}(v, l, r) \qquad e(v, l, r, \mathsf{treerec}(d, e, l), \mathsf{treerec}(d, e, r)) \longrightarrow e'}{\mathsf{treerec}(d, e, t) \longrightarrow e'}\ \mathsf{treerec}\ \text{Comp}$$

Rule 2.59: treerec computation 2

The usual elimination rule for binary trees is:

$$\dfrac{t : \mathsf{BinTree}(A) \qquad \begin{array}{c}[w : \mathsf{BinTree}(A)] \\ \vdots \\ C(w)\ \mathsf{type}\end{array} \qquad d : C(\mathsf{leaf}) \qquad \begin{array}{c}\left[\begin{array}{c} y : A \\ l : \mathsf{BinTree}(A) \\ r : \mathsf{BinTree}(A) \\ u : C(ll) \\ v : C(rr) \end{array}\right] \\ \vdots \\ e(y, l, r, u, v) : C(\mathsf{node}(y, l, r))\end{array}}{\mathsf{treerec}(d, e, t) : C(t)}$$

Rule 2.60: Binary tree elimination

Since the rules for treerec equality follow exactly the same pattern as those for listrec equality we do not present them.

## 2.5 Well-orderings

If we think of well-orderings as well-founded trees we see that we must explain how trees may be formed, and for each way of forming a tree what the parts of the tree are. Thus we need to supply a *constructor set* $A$ and a *selector family* $B$, where $B$ is a family of types over $A$. Hence, the formation rule for the well-order types is:

$$\frac{A\ \mathsf{type} \qquad \begin{array}{c}[x:A]\\ \vdots \\ B(x)\ \mathsf{type}\end{array}}{W(A,B)\ \mathsf{type}}\ W\ \text{form}$$

Rule 2.61: $W$ formation

If we think of values in $W(A,B)$ as well-founded trees, then we see that such a tree is described by giving an object $a$ of type $A$ and a function from $B(a)$ to $W(A,B)$.

So, the introduction rule for the well-order types is:

$$\frac{a : A \qquad \begin{array}{c}[x:B(a)]\\ \vdots \\ b(x) : W(A,B)\end{array}}{\mathsf{sup}(a,b) : W(A,B)}\ W\ \text{Intro}$$

Rule 2.62: $W$ introduction

The recursion operator for well-orderings is usually called wrec, and the rules for its behaviour are:

$$\frac{a \longrightarrow \mathsf{sup}(d,e) \qquad b(d,e,(x)\mathsf{wrec}(e(x),b)) \longrightarrow b'}{\mathsf{wrec}(a,b) \longrightarrow b'}\ \mathsf{wrec}\ \text{Comp}$$

Rule 2.63: wrec computation

The elimination rule for well-orderings is:

$$\dfrac{a : W(A,B) \qquad \begin{matrix}[v : W(A,B)]\\ \vdots \\ C(v)\ \mathsf{type}\end{matrix} \qquad \begin{matrix}\left[\begin{matrix} y : A\\ z(x) : W(A,B)[x : B(y)]\\ u(w) : C(z(w))[w : B(y)]\end{matrix}\right]\\ \vdots\\ b(y,z,u) : C(\mathsf{sup}(y,z))\end{matrix}}{\mathsf{wrec}(a,b) : C(a)}\ W\ \text{Elim}$$

Rule 2.64: *W* elimination

Crudely, in intuitionistic mathematics a *spread* is a tree in which each node has at least one immediate sub-node. Consequently, in a spread every path is infinite. A *fan* is a tree where each node has only finitely many immediate sub-nodes. Again crudely, a fan is *barred* if every path in the tree is finite. The $W$ elimination rule is equivalent to bar induction. For a less crude discussion of the mathematics of spreads see Chapter 3 of [25], §4.8 of [114], and [14]. The relationship between bar induction and $W$ elimination is discussed in more detail in§7.6 of Chapter 11 of [115].

### 2.5.1 Using well-orderings to represent binary trees

We can use $W$ types to represent the inductively defined types we presented in §2.4. We use binary trees (§2.4.3) as an example. In order to represent such trees as well-orders we must give the constructor set $A$ and a selector family $B$. We use the same sleight-of-hand here that we used in §2.3.1, and which we explain in §2.7

For trees of values of type $D$ we begin by making this definition:

$$A =_{\mathsf{def}} (D)(\{\mathsf{leaf}\} + D)$$

We proceed by defining:

$$\begin{aligned} &B(\mathsf{inl}(\mathsf{leaf})) =_{\mathsf{def}} \{\}\\ &B(\mathsf{inr}(d)) =_{\mathsf{def}} \{\mathsf{left}, \mathsf{right}\}\end{aligned}$$

In other words:

$$B =_{\mathsf{def}} \mathsf{when}((y)\{\}, (y)\{\mathsf{left}, \mathsf{right}\})$$

We can then define:

$$\mathsf{Tree}'(D) =_{\mathsf{def}} W(A(D), B)$$

If we make the appropriate substitutions in the $W$ Introduction rule (Rule 2.62), and do a little work, we get the following rules:

$$\frac{D\ \mathsf{type}}{\mathsf{sup}(\mathsf{inl}(\mathsf{leaf}), (i)i) : \mathsf{Tree}'(D)}\ \mathsf{Tree}'\ \text{Intro}$$

Rule 2.65: Tree′ introduction 1

and:

$$\frac{d : D \qquad ll : \mathsf{Tree}'(D) \qquad rr : \mathsf{Tree}'(D)}{\mathsf{sup}(\mathsf{inr}(d), \mathsf{case}_{\{\mathsf{left},\mathsf{right}\}}(ll, rr)) : \mathsf{Tree}'(D)}\ \mathsf{Tree}'\ \text{Intro}$$

Rule 2.66: Tree′ introduction 2

And hence we make the definitions:

$$\mathsf{leaf}' =_{\mathsf{def}} \mathsf{sup}(\mathsf{inl}(\mathsf{leaf}), (i)i)$$

and:

$$\mathsf{node}'(d, ll, rr) =_{\mathsf{def}} \mathsf{sup}(\mathsf{inr}(d), \mathsf{case}_{\{\mathsf{left},\mathsf{right}\}}(ll, rr))$$

We can now see that we have recovered the rules for binary tree introduction by comparing Rule 2.56 with Rule 2.65, and Rule 2.57 with Rule 2.66.

We can now plug the definitions that we have made into the $W$ elimination rule (Rule 2.64). After some work we obtain Rule 2.91 on page 79.

Finally we can define treerec′$(d, e, t)$ as:

```
wrec(t,
     (x, y, z)when((q)d,
                   (q)e(q, y(left), y(right), z(left), z(right)),
                   x))
```

Thus we have shown how we can use a $W$ type to represent binary trees.

## 2.6 Equality

We have seen four different notions of equality in M-LTT:

- $=_{\text{def}}$ definitional equality, in §1.1;
- $\equiv$ syntactic equality, in §1.1;
- the judgement of two types being equal types, in §1.2.3;
- the judgement of two objects being equal objects of a type, in §1.2.4.

We now add two further notions of equality, both at the level of types. These two notions are an intensional and a extensional equality.

### 2.6.1 Intensional equality: ID

The intensional equality is the *judgement* of two objects being equal objects of a type 'dropped' to the level of *types*.

The ID formation rule is:

$$\frac{A\ \text{type} \qquad a : A \qquad b : A}{\text{ID}(a, b, A)\ \text{type}}$$

Rule 2.67: ID formation

The ID introduction rule is:

$$\frac{a : A}{\mathsf{id}(a) : \mathsf{ID}(a,a,A)}\ \mathsf{ID}\ \text{intro}$$

Rule 2.68: ID introduction

Rather more useful than this rule, which merely tells us that $a$ is equal to itself, is the following rule:

$$\frac{a = b : A}{\mathsf{id}(a) : \mathsf{ID}(a,b,A)}\ \mathsf{ID}\ \text{intro}'$$

Rule 2.69: Alternative ID introduction

The non-canonical constant for ID is called idpeel, and has the following computation rule:

$$\frac{a \longrightarrow \mathsf{id}(c) \qquad b(c) \longrightarrow b'}{\mathsf{idpeel}(a,b) \longrightarrow b'}\ \mathsf{idpeel}\ \text{Comp}$$

Rule 2.70: idpeel computation

And the elimination rule is:

$$\frac{a : A \quad b : A \quad c : \mathsf{ID}(a,b,A) \quad \begin{array}{c}\left[\begin{array}{c} x : A \\ y : A \\ z : \mathsf{ID}(x,y,A)\end{array}\right] \\ \vdots \\ C(x,y,z)\ \mathsf{type}\end{array} \quad \begin{array}{c}[w : A] \\ \vdots \\ d(w) : C(w,w,\mathsf{id}(w))\end{array}}{\mathsf{idpeel}(c,d) : C(a,b,c)}$$

Rule 2.71: ID elimination

### 2.6.2 Extensional equality: EQ

The extensional equality is also the *judgement* of two objects being equal objects of a type 'dropped' to the level of *types*. The formation rule for the extensional equality is the same as that for the intensional equality:

$$\frac{A \text{ type} \qquad a : A \qquad b : A}{\mathsf{EQ}(a, b, A) \text{ type}}$$

Rule 2.72: EQ formation

The introduction rule is different, as there is at most one value in an EQ type:

$$\frac{a = b : A}{\mathsf{eq} : \mathsf{EQ}(a, b, A)}\ \mathsf{EQ}\text{ intro}$$

Rule 2.73: EQ introduction

We have lost some information here compared to Rule 2.69.

Working by analogy with Rule 2.70, we call the non-canonical constant to be eqpeel, and we construct a computation rule for eqpeel:

$$\frac{a \longrightarrow \mathsf{eq} \qquad b \longrightarrow b'}{\mathsf{eqpeel}(a, b) \longrightarrow b'}\ \mathsf{eqpeel}\text{ Comp}$$

Rule 2.74: eqpeel computation

Thus eqpeel(eq, $f$) has the same value as $f$. If we use this observation when constructing the elimination rule for EQ we get:

$$\dfrac{a : A \quad b : A \quad c : \mathsf{EQ}(a,b,A) \quad \begin{array}{c}\left[\begin{array}{c} x : A \\ y : A \\ z : \mathsf{EQ}(x,y,A)\end{array}\right] \\ \vdots \\ C(x,y,z)\ \mathsf{type}\end{array} \quad \begin{array}{c}[w : A] \\ \vdots \\ d(w) : C(w,w,\mathsf{eq})\end{array}}{d(a) : C(a,b,c)}$$

Rule 2.75: EQ elimination 1

We also have an elimination rule for EQ which lets us move up to the judgement of equality:

$$\dfrac{\mathsf{eq} : \mathsf{EQ}(a,b,A)}{a = b : A}\ \mathsf{EQ\ elim}$$

Rule 2.76: EQ elimination 2

## 2.7 Universes

Suppose we wish to discuss *all* types. Is there a way in which we can do this? One obvious approach is to have a type of types, that is a judgement of the form:

$$\dfrac{}{\mathsf{TYPE} : \mathsf{TYPE}}$$

Rule 2.77: TYPE as a type

This was the approach proposed in the first version of Martin-Löf's theory in 1971 [74]. Alas, adding this judgement renders the theory absurd, as shown by Girard's paradox [44], itself a variant of the paradox of Burali-Forti. The key failure is treating 'the type of all types' as simply a type, and the solution is to use a ramified theory, with a hierarchy of 'universes'.

Informally the values in the universe correspond to types themselves. [77] presents two different approaches to formally defining universes:

- Tarskian (due to the resemblance with Tarski's notion of truth [108]);
- Russellian (after [100]).

In the Tarskian approach the values in $U_0$ are the *names* of the types which can be formed without involving quantification over types themselves. In the Russellian approach the values in the first universe $U_0$ are the *types* which can be formed without involving quantification over types themselves.

## 2.7.1 Tarskian Universes

For each of the type forming operations we have a corresponding name, e.g. $\mathsf{List}^\wedge$, $\mathsf{Nat}^\wedge$, $\Pi^\wedge$, $\Sigma^\wedge$, $+^\wedge$. Thus we have a way to encode the types. We also have a way to decode, or unquote, the type names. $^\vee$ is a family of types over the names associated with the type forming operations.[6]

The rule for the formation of the first universe is:

$$\frac{}{U_0\ \mathsf{type}}$$

Rule 2.78: Tarskian $U_0$ formation

We have introduction rules for $U_0$ like:

$$\frac{}{\mathsf{Nat}^\wedge : U_0}$$

Rule 2.79: Tarskian $U_0$ introduction 1

and:

[6] [88] uses $\widehat{\ }$ and $\mathsf{Set}$ for $^\wedge$ and $^\vee$, respectively. [77] uses $T$ for $^\wedge$, and $^\vee$ is done implicitly.

$$\frac{A : U_0}{\mathsf{List}^\wedge(A) : U_0}$$

Rule 2.80: Tarskian $U_0$ introduction 2

which reflect the *formation* rules of the types themselves. Compare Rule 2.79 with Rule 2.39, and Rule 2.80 with Rule 2.47.

The formation rule for $^\vee$ is:

$$\frac{A : U_0}{A^\vee \ \mathsf{type}}$$

Rule 2.81: $^\vee$ formation

We have rules which reflect the need for the unquote of a name to be the type we named, such as:

$$\frac{}{\mathsf{Nat}^{\wedge\vee} = \mathsf{Nat}}$$

Rule 2.82: Unquoting $\mathsf{Nat}^\wedge$

and:

$$\frac{A : U_0}{(\mathsf{List}^\wedge(A))^\vee = \mathsf{List}(A^\vee)}$$

Rule 2.83: Unquoting $\mathsf{List}^\wedge$

### Universe elimination

Given a fixed collection of type forming operations we can define an operator urec and an elimination rule for $U_0$. We have stressed earlier that M-LTT is extensible, in the sense that we are free to add new type forming operations. Since we cannot lay down what all the possible type forming operations are we cannot provide an 'absolute' universe elimination rule, only one which deals with the universe which we may have constructed so far. This process may be further complicated by the way we have described our type forming operations. For example, we have described how to form enumerated types (in §2.2) by presenting type forming operations which are *informally* indexed (Rule 2.22). In order to define a universe elimination rule we must formalise the indexing operation, as is done in [88] or [76].

In order to illustrate the universe elimination rule we will suppose that we have a universe where we have only defined Nat and Π.

The computation rules for urec are:

$$\frac{t \longrightarrow \mathsf{Nat}^{\wedge} \qquad a \longrightarrow a'}{\mathsf{urec}(a, b, t) \longrightarrow a'}$$

Rule 2.84: urec computation 1

and

$$\frac{t \longrightarrow \Pi^{\wedge}(A, B) \qquad b(A, B, \mathsf{urec}(a, b, A), (x)(\mathsf{urec}(a, b, B(x)))) \longrightarrow b'}{\mathsf{urec}(a, b, t) \longrightarrow b'}$$

Rule 2.85: urec computation 2

Rule 2.86 on the next page is the $U_0$ elimination rule.

Writing out the urec computation and $U_0$ elimination rules for universes where more types have been defined is merely a case of extending these rules *mutatis mutandis*.

The types in $U_0$ are called *small types*.

We can now discuss the sleight of hand that we used when defining $+_{df}$ using $\{0, 1\}$ and $\Sigma$ in §2.3.1. Recall that we used the $\{0, 1\}$ elimination

$$\dfrac{c : U_0 \qquad \begin{array}{c}[z : U_0]\\ \vdots \\ C(z)\ \mathsf{type}\end{array} \qquad a : C(\mathsf{Nat}^\wedge) \qquad \begin{array}{c}\left[\begin{array}{c} x : U_0 \\ y(w) : U_0[w : x^\vee] \\ u : C(x) \\ v(t) : C(y(t))[t : x^\vee]\end{array}\right]\\ \vdots \\ b(x, y, u, v) : C(\Pi^\wedge(x, y))\end{array}}{\mathsf{urec}(a, b, c) : C(c)}\ U_0\ \text{Elim}$$

Rule 2.86: Tarskian $U_0$ elimination

rule with a judgement of the form $T$ type. Now that we have introduced universes we can justify this judgement:

$$\dfrac{\dfrac{}{\{0,1\}\ \mathsf{type}}\ \{0,1\}\ \text{Form} \qquad \dfrac{\dfrac{[x : \{0,1\}] \quad A^\wedge : U_0 \quad B^\wedge : U_0}{\mathsf{case}_{\{0,1\}}(A^\wedge, B^\wedge, x) : U_0}\ \{0,1\}\ \text{Elim}}{\mathsf{case}_{\{0,1\}}(A^\wedge, B^\wedge, x)^\vee\ \mathsf{type}}\ {}^\vee\ \text{Form}}{\Sigma(\{0,1\}, (x)\mathsf{case}_{\{0,1\}}(A^\wedge, B^\wedge, x)^\vee)\ \mathsf{type}}\ \Sigma\ \text{Form}$$

Rule 2.87: Derivation of $+_{df}$ formation using universes

So, if $A$ and $B$ are small types then we can form $A +_{df} B$. We can perform a similar analysis on the $+_{df}$ introduction rule that we derived (Rule 2.34), and the rules that we gave for the definitions of binary trees using $W$ types in §2.5.1 also require us to use universes.

### 2.7.2 Hierarchies of Universes

We can extend the process of universe construction to construct higher universes for ourselves: $U_1, U_2 \ldots$ Membership of universes is not transitive. For example, just as it does not make sense to go from zero : Nat and $\mathsf{Nat}^\wedge : U_0$ to zero : $U_0$, so it does not make sense to write Nat : $U_1$. As a more concrete example consider the administrative structure of world soccer: Lorenzo Amoruso is a member of Rangers Football Club, Rangers Football Club is one of the member clubs of the Scottish Football Association (SFA), and the SFA is a member of the International Federation of Football Associations (FIFA). Lorenzo Amoruso, is not however, a member of the SFA, nor is Rangers is a member of FIFA. This analogy can also help clarify the notion of different universes at the same level which we had when dealing with universe elimination. The SFA is a member of the European Union of Football Associations (UEFA). Thus FIFA and UEFA appear as different universes at the same level.

### 2.7.3 Russellian Universes

Defining Russellian universes is syntactically simpler than defining Tarskian universes. We dispense with the naming operation and treat the types themselves as members of the universe. The $U_0$ formation rule (Rule 2.78) remains the same, but the $U_0$ introduction rules now look like:

$$\frac{}{\mathsf{Nat} : U_0}$$

Rule 2.88: Russellian $U_0$ introduction 1

and:

$$\frac{A : U_0}{\mathsf{List}(A) : U_0}$$

Rule 2.89: Russellian $U_0$ introduction 2

Rule 2.81 now takes the form:

$$\frac{A : U_0}{A \ \mathsf{type}}$$

Rule 2.90: Inhabitants of the Russellian $U_0$ are types

The Russellian style may be criticised as rather cavalier, and the we should *not* confuse things with their names. However, it is more convenient to work with.

## 2.8 Using Martin-Löf's Type Theory in practice

As we have stated before M-LTT provides us with an integrated framework in which we can specify and derive programs. Because of the extra information about program correctness it is reasonable to expect that writing programs in M-LTT is harder than writing unspecified programs in some other language. However because of the integrated nature of M-LTT writing *proven correct* programs in M-LTT is (in our opinion) easier than in many other theories. Programming in M-LTT can involve a lot of relatively low-level activity, for example, we are often asked to show that some type is well-formed. We also often need to keep track of assumptions, and perform substitutions into rule schemas. These tasks can divert our focus from the real task in hand: finding an algorithm. One obvious solution to this is to provide some automated support. There are computer-based proof-assistants, such as [17, 52, 58, 72, 91] to help us to use M-LTT (or closely related theories) for programming.

We have already pointed out another problem with the use of M-LTT for programming: the programs that we derive, although correct, are often very inefficient.

There are also programs which M-LTT is not well suited to expressing, interactive systems, for example.

In the next chapter we introduce continuations, a programming technique which will allow us to address these last two issues.

## 2.9 Chapter Summary

In this chapter we have:

- defined and described a collection of useful types, including those types which allow us to express logic, and those types which we can use to represent data types common in programming;
- given examples of how we can define within M-LTT some of the types which we typically take as primitive;
- discussed equality and universes;
- explained some of the application areas of M-LTT.

$$\frac{a : \mathsf{Tree}'(D) \qquad \begin{matrix}[v : \mathsf{Tree}'(D)]\\ \vdots \\ C(v)\,\mathsf{type}\end{matrix} \qquad f(\mathsf{leaf},(i)i,u) : C(\mathsf{leaf}') \qquad \begin{matrix}\begin{bmatrix} d : D \\ ll : \mathsf{Tree}'(D) \\ rr : \mathsf{Tree}'(D) \\ v : C(ll) \\ t : C(rr) \end{bmatrix}\\ \vdots \\ g(d, \mathsf{case}_{\{\mathsf{left},\mathsf{right}\}}(ll, rr), \mathsf{case}_{\{\mathsf{left},\mathsf{right}\}}(v, t)) : C(\mathsf{node}'(d, ll, rr))\end{matrix}}{\mathsf{wrec}(a, \mathsf{when}(f, g)) : C(a)}$$

Rule 2.91: Tree′ elimination

# Part II

# Continuations

# 3. Tail-recursion, and continuations

In this chapter we will introduce tail recursion and discuss continuations.

Continuations have been put to a number of uses. For example:

- continuations have applications in compiling;
- continuations can be used to extract constructive content from classical proofs;
- continuations can be used to express control in functional programming, and to deal with other 'stateful' computations;
- continuations can allow us to discuss computations which we expect to continue more-or-less indefinitely, a facility we require of an operating system;
- continuations have applications in denotational semantics.

In Chapter 4 we point to an analogy between continuation passing and type lifting in categorial grammars.

We give a very gentle introduce tail-recursion and continuations in §3.1, and §3.2 we present continuation-passing variants of some familiar functions, in order to get a feel for programming with continuations. In §3.3 we outline some of the uses of continuations.

## 3.1 Introducing tail-recursion and continuations

A computation rule involves a *tail* call if it is of the form:

$$\frac{x_1 \longrightarrow x'_1 \qquad \psi(x'_1, \ldots) \longrightarrow \alpha}{\phi(x_1, \ldots) \longrightarrow \alpha}$$

Rule 3.1: A tail call

This can be read roughly as: 'To evaluate the function $\phi$, we must evaluate the function $\psi$.' Alas the crudity of this reading leads to two different definitions of tail recursion:

- the first is that a function is tail-recursive if *recursive* calls are tail calls (for example, §10.2 of [71]);
- the second is that a function is tail-recursive if *all* calls are tail calls (for example, §6.8 of [110]).

As a simple example we look at 3 versions of a function to compute the length of a list. Our examples are expressed in the programming language Haskell [19, 93, 111].

We will call a function which is not tail-recursive a *direct* function. The function `len` in Program 3.1 is a direct function to compute the length of a list.

```
len :: [a] -> Int


len [] = 0
len h:t = 1 + len t
```

Program 3.1: The length of a list 1

The function `len1` in Program 3.2 is tail-recursive by the first definition, but not the second. A call to `len1 l 0` will compute the length of the list `l`.

```
len1 :: [a] -> Int -> Int

len1 [] n = n
len1 (h:t) n = len1 t (n + 1)
```

Program 3.2: The length of a list 2

The function `len2` in Program 3.3 is a variant of this function which is tail-recursive by the second definition too. A call to `len2 l f` will compute the value of the function `f` applied to the length of the list `l`.

```
len2 :: [a] -> (Int -> b) -> b

len2 [] k = k 0
len2 (h:t) k = len2 t (\n -> k(n + 1))
```

Program 3.3: The length of a list 3

The auxiliary argument `k` to `len2` in program 3.3 is called a *tail function* or *continuation*. A function like `len2` is said to written in *continuation-passing style*. It is our intention to exploit continuations in the framework of M-LTT. Before we can do this we must illustrate some of the uses of continuations.

Informally, a continuation is a function which tell us 'what to do next' [3]. We can think of the continuation as embodying the 'future' of the computation. It is natural to think of state in dynamic terms, so this intuition helps us to see how continuations give us one way to handle stateful computation in a functional setting.

First, however, we build up some familiarity with the use of continuations.

## 3.2 Some simple functions

In this section we shall look at some more simple Haskell functions and present tail-recursive, continuation-passing versions of them.

We begin by looking at a very simple example, the factorial function:

```
fac :: Int -> Int

fac 0 = 1
fac n = n * fac (n-1)
```

Program 3.4: The factorial function

A CPS version of factorial will take a tail function as an auxiliary argument. The tail function expresses how to continue, so the type of our new function will be `Int -> (Int -> a) -> a`. We shall call our new function `cpsfac`:

```
cpsfac :: Int -> (Int -> a) -> a

cpsfac 0 k = k 1
cpsfac n k = cpsfac (n - 1) (\x -> k(n * x))
```

Program 3.5: A CPS factorial

A call to `cpsfac n f` will compute the value of the function `f` applied to the value of $n!$. Notice that in `fac` in Program 3.4 the multiplication is outside the recursion, whereas in `cpsfac` in Program 3.5 the multiplication is inside the recursion. This is the typical pattern that we see when we write a CPS function.

Suppose we call `cpsfac 3 k` for some arbitrary `k`. Evaluation will be as follows (we name some of the expressions involved to aid readability):

```
   cpsfac 3 k
=> cpsfac 2 (\x -> k(3 * x))
=> cpsfac 1 (\y -> k'(2 * y))    -- where k' is \x -> k(3 * x)
=> cpsfac 0 (\z -> k''(1 * z)) -- where k'' is \y -> k'(2 * y)
=> k''' 1                      -- where k''' is \z -> k''(1 * z)
=> k'' 1
=> k' 2
   k 6
```

Figure 3.1: Evaluating `cpsfac 3 k`

### 3.2.1 A CPS version of Fibonacci's function

As a second example we look at Fibonacci's function for modelling the growth of rabbit populations.[1] The direct version of Fibonacci's function that we first write in Haskell looks like:

```
fib :: Int -> Int

fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Program 3.6: Fibonacci's function in Haskell

Notice that this function is not written in primitive recursive form. Since we are aware that we shall be interested in functions written in primitive recursive form in M-LTT we look at a primitive recursive version of Fibonacci's function. We define an auxiliary function:

[1] The oldest known description is in [68]. [116] gives many interesting properties of this function.

```
fibs :: Int -> (Int, Int)

fibs 0 = (1, 1)
fibs n = (snd(fibs (n - 1)),
          snd(fibs (n - 1)) + fst(fibs (n - 1)))
```

Program 3.7: A structurally recursive version of Fibonacci's function

A call to `fst(fibs n)` will compute `fib n`.

A neater definition uses a `let`:

```
fibs :: Int -> (Int, Int)

fibs 0 = (1, 1)
fibs n = let (lo, hi) = fibs (n-1)
         in (hi, hi + lo)
```

Program 3.8: A neater version of Fibonacci's function

These two functions are essentially the same: the `let` just aids readability.[2] CPS-converting these functions is straightforward: again we supply an auxiliary argument and again the order of the operations on the right gets inverted:

```
cpsfibs :: Int -> ((Int, Int) -> a) -> a

cpsfibs 0 k = k (1, 1)
cpsfibs n k = cpsfibs (n - 1) (\(lo, hi) -> k(hi, hi + lo))
```

Program 3.9: A CPS version of Fibonacci's function

Evaluation of `cpsfibs 3 k` for some arbitrary `k` will be as follows (allowing for the simplification of arithmetic expressions):

[2] It also allows the compiler to make a major optimisation!

```
   cpsfibs 3 k
=> cpsfibs 2 (\(a, b) -> k(b, b + a))
=> cpsfibs 1 (\(p, q) -> k'(q, q + p))
                        -- where k' is \(a, b) -> k(b, b + a)
=> cpsfibs 0 (\(x, y) -> k''(y, y + x))
                        -- where k'' is \(p, q) -> k'(q, q + p)
=> k'''(1, 1) -- where k''' is \(x, y) -> k''(y, y + x)
=> k''(1, 2)
=> k'(2, 3)
   k (3, 5)
```

Figure 3.2: Evaluating `cpsfibs 3 k`

We are now at a point where we can make some observations about the CPS versions of the functions we have produced. The first observation is that the time complexity of `cpsfibs` is much better than that of `fib`. The second observation is that there is a possibility to optimise the evaluation of `cpsfib`. Whatever structure we build to represent the call to `cpsfibs 3 k` can simply be replaced by the structure that we build to represent the call to `cpsfibs 2 (\(a, b) -> k(b, b + a))`. In fact the algorithm that we have written is very similar to the C [62] function in Program 3.11.

The function `repeat` in Program 3.10 implements a looping construct in Haskell.

```
repeat :: Int -> (a -> b) -> a -> b

repeat 0 f a = f a
repeat n f a = repeat (n-1) (\x -> f x) a
```

Program 3.10: Looping

So we see that the use of tail-recursive functions allows us to write algorithms which *behave* like imperative loops, and which a compiler can treat

```
int fib ( int n ){
  int lo, hi;
  int i;

  lo = 0;
  hi = 1;

  for( i = 0 ;  i <= n ; i++ ){
  hi = hi + lo;
  lo = hi - lo;
  }
  return lo;
}
```

Program 3.11: Computing Fibonacci's function in C

in the same way as it treats an imperative loop. This is a crucial point: using continuations lets us write imperative algorithms in a functional language. Thus we see that we can use continuations to allow us to handle computations which involve state. The use of continuations which we discuss later in this chapter are essentially the application of this observation to various specific problems.

## 3.3 Uses of continuations

In the previous section we presented some very simple functions and CPS versions of them. In this section we will describe some of the ways in which continuations can be exploited. Some of the uses of continuations, particularly in compiling require us to be able to convert any function into CPS. CPS conversion was first discussed in [35], and a number of variants have since appeared in the literature. We only discuss here the conversion presented in [94], as we only intend to give a flavour of what is involved. Fuller discussion and comparison of different CPS conversions for $\lambda$-calculi can be found in, for example, [54, 109].

In [94] Plotkin is interested in the relationship between call-by-value and call-by-name. He defines the following CPS-conversion to allow us to map terms from a language with call-by-value to one with call-by-name:

$$\begin{array}{rcl}
\overline{x} & \rightarrow & \lambda k.kx \\
\overline{\lambda x.M} & \rightarrow & \lambda k.k(\lambda x.\overline{M}) \\
\overline{MN} & \rightarrow & \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.mnk))
\end{array}$$

Figure 3.3: Plotkin's CPS-conversion

Where $k$, $m$ and $n$ are chosen to avoid variable capture in the usual way. Plotkin proves some theorems relating to the values of $M$ and $\overline{M}$ when call-by-name and call-by-value evaluation strategies are used.

The effect of Plotkin's CPS conversion on some terms is shown in Figure 3.4. We have named the newly introduced bound variables $k_n$.

$$\begin{array}{rcl}
\overline{x} & \rightarrow & \lambda k_1.k_1x \\
\overline{\lambda k.kx} & \rightarrow & \lambda k_1.k_1(\lambda k.(\lambda k_2.(\lambda k_3.k_3k)(\lambda k_4.(\lambda k_5.k_5x)(\lambda k_6.k_4k_6k_2)))) \\
\overline{\lambda x.x} & \rightarrow & \lambda k_1.k_1(\lambda x.(\lambda k_2.k_2x)) \\
\overline{xy} & \rightarrow & \lambda k_1.(\lambda k_2.k_2x)(\lambda k_3.(\lambda k_4.k_4y)(\lambda k_5.k_3k_5k_1))
\end{array}$$

Figure 3.4: CPS-translation some particular terms

We can make a number of observations:

- the conversion is not idempotent;
- new redexes may be introduced by the conversion;

- the behaviour of the interpreter is reflected in the structure the term produced.

Making the structure of the term reflect its evaluation gives us an insight into why CPS-conversions are of interest in compiling. This helps us to formalise the informal observations that we were able to make in the previous section about the small functions we looked at. As far as compilation is concerned we see:

- CPS-conversion can produce terms which tell us useful things about how they will be evaluated;
- such terms likely to tell us a lot of things which are not really very useful, so they need to be optimised.

There is a large literature on the use of CPS in compilation: [3, 36] provide a good start.

Another direction that we can follow from the use of CPS-conversions leads us to the extraction of constructive content from classical proofs. Notoriously, classical proofs need not contain any constructive content. However, we know that we can use the double-negation transformation to produce a intuitionistic theorem from a classical one. In [39] Friedman introduced a related technique, called A-translation, which allowed him to show that Peano arithmetic is a conservative extension of Heyting arithmetic, for $\Pi_2^0$ sentences. The constructive content of Friedman's proof is that we can convert a classical proof of a $\Pi_2^0$ sentence into a constructive one. The constructed proof can be interpreted as the application of a non-local control operator applied to a CPS-conversion of the classical proof. The control operator allows us to replace the current evaluation context with a different one, just as `goto` allows us to make non-local jumps in imperative programs. There is an extensive literature on control operators: [18, 103] are a beginning. [50, 85] provide much more detail on the extraction of constructive content from classical proofs, and the relation with control operators.

Just as in imperative programming we can make jumps available to the programmer by providing goto we can make the control operator available to the programmer. The is done in Scheme [61] and Standard ML of New Jersey [3] using a callcc (call-with-current-continuation) primitive. [105] describes the representation of jumps with continuations in more detail. Just as goto allows the programmer to invent control structures, so does callcc, with all that this entails. Continuations allow us to implement threads, as discussed in, for example, [3] and [24].

Again, connected with their rôle of representing control in a functional setting continuations have applications in denotational semantics as discussed in, for example, [102].

### 3.3.1 Continuations and I/O

In [7] Barendregt points out that many functional programming languages can be considered 'autistic' in that they lack any ability to communicate with the outside world[3]. This is, as he points out, partly because our thinking is in terms of the evaluation of functions, an activity which naturally concerns itself with *termination*, whereas for communication we need a notion of process, and the evolution of processes is a *continuing* activity. Continuations help us formalise this intuition, as, in a simple model of I/O, we can do three things:

- we can stop;
- we can read a value, and continue by performing some computation with it;
- we can write a value, and continue performing computations.

[3] As Estragon puts it, in Beckett's *Waiting for Godot*: 'Nothing happens, nobody comes, nobody goes, it's awful!'

At this level the world is almost as simple: it is just a pair of lists of natural numbers. We see that CPS functions let us thread a state or world value through our programs.

## 3.4 Chapter Summary

In this chapter we gave a very gentle introduction to programming in a continuation-passing style, presented a CPS-transformation for $\lambda$-terms and mentioned some of the uses to which continuations have been put. One common theme to these uses is the representation of control.

# 4. Type-lifting in categorial grammar

In Chapter 3 we mentioned several well-known applications of continuations. In this chapter we will show a novel application, by showing that we can interpret type-lifting in categorial grammar as CPS-conversion.

Categorial grammars are a family of grammar formalisms which are used in computational linguistics and are of interest to philosophers of language. They have their roots in the work of Adjukiewicz [2], Bar-Hillel [5], and Lambek [67], and ultimately in Frege's context principle ('never to ask for the meaning of a word in isolation, but only in the context of a proposition'. The principle is enunciated in [37], and is discussed in some depth in Chapter 19 of [29].). Formally, categorial grammars can be seen as sub-structural logics [99] which bear a close relation to linear logic [46, 113]. [15, 84] make many of the details of this relationship more explicit. The slogan that is associated with the view of categorial grammar as a logic is 'parsing as deduction' [65].

The essential idea behind categorial grammar is that linguistic expressions come in a variety of syntactic categories, and that the syntactic categories have some close relationship with the semantics of the expressions. The meaning of a complex expression is formed from the meanings of the subexpressions. It is therefore natural to represent the meanings of expressions using $\lambda$-terms. So, once again we see the Curry-Howard analogy, albeit in a slightly different setting.

This chapter proceeds as follows:

- we present a simple categorial grammar;
- we present some simple derivations to illustrate the use of categorial

grammar;

- we explain type raising, and point out an analogy between type-raising and continuation-passing.

## 4.1 A simple categorial grammar

We will present a simple categorial grammar. We will first present a sequent formulation.[1] Sequent systems have one obvious advantage when discussing sub-structural logics as the structural rules are easy to deal with, either by using sequences (lists) of formulas in sequents and stating structural rules explicitly or by dealing with sequences, multi-sets or sets of formulas in sequents, as appropriate for the logic in question. In categorial grammar we deal with sequences of formulas on the left of the sequent arrow ($\Longrightarrow$) and a single formula on the right. We do not have the structural rules of thinning, weakening or exchange. The grammar that we will consider has two directed implications, written $X\backslash Y$ and $Y/X$, where $X$ is the implicans and $Y$ the implicate.[2] Categorial grammar can be presented with more connectives, (see, for example [84]), but the fragment with only \ and / is sufficient for our purposes. The \ and / right rules, annotated with meaning-terms, are:

$$\frac{x : X, \Gamma \Longrightarrow y : Y}{\Gamma \Longrightarrow \lambda x.y : X\backslash Y} \Longrightarrow \backslash$$

Rule 4.1: The $\Longrightarrow \backslash$ sequent rule

[1] Natural deduction systems follow the pattern of Gentzen's **N** systems, sequent calculi Gentzen's **L** systems, from [42]. [107] also carefully explains the distinction.

[2] The use of slashes is a very unfortunate choice of notation. We use the convention of [15]. Unfortunately some authors (for example, [22]) use different conventions. Arrows are an improvement, but we will use arrows for types.

$$\frac{\Gamma, x : X \Longrightarrow y : Y}{\Gamma \Longrightarrow \lambda x.y : Y/X} \Longrightarrow /$$

Rule 4.2: The $\Longrightarrow$ / sequent rule

In these rules $\Gamma$ is a non-empty[3] sequence of formulas, $X$ and $Y$ are formulas and $x$ is chosen in the usual way to avoid variable capture problems.

The \ and / left rules, annotated with meaning-terms, are:

$$\frac{\Delta \Longrightarrow x : X \qquad \Gamma, f(x) : Y, \Theta \Longrightarrow z : Z}{\Gamma, \Delta, f : X\backslash Y, \Theta \Longrightarrow z : Z} \backslash \Longrightarrow$$

Rule 4.3: The \ $\Longrightarrow$ sequent rule

$$\frac{\Delta \Longrightarrow x : X \qquad \Gamma, f(x) : Y, \Theta \Longrightarrow z : Z}{\Gamma, f : Y/X, \Delta, \Theta \Longrightarrow z : Z} / \Longrightarrow$$

Rule 4.4: The / $\Longrightarrow$ sequent rule

In these rules $\Gamma$, $\Delta$ and $\Theta$ are (possibly empty) sequences of formulas and $X$, $Y$ and $Z$ are formulas.

There are two structural rules: a sequent of the form $x : X \Longrightarrow x : X$ is trivial, and there is an (admissible) cut rule:

$$\frac{\Delta \Longrightarrow x : X \qquad \Gamma, x : X, \Theta \Longrightarrow z : Z}{\Gamma, \Delta, \Theta \Longrightarrow z : Z} \text{Cut}$$

Rule 4.5: The cut rule

In this rule $\Gamma$, $\Delta$ and $\Theta$ are (possibly empty) sequences of formulas and $X$ and $Z$ are formulas.

[3] The motivation for this restriction is linguistic rather than logical. One may observe that there are no valid categorical judgements in categorial grammar.

The $\lambda$-terms which we use to represent meaning are terms of the simple typed $\lambda$-calculus. It is useful to remember that the types and categories of the terms are distinct notions. For example if the term $\lambda p.q$ has *type* $P^t \to Q^t$ then it will have *categories* $P^c\backslash Q^c$ and $Q^c/P^c$, where the superscript $^t$ indicates type and $^c$ category. The types involved in categorial grammars may be called *semantic types*, because they are the types of the meaning terms.

We can give a natural deduction presentation of the rules of categorial grammar. As one might expect we must be careful with the structure of proof trees: the *order* in which the premisses to rules occur of premisses to rules is important as is the *number* of occurrences of a formula discharged. Although the natural deduction rules are harder to present it is typically easier to read completed natural deduction derivations. The \ introduction rule is:

$$\dfrac{\begin{array}{c}[x : X] \quad \cdots \\ \vdots \\ y : Y\end{array}}{\lambda x.y : X\backslash Y}\ \backslash\ \text{Introduction}$$

Rule 4.6: \ Introduction

The extra side-conditions on this rule are:

- exactly one occurrence of $X$ is discharged;
- it must be the leftmost assumption on which the proof of $Y$ relies;
- it must not be the only assumption on which the proof of $Y$ relies.

The / introduction rule is:

$$\frac{\begin{array}{cc} \cdots & [x : X] \\ & \vdots \\ & y : Y \end{array}}{\lambda x.y : Y/X} \; / \text{ Introduction}$$

Rule 4.7: / introduction

The extra side-conditions on this rule are:

- exactly one occurrence of $X$ is discharged;
- it must be the rightmost assumption on which the proof of $Y$ relies;
- it must not be the only assumption on which the proof of $Y$ relies.

The \ and / elimination rules are presented after the pattern of *Modus Ponens* (see § 2.3.2 on page 50). The rules are:

$$\frac{x : X \quad f : X\backslash Y}{f(x) : Y} \; \backslash \text{ Elimination}$$

Rule 4.8: \ elimination

and:

$$\frac{f : Y/X \quad x : X}{f(x) : Y} \; / \text{ Elimination}$$

Rule 4.9: / elimination

The order of the premisses *is* important in Rules 4.8 and 4.9. This is one reason why we choose to formulate these rules after the pattern of *Modus Ponens*.

Proofs which have maximal formulas (i.e. formulas which are the conclusion of an introduction rule and the major premiss of an elimination rule) in them can be reduced in the expected way, as shown in Figures 4.1 and 4.2.

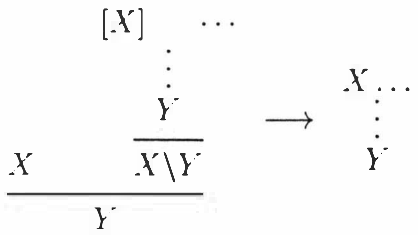


Figure 4.1: Proof normalisation in categorial grammar 1

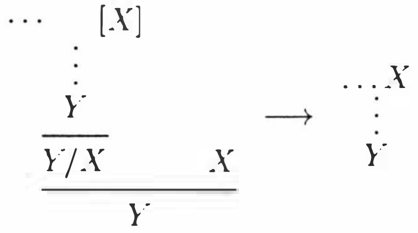


Figure 4.2: Proof normalisation in categorial grammar 2

### 4.1.1 Section summary

We have presented a simple categorial grammar in both sequent and natural deduction format. We presented both formats because the sequent calculus rules make the structural conditions easier to express, but the natural deduction proof trees are easier to read.

## 4.2 Examples of simple derivations

What we have presented so far is, essentially, a logic, which we intend to treat as a grammar. We must provide a lexicon. The lexicon consists of:

- words;
- their semantics;
- their categories; and
- their types.

For example:

| Word | Semantics | Category | Type |
|---|---|---|---|
| eats | $\lambda(x)\mathsf{eats}'(x)$ | $\mathsf{NP\backslash S}$ | $e \to t$ |
| John | $\mathsf{john}'$ | $\mathsf{NP}$ | $e$ |
| loves | $\lambda(x,y)\mathsf{loves}'(y,x)$ | $\mathsf{(NP\backslash S)/NP}$ | $e \to e \to t$ |
| Mary | $\mathsf{mary}'$ | $\mathsf{NP}$ | $e$ |

Figure 4.3: A lexicon

The meanings of the proper nouns *John* and *Mary* are represented by individuals chosen from some suitable type $e$, the type of 'entities', following Montague [81]. The meaning of the verb *loves* is represented by a function of type $e \to e \to t$, where $t$ is, again following Montague, the type of truth values. We parse the sentence 'John loves Mary' by looking in the lexicon for the entries 'John', 'loves' and 'Mary'. Although looking in the lexicon is not, strictly, an inference rule we indicate lexicon lookup in our proofs like this:

$$\frac{\text{Word}}{\mathsf{Semantics}' : \mathsf{Category}}\ \text{Lexicon}$$

Rule 4.10: Lexicon lookup

We can now perform this simple proof.

$$\dfrac{\dfrac{\text{John}}{\mathsf{john}' : \mathsf{NP}}\,\text{Lex.} \qquad \dfrac{\dfrac{\text{loves}}{\lambda(x,y)\mathsf{loves}'(y,x) : (\mathsf{NP}\backslash\mathsf{S})/\mathsf{NP}}\,\text{Lex.} \quad \dfrac{\text{Mary}}{\mathsf{mary}' : \mathsf{NP}}\,\text{Lex.}}{\lambda(y)\mathsf{loves}'(y,\mathsf{mary}') : \mathsf{NP}\backslash\mathsf{S}}\,/\,\text{E.}}{\mathsf{loves}'(\mathsf{john}',\mathsf{mary}') : \mathsf{S}}\,\backslash\,\text{E.}$$

Rule 4.11: Showing 'John loves Mary'

The phrase parsed can be read from the leaves of the tree, and its meaning and category can be read from the root of the tree.

As a second example, suppose we extend the lexicon as follows:

| Word | Semantics | Category | Type |
|---|---|---|---|
| believes | $\lambda(u,v)\mathsf{bel}'(v,u)$ | $(\mathsf{NP}\backslash\mathsf{S})/\mathsf{S}$ | $t \to (e \to t)$ |
| foolishly | $\lambda(y,x)\mathsf{fool}'(y(x))$ | $(\mathsf{NP}\backslash\mathsf{S})\backslash(\mathsf{NP}\backslash\mathsf{S})$ | $(e \to t) \to (e \to t)$ |
| Murray | $\mathsf{mur}'$ | $\mathsf{NP}$ | $e$ |
| resigned | $\lambda(x)\mathsf{res}'(x)$ | $\mathsf{NP}\backslash\mathsf{S}$ | $e \to t$ |
| Souness | $\mathsf{sou}'$ | $\mathsf{NP}$ | $e$ |

Figure 4.4: Extending our lexicon

There are two readings of the sentence:

'Murray believes Souness resigned foolishly',

depending on whether it is the belief or the resignation which is foolish. These readings correspond to two different derivations given in Rule 4.18 on page 106 and Rule 4.19 on page 107. The two proofs differ in their structure, and the corresponding $\lambda$-terms reflect this difference.

### 4.2.1 Section summary

In this section we have presented a small lexicon and given some simple proofs. Using the 'parsing as deduction' analogy we can see:

- searching for a proof is the same as attempting to parse;
- whereas in theorem proving we are often only concerned with finding one proof, in categorial grammar we are concerned with finding all the distinct proofs;
- the structure of the $\lambda$-term that we construct reflects the meaning of the phrase we have parsed.

Clearly we have not introduced nearly enough material to handle anything other than the simplest of sentences. [15, 83] develop the theory further.

In English word order is significant, and all the sentences we have looked at have had a very simple word order. Constructing the meaning of a phrase has only required us to compose the meanings of immediately adjacent components. There are phenomena, even in fixed word-order languages like English, for which this is not the case. On occasion we need to have the ability to manipulate the proof tree, to pass values around, in a way reminiscent of continuation passing.

## 4.3 Type-raising as continuation-passing

Our current interest is in the use of 'type lifting' or 'type raising'. Type raising was discussed in [67] (although not named). [22] defines type raising as:

> 'the process of re-analyzing an argument category as a new functor which takes as its argument the functor that would have applied to it before type raising.'

We were careful before to keep the notions of type and category distinct. An expression of type $\alpha$ will be raised to type $(\alpha \to \beta) \to \beta$. A category $X$ will be raised to $Y/(X\backslash Y)$ or to $(Y/X)\backslash Y$. The possibilities $Y\backslash Y\backslash X$ and $X/Y/Y$ are ruled out by the need to preserve word-order when type-raising is performed. Syntactically, the effect of type-raising is to CPS-convert the term involved.

Type-raising can be expressed with the following sequent rules:

$$\frac{\Gamma \Longrightarrow x : X}{\Gamma \Longrightarrow \lambda(f)(f(x)) : Y/(X\backslash Y)} \uparrow$$

Rule 4.12: Type raising sequent rule 1

and:

$$\frac{\Gamma \Longrightarrow x : X}{\Gamma \Longrightarrow \lambda(f)(f(x)) : (Y/X)\backslash Y} \uparrow$$

Rule 4.13: Type raising sequent rule 2

and the following natural deduction rules:

$$\frac{x : X}{\lambda(f)(f(x)) : Y/(X\backslash Y)} \uparrow$$

Rule 4.14: Type raising natural deduction rule 1

and:

$$\frac{x : X}{\lambda(f)(f(x)) : (Y/X)\backslash Y} \uparrow$$

Rule 4.15: Type raising natural deduction rule 2

The examples given in [22] of the uses of type-raising include:

- obtaining the correct interpretations sentences involving both extensional (e.g. 'John bought and read a book.') and intensional verbs (e.g. 'John wants and needs a haircut.');
- obtaining the correct interpretations of conjunctions of common nouns: the phrase 'fresh and salt water' should be read as 'fresh water and salt water' *not* 'water which is both fresh and salt';
- obtaining the correct reading of sentences involving disjunctions with a wide scope (e.g. 'John would like a coffee or a tea.').

We illustrate type-raising using an example from [82]: correctly parsing 'He or John loves Mary.'

We extend our lexicon as follows:

| Word | Semantics | Category | Type |
|---|---|---|---|
| he | $\lambda f(f(\mathsf{he}'))$ | S/(NP\S) | $(e \to t) \to t$ |
| or | $\lambda(f,g,h)(g(h)\ \mathsf{or}'\ f(h))$ | ((S/(NP\S))<br>\(S/(NP\S)))<br>/(S/(NP\S)) | $((e \to t) \to t) \to$<br>$((e \to t) \to t) \to$<br>$(e \to t) \to t$ |

Figure 4.5: Further extending our lexicon

Notice that 'he' has already been lifted: we need to prohibit the sentence 'Mary loves he' and permit 'He loves Mary.'

With a wide-scope disjunction'He or John loves Mary' means 'He loves Mary, or John loves Mary' so the $\lambda$-term constructed when parsing 'He or John loves Mary' should be the disjunction of those constructed when parsing 'He loves Mary' and 'John loves Mary'. We have already seen a derivation of 'John loves Mary' in Rule 4.11. Now we derive 'He loves Mary':

$$\dfrac{\dfrac{\text{He}}{\lambda(f)(f(\mathsf{he}')) : \mathsf{S}/(\mathsf{NP}\backslash\mathsf{S})} \qquad \dfrac{\dfrac{\text{loves}}{\lambda(x,y)\mathsf{loves}'(y,x) : (\mathsf{NP}\backslash\mathsf{S})/\mathsf{NP}} \quad \dfrac{\text{Mary}}{\mathsf{mary}' : \mathsf{NP}}}{\lambda(y)\mathsf{loves}'(y,\mathsf{mary}') : \mathsf{NP}\backslash\mathsf{S}}\,/\,\mathrm{E}}{\mathsf{loves}'(\mathsf{he}',\mathsf{mary}') : \mathsf{S}}\,/\,\mathrm{E}$$

Rule 4.16: Showing 'He loves Mary'

Without using type-raising any attempt to parse 'He or John loves Mary' will fail: we can only progress by parsing the phrase 'John loves Mary' of category S. At this point we find we are blocked because we cannot apply 'or' to 'John loves Mary' or 'He' to 'or'. The insight is to raise the type of 'John':

$$\dfrac{\dfrac{\text{John}}{\mathsf{john}' : \mathsf{NP}}}{\lambda(q)(q(\mathsf{john}') : \mathsf{S}/(\mathsf{NP}\backslash\mathsf{S})}\uparrow$$

Rule 4.17: Lifting 'John'

The complete derivation is shown in Rule 4.20 on page 108.[4]

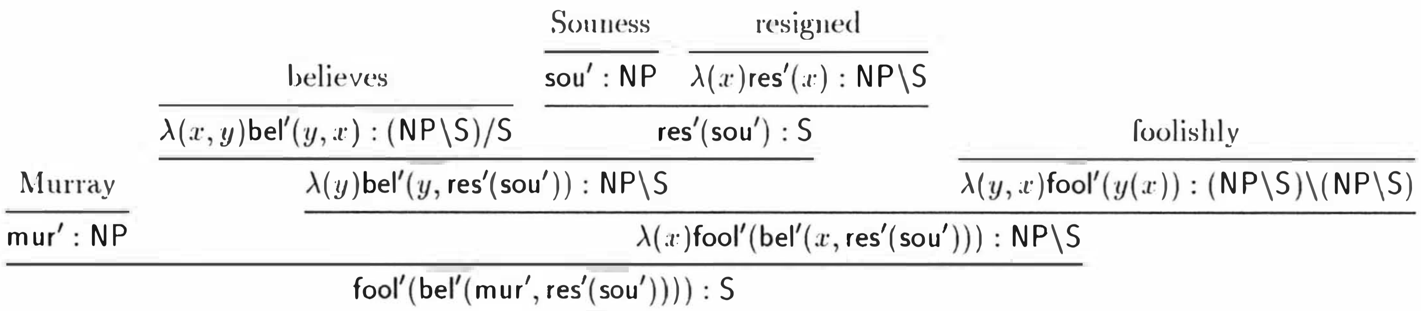
The effect of type raising in this proof is to allow us to pass the semantic term associated with 'loves Mary' around. This has a strong resemblance to the way in which continuation-passing allows us to pass expressions around in the programming setting. We claim that other applications of type-lifting can be viewed analogously.

## 4.4 Chapter Summary

In this chapter we have introduced categorial grammar, and presented some simple derivations. Our intention in doing this was to allow us to

[4] In this proof the category of 'or' has been suppressed, for clarity.

draw an analogy between type-lifting and continuation-passing, thereby presenting a novel application of continuation-passing.


Murray
believes
Souness
resigned
foolishly
mur′ : NP
λ(x, y)bel′(y, x) : (NP\S)/S
sou′ : NP
λ(x)res′(x) : NP\S
res′(sou′) : S
λ(y)bel′(y, res′(sou′)) : NP\S
λ(y, x)fool′(y(x)) : (NP\S)\(NP\S)
λ(x)fool′(bel′(x, res′(sou′))) : NP\S
fool′(bel′(mur′, res′(sou′)))) : S

Rule 4.18: One reading of ‘Murray believes Souness resigned foolishly’

$$\dfrac{\dfrac{\text{Murray}}{\mathsf{mur}' : \mathsf{NP}} \quad \dfrac{\dfrac{\text{believes}}{\lambda(x,y)\mathsf{bel}'(y,x) : (\mathsf{NP}\backslash\mathsf{S})/\mathsf{S}} \quad \dfrac{\dfrac{\text{Souness}}{\mathsf{sou}' : \mathsf{NP}} \quad \dfrac{\dfrac{\text{resigned}}{\lambda(x)\mathsf{res}'(x) : \mathsf{NP}\backslash\mathsf{S}} \quad \dfrac{\text{foolishly}}{\lambda(y,x)\mathsf{fool}'(y(x)) : (\mathsf{NP}\backslash\mathsf{S})\backslash(\mathsf{NP}\backslash\mathsf{S})}}{\lambda(x)\mathsf{fool}'(\mathsf{res}'(x)) : \mathsf{NP}\backslash\mathsf{S}}}{\mathsf{fool}'(\mathsf{res}'(\mathsf{sou}')) : \mathsf{S}}}{\lambda(x)\mathsf{bel}'(x,\mathsf{fool}'(\mathsf{res}'(\mathsf{sou}'))) : \mathsf{NP}\backslash\mathsf{S}}}{\mathsf{bel}'(\mathsf{mur}',\mathsf{fool}'(\mathsf{res}'(\mathsf{sou}'))) : \mathsf{S}}$$

Rule 4.19: The other reading of 'Murray believes Souness resigned foolishly'

$$\dfrac{\dfrac{\dfrac{\text{He}}{\lambda(p)(p(\mathsf{he}')) : \mathsf{S}/(\mathsf{NP}\backslash\mathsf{S})} \qquad \dfrac{\dfrac{\text{or}}{\lambda(f,g,h)(g(h)\ \mathsf{or}'\ f(h)) : (\alpha\backslash\alpha)/\alpha} \qquad \dfrac{\dfrac{\text{John}}{\mathsf{john}' : \mathsf{NP}}}{\lambda(q)(q(\mathsf{john}')) : \mathsf{S}/(\mathsf{NP}\backslash\mathsf{S})}\,\uparrow}{\lambda(g,h)(g(h)\ \mathsf{or}'\ h(\mathsf{john}')) : (\mathsf{S}/(\mathsf{NP}\backslash\mathsf{S}))\backslash(\mathsf{S}/(\mathsf{NP}\backslash\mathsf{S}))}\,/\,\mathrm{E}}{\lambda(h)(h(\mathsf{he}')\ \mathsf{or}'\ h(\mathsf{john}')) : \mathsf{S}/(\mathsf{NP}\backslash\mathsf{S})}\,\backslash\,\mathrm{E} \qquad \dfrac{\dfrac{\text{loves}}{\lambda(x,y)\mathsf{loves}'(y,x) : (\mathsf{NP}\backslash\mathsf{S})/\mathsf{NP}} \qquad \dfrac{\text{Mary}}{\mathsf{mary}' : \mathsf{NP}}}{\lambda(y)\mathsf{loves}'(y,\mathsf{mary}') : \mathsf{NP}\backslash\mathsf{S}}\,/\,\mathrm{E}}{\mathsf{loves}'(\mathsf{he}',\mathsf{mary}')\ \mathsf{or}'\ \mathsf{loves}'(\mathsf{john}',\mathsf{mary}') : \mathsf{S}}$$

Rule 4.20: 'He or John loves Mary'

# Part III

# Adding continuations to Martin-Löf's Type Theory

# 5. CPS non-canonical constants for non-inductive types

In this chapter and in Chapters 6 and 7 we will show how we can program with continuations in M-LTT. Our focus in these chapters is on the left of the :. The novelty is not in the types, but in the ways that we can construct objects.

In this chapter we deal with some types which are *not* defined inductively, and whose non-canonical constants are not, therefore, recursive. In Chapter 6 we will look at inductively defined types, and in Chapter 7 we will look at well-orders. Starting with types which are not defined inductively allows us to introduce the CPS non-canonical constants in a relatively simple setting.

We follow the presentation from the first part of Chapter 2 and look at:

- disjoint union of two types: $+$,
- disjoint union of a family of types: $\Sigma$,
- Cartesian product of a family of types: $\Pi$.

The general pattern for these types is that we define a new non-canonical constant which takes an extra auxiliary argument, the continuation. Because *all* the new non-canonical constant now involve a tail call we name them by suffixing tail to the name of the non-CPS non-canonical constant. We also adopt the convention that the continuation is the final argument.

The informal interpretation of the new CPS non-canonical constants that we will define is that the continuation expresses 'what to do next.' The computation rules for the new non-canonical constants will appear slightly more complicated than those for the non-CPS non-canonical constants, and the elimination rule, which we justify in the same way as the conventional elimination rules are justified, will contain extra premisses to deal with the typing of the extra argument. Because there is no recursion involved the notion of 'what to do next' is particularly simple, and, as a check that we have grasped correctly what we are doing we also present a *defined* version of the CPS non-canonical constant, and show that we obtain the same rules when we use this.

## 5.1 Alternative extensions

When we are adding new material to M-LTT we often have a choice between giving a direct presentation and making a definition, for example:

- in §2.1.3 we defined × using Σ;
- in §2.1.5 we defined → using Π;
- in §2.2.1 we defined Bool using enumerated types;
- in §2.3.1 we showed that we could define $+_{df}$ using Σ and $\{0, 1\}$;
- in §2.5.1 we showed how to represent binary trees using $W$ types;
- in [69] mutually recursive types were presented directly and *via* an encoding using $W$ types.

There is no hard and fast rule which determines in every case whether we should give a direct presentation or make a syntactic definition: it would appear in practice that ×, →, Bool are better treated as defined constants; and +, lists and trees are more conveniently treated directly. It is clear that:

- extensions must not render our theory absurd;
- it must be clear when we make an extension that we have correctly formalised the notion we have in mind.

One argument that can be mustered in favour of using syntactic definitions is that this is more secure: since we are not introducing any really new material into the theory we are not introducing a new way to make absurdities. There is much merit to this argument, but it does depend on the security of our mechanism for making definitions. This is not a trivial point. For instance it is notorious that if we define substitution incorrectly we can capture free variables. This mistake was made by Hilbert when writing with Ackermann and when writing with Bernays [56, 57, 16], and lies behind Jensen's device [90], and the problems associated with dynamic scoping in LISP [73], and the caution which is often required when using macro expansion languages.[1] It is also often the case that it is hard to be sure that a definition has correctly captured the notion we are seeking to formalise. The case of defining binary trees in terms of well-orders (§2.5.1) illustrates this. It is hard to conceive that we are more likely to make a mistake describing the type of binary trees directly than we are describing the type of well-orders and then using these to express trees.

For the types discussed in this chapter the defined versions of the CPS non-canonical constants are probably the most convenient. This is because the definitions are very straightforward. For the inductively-defined types and well-orders the situation is rather different. We do not get such a simple definition of the CPS non-canonical constants, and, as we saw in §3.2 using tail-recursion can allow us to express naturally more efficient algorithms.

[1] Curiously, Jensen's device is described as 'ingenious' in [20], and appears as an advanced technique in [49]. Computer scientists and logicians would appear to regard constructions of the absurd differently.

## 5.2 CPS non-canonical constant for the disjoint union of two types

The conventional non-canonical constant for the + types is when. Rules 2.4 and 2.5 explain how to evaluate expressions formed with when. Supplying when with a continuation means providing an auxiliary function to be applied to the result of evaluating the expression formed by when. In §5.2.1 we treat whentail as a primitive. The elimination rule which we obtain looks unusual, so, in §5.2.2, we treat the CPS non-canonical constant as a defined constant, and derive rules using the defined form.

### 5.2.1 Presenting whentail as a primitive

First we give the computation rules, then we give the elimination rule.

#### Computation rules

The computation rules for whentail$(d, e, f, k)$ are as follows:

$$\frac{f \longrightarrow \mathsf{inl}(l) \qquad d(l) \longrightarrow d' \qquad k \longrightarrow \lambda(b) \qquad b(d') \longrightarrow b'}{\mathsf{whentail}(d, e, f, k) \longrightarrow b'}\ \text{whentail Comp}$$

Rule 5.1: whentail computation 1

and:

$$\frac{f \longrightarrow \mathsf{inr}(r) \qquad e(r) \longrightarrow e' \qquad k \longrightarrow \lambda(b) \qquad b(e') \longrightarrow b'}{\mathsf{whentail}(d, e, f, k) \longrightarrow b'}\ \text{whentail Comp}$$

Rule 5.2: whentail computation 2

These rules tell us what we need to show in order to show that the evaluation of whentail$(d, e, f, k)$ will terminate. There is one subtlety here: we need to know that $k \longrightarrow \lambda(b)$, that is we need to know that $k$ is a function. We might have tried to have a premiss to these rules like: $k \longrightarrow (z)y$. However $(z)y$ is not a *value*, and we require $k$ to have a value so that we can infer a type for the conclusion of the elimination rule that we present.

Thus it is clear that we need to have a grasp of the Π types before we can program with continuations.

### Elimination rule

Now we have the task of constructing a rule which will allow us to make a judgement about the type of an expression formed with whentail. This rule will be a + elimination rule. It will also be a Π elimination rule.

We are trying to type:

$$\mathsf{whentail}(d, e, f, k)$$

We explain the premisses in turn.

First we need to be able to judge:

$$f : A + B$$

The first computation rule tells us that we need to be able to make a judgement about the type of $d(x)$, given some $x$ of type $A$. The second computation rule tells us that we need to be able to make a judgement about the type of $e(y)$, given some $y$ of type $B$. We can apply $b$ to either $d(x)$ or to $e(y)$, so they must have types formed in a uniform fashion. If $C$ is a family of type over $A + B$, then $d(x) : C(\mathsf{inl}(x))$ and $e(y) : C(\mathsf{inr}(y))$ are suitable. So we need the following premisses:

$$\begin{array}{ccc} [z : A + B] & [x : A] & [y : B] \\ \vdots & \vdots & \vdots \\ C(z)\ \mathsf{type} & d(x) : C(\mathsf{inl}(x)) & e(y) : C(\mathsf{inr}(y)) \end{array}$$

Now we consider the type of $k$. Since we apply $b$ to an object with type $C(f)$, where $f$ is of type $A + B$, $k$ must be of a Π type from $C(f)$. The Π introduction rule (Rule 2.14 on page 38) is then:

$$\dfrac{C(f)\ \mathsf{type} \qquad \begin{array}{c}[v : C(f)]\\ \vdots \\ D(v)\ \mathsf{type}\end{array}}{\Pi(C(f), D)\ \mathsf{type}}$$

Since we already have $f : A + B$ and that $C$ is a family of types over $A + B$ the premiss $C(f)$ type adds nothing. So we are left with the judgement that $D$ is a family of types over $C(f)$ as a premiss to the rule.

Thus we also have this premiss:

$$k : \Pi(C(f), D)$$

Now we must consider typing $b'$. We need now to look at the rules for equality of expression formed with when (Rules 2.7 and 2.7 on page 35). Considering these two cases shows us that $b'$ has type $D(\mathsf{when}(d, e, f))$, which is consequently the type of $\mathsf{whentail}(d, e, f, k)$. That the type of $\mathsf{whentail}(d, e, f, k)$ may, in general, depend on $\mathsf{when}(d, e, f)$ seems surprising at first, but in §5.2.2 we will derive this rule by using a defined form of whentail.

Rule 5.11 on page 124 is the new rule which we have constructed.

The main differences between Rule 5.11 and the usual + elimination rule (Rule 2.6) are:

- there is a premiss which relates to the type of the continuation;
- the conclusion is a judgement about the type of an expression formed by whentail, rather than when;
- the type of the conclusion may, in general, depend on the value of an expression formed with when.

These differences are typical of the CPS elimination rules that we will see later on.

### 5.2.2 Presenting $\mathsf{whentail}_{df}$ as a defined constant

Since the computation rules (Rules 5.1 and 5.2), and the elimination rule (Rule 5.11) look rather odd we might seek to define a CPS version of when in terms of apply and when, like this:

$$\begin{aligned}\mathsf{whentail}_{df}(d, e, f, k) &=_{\mathrm{def}} \mathsf{apply}(k, \mathsf{when}(d, e, f))\\ &=_{\mathrm{def}} \mathsf{funsplit}((y)(y(\mathsf{when}(d, e, f))), k)\end{aligned}$$

It should be clear that this definition captures our informal notion that the continuation function is a function to apply after we have evaluated $\mathsf{when}(d, e, f)$. We use this definition to re-derive the rules we derived previously.

#### Computation rules

The computation rules for $\mathsf{whentail}_{df}$ can be obtained by considering:

- the definition of $\mathsf{whentail}_{df}(d, e, f, k)$;
- the definition of apply;
- the funsplit computation rule (Rule 2.15);
- the two when computation rules (Rules 2.4 and 2.5).

What we are doing is very similar to our derivation of Rule 2.18, the apply computation rule.

Initially we obtain:

$$\cfrac{\cfrac{\cfrac{k \longrightarrow \lambda(b) \qquad \cfrac{b(\mathsf{when}(d, e, f)) \longrightarrow b'}{(x)(x(\mathsf{when}(d, e, f)))b \longrightarrow b'}\ \equiv, \beta}{\mathsf{funsplit}((x)(x(\mathsf{when}(d, e, f))), k)}\ \text{funsplit Comp.}}{\mathsf{apply}(k, \mathsf{when}(d, e, f)) \longrightarrow b'}\ =_{\mathsf{def}}}{\mathsf{whentail}_{df}(d, e, f, k) \longrightarrow b'}\ =_{\mathsf{def}}$$

Rule 5.3: Derivation of the computation rules for $\mathsf{whentail}_{df}$

We continue by observing that evaluation of $b(\mathsf{when}(d, e, f))$ will terminate with $b'$ if evaluation of $\mathsf{when}(d, e, f)$ terminates with $w'$, and if evaluation of $b(w')$ terminates with $b'$. We can then use the two when computation rules to obtain the two rules:

$$\cfrac{\cfrac{\cfrac{k \longrightarrow \lambda(b) \qquad \cfrac{\cfrac{\cfrac{f \longrightarrow \mathsf{inl}(l) \qquad d(l) \longrightarrow d'}{\mathsf{when}(d, e, f) \longrightarrow d'}\ \text{when Comp} \qquad b(d') \longrightarrow b'}{b(\mathsf{when}(d, e, f)) \longrightarrow b'}}{(x)(x(\mathsf{when}(d, e, f)))b \longrightarrow b'}\ \equiv, \beta}{\mathsf{funsplit}((x)(x(\mathsf{when}(d, e, f))), k)}\ \text{funsplit Comp.}}{\mathsf{apply}(k, \mathsf{when}(d, e, f)) \longrightarrow b'}\ =_{\mathsf{def}}}{\mathsf{whentail}_{df}(d, e, f, k) \longrightarrow b'}\ =_{\mathsf{def}}$$

Rule 5.4: Derivation of a computation rule for $\mathsf{whentail}_{df}$ 1

and Rule 5.5 on the next page.

Rules 5.4 and 5.5 can be re-written as:

$$\cfrac{f \longrightarrow \mathsf{inl}(l) \qquad d(l) \longrightarrow d' \qquad k \longrightarrow \lambda(b) \qquad b(d') \longrightarrow b'}{\mathsf{whentail}_{df}(d, e, f, k) \longrightarrow b'}\ \mathsf{whentail}_{df}\ \text{Comp}$$

Rule 5.6: $\mathsf{whentail}_{df}$ computation 1

and:

$$\dfrac{\dfrac{k \longrightarrow \lambda(b) \qquad \dfrac{\dfrac{\dfrac{f \longrightarrow \mathsf{inr}(r) \qquad e(r) \longrightarrow e'}{\mathsf{when}(d, e, f) \longrightarrow e'}\,\text{when Comp} \qquad b(e') \longrightarrow b'}{b(\mathsf{when}(d, e, f)) \longrightarrow b'}}{(x)(x(\mathsf{when}(d, e, f)))b \longrightarrow b'}\,\equiv, \beta}{\mathsf{funsplit}((x)(x(\mathsf{when}(d, e, f))), k)}\,\mathsf{funsplit}\text{ Comp.}}{\dfrac{\mathsf{apply}(k, \mathsf{when}(d, e, f)) \longrightarrow b'}{\mathsf{whentail}_{df}(d, e, f, k) \longrightarrow b'}\,=_{\text{def}}}\,=_{\text{def}}$$

Rule 5.5: Derivation of a computation rule for $\mathsf{whentail}_{df}$ 2

$$\dfrac{f \longrightarrow \mathsf{inr}(r) \qquad e(r) \longrightarrow e' \qquad k \longrightarrow \lambda(b) \qquad b(e') \longrightarrow b'}{\mathsf{whentail}_{df}(d, e, f, k) \longrightarrow b'}\,\mathsf{whentail}_{df}\text{ Comp}$$

Rule 5.7: $\mathsf{whentail}_{df}$ computation 2

Rules 5.6 and 5.7 are precisely the same as rules Rules 5.1 and 5.2, respectively.

### Elimination rule

Just as we derived a computation rule for $\mathsf{whentail}_{df}$ by expanding its definition we can derive an elimination rule: that is we derive a rule which has as a conclusion a judgement about the type of $\mathsf{whentail}_{df}(d, e, f, k)$. We expand the definition of $\mathsf{whentail}_{df}$ and then use the for $\Pi$ and $+$ elimination (Rules 2.19 and 2.6) to obtain Rule 5.12 on page 125. This rule does not share exactly the same premisses same as Rule 5.11 on page 124. The offending premiss is a well-formedness judgement which has been left implicit in the $\Pi$ elimination rule, so in fact the rules are the same.

Since we have shown that:

$$\mathsf{whentail}(d, e, f, \lambda((x)x))$$

and:

$$\mathsf{when}(d, e, f)$$

have the same value, we can replace:

$$D(\mathsf{when}(d, e, f))$$

in Rule 5.11 with:

$$D(\mathsf{whentail}(d, e, f, \lambda((x)x)))$$

### 5.2.3 Section summary

In this section we have given two versions of a CPS non-canonical constant for the disjoint union of two types. One was presented by writing down what we considered the appropriate computation rules, and then generating an elimination rule in the usual way. The other was presented by giving a syntactic definition in terms of known constants. We showed that the computation and elimination rules for the defined version were exactly the same as those for the directly presented version.

## 5.3 CPS non-canonical constant for the disjoint union of a family of types

In this section we will treat the task of presenting a CPS non-canonical constant for the disjoint union of a family of types just as we treated the task of presenting a CPS non-canonical constant for the disjoint union of two types.

The conventional non-canonical constant for the $\Sigma$ types is split and Rule 2.11 explains how to evaluate expressions formed with split. Supplying split with a continuation means providing an auxiliary function to be applied to the result of evaluating the expression formed by split.

### 5.3.1 Presenting splittail as a primitive

First we give the computation rule, and then the elimination rule.

#### Computation rule

splittail has the following computation rule:

$$\frac{e \longrightarrow \mathsf{pair}(a,b) \quad k \longrightarrow \lambda(c) \quad d(a,b) \longrightarrow d' \quad c(d') \longrightarrow c'}{\mathsf{splittail}(d,e,k) \longrightarrow c'}\ \mathsf{splittail\ Comp}$$

Rule 5.8: splittail computation

#### Elimination rule

Rule 5.13 on page 126 is the new elimination rule we construct to allow us to compute with splittail.

This elimination rule is justified using exactly the same reasoning that we used to justify the new elimination rule presented in the previous section. The rule is also a Π elimination rule, and we find that we have a conclusion whose type may, in general, depend on the value of an expression formed using split. As before we give a syntactic definition of the CPS non-canonical constant and re-derive the rules.

### 5.3.2 Presenting $\mathsf{splittail}_{df}$ as a defined constant

Just as we did previously we can define $\mathsf{splittail}_{df}$:

$$\begin{aligned}\mathsf{splittail}_{df}(d,e,k) &=_{\mathsf{def}} \mathsf{apply}(k,\mathsf{split}(d,e)) \\ &=_{\mathsf{def}} \mathsf{funsplit}((y)(y(\mathsf{split}(d,e))),k)\end{aligned}$$

As we have done previously we present the computation rule and then the elimination rule.

### Computation rule

The computation rule for $\mathsf{splittail}_{df}$ can be derived just as we derived the computation rule for $\mathsf{whentail}_{df}$:

$$\dfrac{k \longrightarrow \lambda(c) \qquad \dfrac{\dfrac{e \longrightarrow \mathsf{pair}(a,b) \qquad d(a,b) \longrightarrow d'}{\mathsf{split}(d,e) \longrightarrow d'}\,\text{split Comp.} \qquad c(d') \longrightarrow c'}{c(\mathsf{split}(d,e)) \longrightarrow c'}}{\dfrac{\mathsf{apply}(k, \mathsf{split}(d,e)) \longrightarrow c'}{\mathsf{splittail}_{df}(d,e,k) \longrightarrow c'}\,=_{\mathsf{def}}}\,apply\ \text{Comp.}$$

Rule 5.9: Derivation of $\mathsf{splittail}_{df}$ computation

This rule can be re-written as:

$$\dfrac{k \longrightarrow \lambda(c) \qquad e \longrightarrow \mathsf{pair}(a,b) \qquad d(a,b) \longrightarrow d' \qquad c(d') \longrightarrow c'}{\mathsf{splittail}_{df}(d,e,k) \longrightarrow c'}$$

Rule 5.10: $\mathsf{splittail}_{df}$ computation

As we expected, Rule 5.10 is just Rule 5.8.

### Elimination rule

We can use the $\Sigma$ and $\Pi$ elimination rules, along with the definition of $\mathsf{splittail}_{df}$ to produce an elimination rule, given in Rule 5.14 on page 127. This rule is, with the exception of the premiss relating to the well-formedness of the family of types $D$ suppressed in the $\Pi$ elimination rule, just Rule 5.13.

Since we have shown that:

$$\mathsf{splittail}(d, e, \lambda((x)x))$$

and:

$$\mathsf{split}(d, e)$$

have the same value, we can replace:

$$D(\mathsf{split}(d, e)))$$

in Rule 5.13 with:

$$D(\mathsf{splittail}(d, e, \lambda((x)x)))$$

### 5.3.3 Section summary

Again we have given two versions of a CPS non-canonical constant, and again we have shown that both versions lead us to produce the same rules.

## 5.4 Cartesian product of a family of types: Π

Following our understanding of a continuation as a function to apply after computing a value we might decide to define immediately a CPS version of apply as:

$$\mathsf{applytail}(f, a, k) =_{\mathsf{def}} \mathsf{apply}(k, \mathsf{apply}(f, a))$$

Thus supplying a function with another function to call after itself does not provide much extra technical interest.

## 5.5 Other non-inductive types

Defining CPS non-canonical constants for the other non-inductively defined types that we have presented follows the same pattern as the one we have presented already.

## 5.6 An observation about equality

Since we have made definitions like:

$$\mathsf{whentail}(d, e, f, k) =_{\mathrm{def}} \mathsf{apply}(k, \mathsf{when}(d, e, f))$$

we can, if the expression is well typed, make equality judgements about $\mathsf{whentail}(d, e, f, k)$ and $\mathsf{apply}(k, \mathsf{when}(d, e, f))$. In the next chapter we will see that we cannot make such syntactic definitions for the CPS non-canonical constants associated with the inductively defined types, but we can make a related judgement of equality.

## 5.7 Chapter Summary

In this chapter we have presented CPS non-canonical constants, and the associated elimination rules, for some types which are not defined inductively. Since the elimination rules look unorthodox we also presented the CPS non-canonical constants as defined constants, and derived the associated elimination rules.

$$\dfrac{f : A + B \qquad \begin{array}{c}[z : A + B]\\ \vdots\\ C(z)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[v : C(f)]\\ \vdots\\ D(v)\ \mathsf{type}\end{array} \qquad k : \Pi(C(f), D) \qquad \begin{array}{c}[x : A]\\ \vdots\\ d(x) : C(inl(x))\end{array} \qquad \begin{array}{c}[y : B]\\ \vdots\\ e(y) : C(inr(y))\end{array}}{\mathsf{whentail}(d, e, f, k) : D(\mathsf{when}(d, e, f))}$$

Rule 5.11: + elimination using whentail

$$\dfrac{\dfrac{k : \Pi(C(f), D) \qquad \dfrac{f : A + B \qquad \begin{matrix}[w : A + B]\\ \vdots \\ C(w)\ \mathsf{type}\end{matrix} \qquad \begin{matrix}[x : A]\\ \vdots \\ d(x) : C(inl(x))\end{matrix} \qquad \begin{matrix}[y : B]\\ \vdots \\ e(y) : C(inr(y))\end{matrix}}{\mathsf{when}(d, e, f) : C(f)}}{\mathsf{apply}(k, \mathsf{when}(d, e, f)) : D(\mathsf{when}(d, e, f))}\ \Pi\ \mathrm{Elim}}{\mathsf{whentail}_{df}(d, e, f, k) : D(\mathsf{when}(d, e, f))}=_{\mathrm{def}}$$

Rule 5.12: Typing $\mathsf{whentail}_{df}$

$$\dfrac{c : \Sigma(A,B) \qquad \begin{matrix} [z : \Sigma(A,B)] \\ \vdots \\ C(z)\ \mathsf{type} \end{matrix} \qquad \begin{matrix} [u : C(c)] \\ \vdots \\ D(c)\ \mathsf{type} \end{matrix} \qquad k : \Pi(C(c), D) \qquad \begin{matrix} \begin{bmatrix} x : A \\ y(w) : B(w)[w : A] \end{bmatrix} \\ \vdots \\ d(x,y) : C(\mathsf{pair}(x,y)) \end{matrix}}{\mathsf{splittail}(d,c,k) : D(\mathsf{split}(d,c))}$$

Rule 5.13: Σ elimination using splittail

$$\dfrac{k : \Pi(C(e), D) \qquad \dfrac{e : \Sigma(A, B) \qquad \begin{matrix}[z : \Sigma(A, B)] \\ \vdots \\ C(z)\ \mathsf{type}\end{matrix} \qquad \begin{matrix}\left[\begin{matrix} x : A \\ y(w) : B(w)[w : A]\end{matrix}\right] \\ \vdots \\ d(x, y) : C(\mathsf{pair}(x, y))\end{matrix}}{\mathsf{split}(d, e) : C(e)}\ \Sigma\ \mathrm{Elim}}{\dfrac{\mathsf{apply}(k, \mathsf{split}(d, e)) : D(\mathsf{split}(d, e))}{\mathsf{splittail}_{df}(d, e, k) : D(\mathsf{split}(d, e))}\ =_{\mathrm{def}}}\ \Pi\ \mathrm{Elim}$$

Rule 5.14: Derivation of $\Sigma$ elimination with $\mathsf{splittail}_{df}$

# 6. CPS non-canonical constants for inductive types

In this chapter we present computation rules, and the associated elimination rules, for CPS non-canonical constants for the inductive types we discussed in Chapter 2:

- natural numbers;
- polymorphic lists;
- binary trees.

We compare these rules to the rules which use structurally recursive non-canonical constants, and in the case of the natural numbers compare our CPS non-canonical constant to a tail-recursive operator defined in [110].

In Chapter 2 we saw that, in M-LTT, computations using values of inductively defined types, like the natural numbers and polymorphic lists, typically use structural recursion. Generally, a function over these types is defined in terms of a non-canonical constant which is the structural recursion operator for the type. In Chapter 3 we saw a number of advantages of tail-recursive functions. In this chapter we provide a natural way to express and exploit tail-recursive functions in M-LTT.

The relationship between the elimination rules which we present in this chapter and the standard elimination rules for the inductively defined types is similar to the relationship between the elimination rules for the $+$ and $\Sigma$ types which we presented in Chapter 5 and the usual elimination rules for the + and Σ types. We cannot, however, give such a simple definition of the CPS non-canonical constants as we were able to in Chapter 5. We can show an equality between the CPS non-canonical constant and the structurally recursive ones.

## 6.1 Natural numbers

As always we present the computation rules, and then construct an elimination rule.

Rules 2.40 and 2.41 are the natural number introduction rules.

### 6.1.1 Computation rules

Rules 6.1 and 6.2 are the rules for using tail recursion with the natural numbers.

$$\frac{n \longrightarrow \mathsf{zero} \qquad k \longrightarrow \lambda(b) \qquad b(d) \longrightarrow b'}{\mathsf{nattail}(d, e, n, k) \longrightarrow b'}\ \mathsf{nattail}\ \text{Comp.}$$

Rule 6.1: nattail computation 1

$$\frac{n \longrightarrow \mathsf{succ}(m) \quad k \longrightarrow \lambda(b) \quad \mathsf{nattail}(d, e, m, \lambda((x)b(e(m, x)))) \longrightarrow e'}{\mathsf{nattail}(d, e, n, k) \longrightarrow e'}$$

Rule 6.2: nattail computation 2

As before we supply a tail-function as an auxiliary argument to the non-canonical constant.

### 6.1.2 Elimination rule

We construct the elimination rule using exactly the same reasoning that we have used previously. We are trying to infer a type for:

$$\mathsf{nattail}(d, e, n, k)$$

The premises to the rule are justified as follows.

First we must be able to judge:

$$n : \mathsf{Nat}$$

Just as in the usual Nat elimination rule we need to have premisses:

$$\begin{array}{c}[x : \mathsf{Nat}] \\ \vdots \\ C(x)\ \mathsf{type}\end{array}$$

and:

$$d : C(\mathsf{zero})$$

and:

$$\begin{array}{c}\left[\begin{array}{c} p : \mathsf{Nat} \\ q : C(p) \end{array}\right] \\ \vdots \\ e(p, q) : C(\mathsf{succ}(p))\end{array}$$

We also need a premiss relating to the type of $k$. We follow the same reasoning that we used to explain the premiss relating to the types of the continuations we used in the previous chapter. We need to know that $D$ is a family of types over $C(n)$. Thus we require these premisses:

$$\begin{array}{c}[w : C(n)] \\ \vdots \\ D(w)\ \mathsf{type}\end{array}$$

and:

$$k : \Pi(C(n), D)$$

The type of $\mathsf{nattail}(d, e, n, k)$ is then $D(\mathsf{natrec}(d, e, n))$.

Rule 6.30 on page 148 is the elimination rule which we have constructed.

### 6.1.3 Equality rules

Rule 6.3 and Rule 6.31 on page 149 are the rules for equality of expressions involving $\mathsf{nattail}$.

$$\dfrac{C(\mathsf{zero})\ \mathsf{type} \qquad \begin{array}{c}[x : C(\mathsf{zero})]\\ \vdots \\ D(x)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(\mathsf{zero})]\\ \vdots \\ f(y) : D(y)\end{array} \qquad d : C(\mathsf{zero})}{\mathsf{nattail}(d, e, \mathsf{zero}, \lambda(f)) = f(d) : D(d)}\ \text{Nat Elim.} =$$

Rule 6.3: Equality of expressions involving $\mathsf{nattail}$ 1

## 6.2 Polymorphic lists

We can provide a similar treatment for polymorphic lists.

Rules 2.48 and 2.49 are the list introduction rules.

### 6.2.1 Computation rules

The rules for tail recursion are:

$$\dfrac{n \longrightarrow \mathsf{nil} \qquad k \longrightarrow \lambda(b) \qquad b(d) \longrightarrow b'}{\mathsf{listtail}(d, e, l, k) \longrightarrow b'}\ \mathsf{listtail}\ \text{Comp.}$$

Rule 6.4: $\mathsf{listtail}$ computation 1

$$\frac{l \longrightarrow \mathsf{cons}(a, as) \quad k \longrightarrow \lambda(b) \quad \mathsf{listtail}(d, \epsilon, as, \lambda((x)b(\epsilon(a, as, x)))) \longrightarrow \epsilon'}{\mathsf{listtail}(d, e, l, k) \longrightarrow e'}$$

Rule 6.5: listtail computation 2

### 6.2.2 Elimination rule

Rule 6.32 on page 150 is the elimination rule which we obtain from considering these computation rules. This rule is justified in exactly the same way as the rule for natural number elimination was.

### 6.2.3 Equality rules

Rule 6.6 and Rule 6.33 on page 151 are the rules for equality of expressions involving listtail.

$$\frac{C(\mathsf{nil})\ \mathsf{type} \qquad \begin{array}{c}[y : C(\mathsf{nil})] \\ \vdots \\ D(y)\ \mathsf{type}\end{array} \qquad d : C(\mathsf{nil}) \qquad \begin{array}{c}[z : C(\mathsf{nil})] \\ \vdots \\ b(z) : D(z)\end{array}}{\mathsf{listtail}(d, e, \mathsf{nil}, \lambda(f)) = f(d) : D(d)}$$

Rule 6.6: Equality of expressions involving listtail 1

## 6.3 Binary trees

We can give binary trees the same treatment.

Rules 2.56 and 2.57 are the binary tree introduction rules.

### 6.3.1 Computation rules

Rule 6.7 and Rule 6.34 on page 152 are the computation rules for treetail.

$$\frac{t \longrightarrow \mathsf{leaf} \qquad k \longrightarrow \lambda(f) \qquad f(d) \longrightarrow f'}{\mathsf{treetail}(d, e, t, k) \longrightarrow f'}\ \mathsf{treetail}\ \mathrm{Comp}$$

Rule 6.7: treetail computation 1

The second rule introduces one minor novelty. Whereas in the computation rule for treerec for the case of a node (Rule 2.59 on page 64) we had this expression to evaluate:

$$e(v, l, r, \mathsf{treerec}(d, e, l), \mathsf{treerec}(d, e, r))$$

in Rule 6.34 we have to evaluate:

$$\mathsf{treetail}(d, e, l, \lambda((x)\mathsf{treetail}(d, e, r, \lambda((y)f(v, l, r, x, y)))))$$

Clearly we could have chosen to evaluate:

$$\mathsf{treetail}(d, e, r, \lambda((x)\mathsf{treetail}(d, e, l, \lambda((y)f(v, l, r, y, x)))))$$

## 6.3.2 Elimination rule

Rule 6.35 on page 153 is the elimination rule which we obtain from considering these computation rules.

## 6.3.3 Equality rules

Rule 6.8 and Rule 6.36 on page 154 are the treetail equality rules.

$$\frac{C(\mathsf{leaf})\ \mathsf{type} \qquad k : \Pi(C(t), D) \qquad d : C(\mathsf{leaf})}{\mathsf{treetail}(d, e, \mathsf{leaf}, \lambda(f)) = f(d) : D(d)}$$

Rule 6.8: treetail equality

## 6.4 Comparison with the standard rules

What we have done so far is to take a different notion of computation (tail recursion, rather than structural recursion) and produced elimination rules which allow us to make judgements about computations of tail-recursive functions. These rules are slightly different from the usual rules, but justified them in the same way that the usual rules are justified. Although we have produced new *rules* we have not introduced a new *form* of rule, nor have we introduced a new *judgement* into the theory, nor have we changed the *interpretation* of any of the judgements of the theory. We *have* given ourselves direct access to a different way of writing functions. It remains useful to show that we have not introduced any new absurdity to the theory. We do this by establishing the relationship between the CPS non-canonical constants and the structurally recursive ones. Informally, we can expect a relatively simple relationship, as the CPS- and the structurally-recursive non-canonical constants have the same proof theoretic power, unlike wrec which makes appeal to a more powerful induction principle. Although the relationship here is different we are mirroring the work we did in Chapter 5. For the case of the natural numbers we also show the relationship between natrec and tprim, a tail-recursive operator for the natural numbers described in [110].

### 6.4.1 Relating natrec and nattail

In this section will justify Rule 6.37 on page 155. The conclusion of this rule is the judgement:

$$\mathsf{nattail}(d, e, n, \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, n)) : D(\mathsf{natrec}(d, e, n))$$

To justify this judgement we must show that, given the premisses of Rule 6.37:

$$\mathsf{nattail}(d, e, n, \lambda(f))$$

and

$$\mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, n))$$

evaluate to equal values of type:

$$D(\mathsf{natrec}(d, e, n))$$

The proof proceeds by induction on $n$.

For the case $n \longrightarrow \mathsf{zero}$ we must show:

$$\mathsf{nattail}(d, e, \mathsf{zero}, \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, \mathsf{zero})) : D(\mathsf{natrec}(d, e, \mathsf{zero}))$$

On the left-hand side, considering the first computation rule for nattail (Rule 6.1), we have (ignoring evaluation of canonical forms):

$$\frac{d \longrightarrow d' \qquad f(d') \longrightarrow f'}{\mathsf{nattail}(d, e, \mathsf{zero}, \lambda(f)) \longrightarrow f'}$$

Rule 6.9: Evaluating $\mathsf{nattail}(d, e, \mathsf{zero}, \lambda(f))$

The premisses to Rule 6.37 on page 155 relating to the types of $d$ and $f$ allow us to conclude that $f' : D(\mathsf{natrec}(d, e, \mathsf{zero}))$.

On the right-hand side, considering the apply computation rule (Rule 2.18 on page 39) and the first computation rule for natrec (Rule 2.42 on page 59), we have:

$$\frac{\dfrac{d \longrightarrow d'}{\mathsf{natrec}(d, e, \mathsf{zero}) \longrightarrow d'}\ \mathsf{natrec}\ \text{Comp} \qquad f(d') \longrightarrow f'}{\mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, \mathsf{zero})) \longrightarrow f'}$$

Rule 6.10: Evaluating $\mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, \mathsf{zero}))$

Hence both left- and right-hand sides evaluate to the same value of $D(\mathsf{natrec}(d, e, \mathsf{zero}))$. Thus the base case is established.

For the case $n \longrightarrow \mathsf{succ}(m)$ we must show:

$$\mathsf{nattail}(d, e, \mathsf{succ}(m), \lambda(f)) = \mathsf{apply}(\lambda(f), (\mathsf{natrec}(d, e, \mathsf{succ}(m))))$$

as a value of

$$D(\mathsf{natrec}(d, e, \mathsf{succ}(m)))$$

When evaluating $\mathsf{nattail}(d, e, \mathsf{succ}(m), \lambda(f))$, the induction step justifies this rule :

$$\frac{\mathsf{apply}(\lambda(g), \mathsf{natrec}(d, e, m)) \longrightarrow v}{\mathsf{nattail}(d, e, m, \lambda(g)) \longrightarrow v}$$

Rule 6.11: Induction step

On the left-hand side, we make use of the second nattail computation rule (Rule 6.1 on page 129), the rule we have just given, and the apply computation rule to obtain:

$$\cfrac{\cfrac{\cfrac{\cfrac{e(m, \mathsf{natrec}(d, e, m))) \longrightarrow e' \qquad f(e') \longrightarrow f'}{f(e(m, \mathsf{natrec}(d, e, m))) \longrightarrow f'}}{\mathsf{apply}(\lambda((z)f(e(m, z))), \mathsf{natrec}(d, e, m)) \longrightarrow f'}\;\text{apply Comp}}{\mathsf{nattail}(d, e, m, \lambda((z)f(e(m, z)))) \longrightarrow f'}\;\text{Induction step}}{\mathsf{nattail}(d, e, \mathsf{succ}(m), \lambda(f)) \longrightarrow f'}\;\text{nattail Comp}$$

Rule 6.12: Evaluating $\mathsf{nattail}(d, e, \mathsf{succ}(m), \lambda(f))$

On the right-hand side we have, making use of apply computation and the second natrec computation rule (Rule 2.43 on page 59):

$$\cfrac{\cfrac{\cfrac{e(m, \mathsf{natrec}(d, e, m)) \longrightarrow e'}{\mathsf{natrec}(d, e, \mathsf{succ}(m)) \longrightarrow e'}\;\text{natrec Comp} \qquad f(e') \longrightarrow f'}{f(\mathsf{natrec}(d, e, \mathsf{succ}(m))) \longrightarrow f'}}{\mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, \mathsf{succ}(m))) \longrightarrow f'}\;\text{apply Comp}$$

Rule 6.13: Evaluating $\mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, \mathsf{succ}(m)))$

The typing judgements which are premisses to Rule 6.37 allow us to conclude that:

$$\mathsf{nattail}(d, e, \mathsf{succ}(m), \lambda(f))$$

and

$$\mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, \mathsf{succ}(m)))$$

evaluate to the same value of type $D(\mathsf{natrec}(d, e, \mathsf{succ}(m)))$. So the inductive case has been dealt with.

So both cases have been handled, justifying Rule 6.37.

### Internalising the relationship between nattail and natrec

The argument given in the preceding section is an argument *outside* M-LTT. Rule 6.14, and the series of sub-proofs from Rule 6.38 on page 156 to 6.42 on page 160 *internalise* this argument. For brevity in these proofs we write $D'(p)$ for $D(g(\mathsf{natrec}(d, e, p)))$. We also suppress type judgements.

$$\dfrac{\dfrac{n : \mathsf{Nat} \qquad \pi_1 \qquad \pi_2}{\mathsf{natrec}(\mathsf{eq}, (x, y, z)\mathsf{eq}, n) : \mathsf{EQ}(\mathsf{nattail}(d, e, n, g), \mathsf{apply}(g, \mathsf{natrec}(d, e, n)), D'(n))}}{\mathsf{nattail}(d, e, n, g) = \mathsf{apply}(g, \mathsf{natrec}(d, e, n)) : D'(n))}$$

Rule 6.14: A proof that $\mathsf{nattail}(d, e, n, g)$ is equal to $\mathsf{apply}(g, \mathsf{natrec}(d, e, n))$

### Section summary

In this section we have shown that any value we can compute using nattail we can also compute using natrec. One corollary of this is that if we can construct a value in the empty type using nattail then we could have constructed the value using natrec. So we have a relative 'simple-minded consistency', to borrow a phrase from Martin-Löf [77].

A second corollary is that, if the continuation is the identity function:

$$\frac{n : \mathsf{Nat} \qquad \begin{array}{c}[x : \mathsf{Nat}]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad d : C(\mathsf{zero}) \qquad \begin{array}{c}\left[\begin{array}{c}p : \mathsf{Nat}\\ q : C(p)\end{array}\right]\\ \vdots\\ e(p,q) : C(\mathsf{succ}(p))\end{array}}{\mathsf{nattail}(d,e,n,\lambda((x)x)) = \mathsf{natrec}(d,e,n) : C(n)}$$

Rule 6.15: A corollary of Rule 6.37

So we can replace any structurally recursive function on the natural numbers with a tail-recursive one. This function need not be the only suitable tail-recursive function, of course.

A corollary of Rule 6.15 is that we can replace:

$$D(\mathsf{natrec}(d, e, n))$$

with:

$$D(\mathsf{nattail}(d, e, n, \lambda((x)x)))$$

in Rule 6.30.

### 6.4.2 Comparison with Thompson's tprim

Thompson (in §6.8 of [110]) discusses the use of M-LTT for imperative programming, and asserts that this can be (partially) done:

> 'via an identification of a particular class of functional programs, the *tail-recursive functions*, with imperative programs.'[1]

His approach is rather different from ours:

- he supposes that we already have a structurally recursive function and wish to perform some (presumably optimising) transformations on it;

[1] Emphasis in the original.

- he does not discuss continuations;
- he only considers the type Nat.

Thompson's operator for tail recursion is tprim and has the following behaviour (after uncurrying tprim and a little abuse of notation):

$$\frac{p \longrightarrow \mathsf{zero} \qquad v \longrightarrow v'}{\mathsf{tprim}(n, c, f, p, v) \longrightarrow v'}\ \mathsf{tprim}\ \text{Comp.}$$

Rule 6.16: tprim computation 1

$$\frac{p \longrightarrow \mathsf{succ}(m) \qquad m < n \qquad \mathsf{tprim}(n, c, f, m, f(n - m - 1, v)) \longrightarrow t'}{\mathsf{tprim}(n, c, f, p, v) \longrightarrow t'}$$

Rule 6.17: tprim computation 2

$$\frac{p \longrightarrow \mathsf{succ}(m) \qquad m \nless n \qquad v \longrightarrow v'}{\mathsf{tprim}(n, c, f, p, v) \longrightarrow v'}\ \mathsf{tprim}\ \text{Comp.}$$

Rule 6.18: tprim computation 3

The second argument to tprim is merely copied about and plays no active role. The last rule is slightly odd: the intention surely is that the operator will only be used when $m < n$.

Thompson states:

> 'For all $n$, $c$ and $f$, tprim $n\ c\ f\ n\ c =$ prim $n\ c\ f$'

prim is a curried variant of natrec: prim $n\ c\ f =_{\text{def}}$ natrec$(c, f, n)$.

Consider an informal evaluation of tprim$(n, c, f, p, v)$[2]:

[2] We ignore the case of Rule 6.18

$$\begin{array}{ll}
 & \mathsf{tprim}(n, c, f, p, v) \\
\longrightarrow & \mathsf{tprim}(n, c, f, p-1, f(n-p, v)) \\
\longrightarrow & \mathsf{tprim}(n, c, f, p-2, f(n-p+1, f(n-p, v))) \\
\longrightarrow & \mathsf{tprim}(n, c, f, p-3, f(n-p+2, f(n-p+1, f(n-p, v)))) \\
\longrightarrow & \ldots \\
\longrightarrow & \mathsf{tprim}(n, c, f, p-p, f(n-p+(p-1), \ldots f(n-p+1, f(n-p, v)))) \\
\longrightarrow & f(n-1, \ldots, f(n-p+1, f(n-p, v))))
\end{array}$$

Figure 6.1: Informal evaluation of $\mathsf{tprim}(n, c, f, p, v)$

If we compare this to a similar informal evaluation of $\mathsf{nattail}(d, e, q, k)$:

$$\begin{array}{ll}
 & \mathsf{nattail}(d, e, q, k) \\
\longrightarrow & \mathsf{nattail}(d, e, q-1, \lambda((x)(k(e(q-1, x))))) \\
\longrightarrow & \mathsf{nattail}(d, e, q-2, \lambda((y)((\lambda((x)(k(e(q-1, x)))))(e(q-2, y))))) \\
\longrightarrow & \ldots \\
\longrightarrow & \mathsf{nattail}(d, e, q-q, \lambda((z)((\ldots)(e(q-q, z))))) \\
\longrightarrow & \lambda((z)((\ldots)(e(0, z))))d \\
\longrightarrow & k(e(q-1, e(q-2, \ldots e(0, d))))
\end{array}$$

Figure 6.2: Informal evaluation of $\mathsf{nattail}(d, e, q, k)$

We see that we can now assert:

$$\mathsf{tprim}(n, c, f, p, v) = \mathsf{nattail}(v, (a, b)f(a+n-p, b), p, \lambda((x)x))$$

### Section summary

In this section we have shown that we can represent a tail-recursive, not CPS, operator presented in [110] using nattail. We gave a direct characterisation of tprim, although we could have used Rule 6.37, and the characterisation of tprim in terms of natrec given in [110].

### 6.4.3 Relating listrec and listtail

In this section we relate listrec and listtail. We justify Rule 6.43 on page 161. The justification is similar to that given in §6.4.1 for Rule 6.37, and proceeds by list induction.

First we consider the case $l \longrightarrow \mathsf{nil}$. We are seeking to justify:

$$\mathsf{listtail}(d, e, \mathsf{nil}, \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{listrec}(d, e, \mathsf{nil})) : D(\mathsf{listrec}(d, e, \mathsf{nil}))$$

On the left-hand side we use the first listtail computation rule (Rule 6.4) to obtain:

$$\frac{d \longrightarrow d' \qquad f(d') \longrightarrow f'}{\mathsf{listtail}(d, e, \mathsf{nil}, \lambda(f)) \longrightarrow f'}$$

Rule 6.19: Evaluating $\mathsf{listtail}(d, e, \mathsf{nil}, \lambda(f))$

On the right-hand side we use the first listrec computation rule (Rule 2.50 on page 61), and the apply computation rule to obtain:

$$\frac{\dfrac{d \longrightarrow d'}{\mathsf{listrec}(d, e, \mathsf{nil}) \longrightarrow d'}\ \mathsf{listrec}\ \mathrm{Comp} \qquad f(d') \longrightarrow f'}{\mathsf{apply}(\lambda(f), \mathsf{listrec}(d, e, \mathsf{nil})) \longrightarrow f'}$$

Rule 6.20: Evaluating $\mathsf{apply}(\lambda(f), \mathsf{listrec}(d, e, \mathsf{nil}))$

The premisses of Rule 6.43 establish that $f' : D(\mathsf{natrec}(d, e, \mathsf{nil})))$. So the base case has been established.

The second case is where $l \longrightarrow \mathsf{cons}(a, as)$. We are trying to establish that:

$$\mathsf{listtail}(d, e, \mathsf{cons}(a, as), \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{listrec}(d, e, \mathsf{cons}(a, as)))$$

as a value of

$$D(\mathsf{listrec}(d, e, \mathsf{cons}(a, as)))$$

When evaluating $\mathsf{listtail}(d, e, \mathsf{cons}(a, as), \lambda(k))$ the induction step justifies this rule:

$$\frac{\mathsf{apply}(\lambda(g), \mathsf{listrec}(d, e, as)) \longrightarrow v}{\mathsf{listtail}(d, e, as, \lambda(g)) \longrightarrow v}$$

Rule 6.21: Induction step

On the left-hand side, using the second nattail computation rule (Rule 6.2 on page 129), the rule we have just presented, and apply computation, we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{e(a, as, \mathsf{listrec}(d, e, as)) \longrightarrow e' \qquad f(e') \longrightarrow f'}{f(e(a, as, \mathsf{listrec}(d, e, as))) \longrightarrow f'}}{\mathsf{apply}(\lambda((x)(f(e(a, as, x)))), \mathsf{listrec}(d, e, as)) \longrightarrow f'}\;\text{apply Comp}}{\mathsf{listtail}(d, e, as, \lambda((x)(f(e(a, as, x))))) \longrightarrow f'}\;\text{Induction step}}{\mathsf{listtail}(d, e, \mathsf{cons}(a, as), \lambda(f)) \longrightarrow f'}\;\text{listtail Comp}$$

Rule 6.22: Evaluating $\mathsf{listtail}(d, e, \mathsf{cons}(a, as), \lambda(f))$

On the right-hand side, using apply computation and the second natrec computation rule (Rule 2.43 on page 59), we have:

$$\dfrac{\dfrac{\dfrac{e(a, as, \mathsf{listrec}(d, e, as)) \longrightarrow e'}{\mathsf{listrec}(d, e, \mathsf{cons}(a, as)) \longrightarrow e'}\;\text{listrec Comp} \qquad f(e') \longrightarrow f'}{f(\mathsf{listrec}(d, e, \mathsf{cons}(a, as))) \longrightarrow f'}}{\mathsf{apply}(\lambda(f), \mathsf{listrec}(d, e, \mathsf{cons}(a, as))) \longrightarrow f'}\;\text{apply Comp}$$

Rule 6.23: Evaluating $\mathsf{apply}(\lambda(f), \mathsf{listrec}(d, e, \mathsf{cons}(a, as)))$

The premisses of Rule 6.43 allow us to infer that left- and right-hand sides evaluate to the same value of type $D(\mathsf{listrec}(d, e, \mathsf{cons}(a, as)))$.

Hence the inductive case has been dealt with.

Hence we have justified Rule 6.43.

### Internalising the relationship between listtail and listrec

Just as we could internalise the relationship between nattail and natrec, so we can internalise the relationship between listtail and listrec. The proof follows the same pattern as 6.14 on page 137, and, in the interests of brevity, we omit it.

### Section summary

In this section we have shown that any value we can compute using listtail we can also compute using listrec. So, as with the natural numbers, we retain 'simple-minded' consistency.

Just as with the natural numbers, if the continuation is the identity function, we have:

$$\dfrac{l : \mathsf{List}(A) \qquad \begin{array}{c}[x : \mathsf{List}(A)]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad d : C(\mathsf{nil}) \qquad \begin{array}{c}\left[\begin{array}{c} a : A \\ as : \mathsf{List}(A) \\ q : C(as)\end{array}\right] \\ \vdots \\ e(a, as, q) : C(\mathsf{cons}(a, as))\end{array}}{\mathsf{listtail}(d, e, l, \lambda((x)x)) = \mathsf{listrec}(d, e, l) : C(l)}$$

Rule 6.24: A corollary of Rule 6.43

A corollary of Rule 6.24 is that we can replace:

$$D(\mathsf{listrec}(d, e, l))$$

with:

$$D(\mathsf{listtail}(d, e, l, \lambda((x)x)))$$

in Rule 6.32.

### 6.4.4 Relating treerec and treetail

For completeness we justify Rule 6.44 on page 162. The justification follows the same pattern as before, this time using structural induction on binary trees.

For the case $t \longrightarrow \mathsf{leaf}$, we have to show that:

$$\mathsf{treetail}(d, e, \mathsf{leaf}, \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{treerec}(d, e, \mathsf{leaf}))$$

as a value of type $D(\mathsf{treerec}(d, e, \mathsf{leaf}))$.

On the left-hand side we use the first treetail computation rule (Rule 6.7 on page 133) to obtain:

$$\frac{d \longrightarrow d' \qquad f(d') \longrightarrow f'}{\mathsf{treetail}(d, e, \mathsf{zero}, \lambda(f)) \longrightarrow f'}$$

Rule 6.25: Evaluating $\mathsf{treetail}(d, e, \mathsf{leaf}, \lambda(f))$

On the right-hand side we the use first treerec computation rule (Rule 2.58 on page 63) and the apply computation rule to obtain:

$$\cfrac{\dfrac{d \longrightarrow d'}{\mathsf{treerec}(d, e, \mathsf{leaf}) \longrightarrow d'}\,\text{treerec Comp} \qquad f(d') \longrightarrow f'}{\dfrac{f(\mathsf{treerec}(d, e, \mathsf{leaf})) \longrightarrow f'}{\mathsf{apply}(\lambda(f), \mathsf{treerec}(d, e, \mathsf{leaf})) \longrightarrow f'}\,\text{apply Comp}}$$

Rule 6.26: Evaluating $\mathsf{apply}(\lambda(f), \mathsf{treerec}(d, e, \mathsf{leaf}))$

The premisses of Rule 6.44 allow us to infer that both left- and right-hand sides are values of $D(\mathsf{treerec}(d, e, \mathsf{leaf}))$.

So the base case has been established.

In the case $t \longrightarrow \mathsf{node}(v, l, r)$ we are trying to establish:

$$\mathsf{treetail}(d, e, \mathsf{node}(v, l, r), \lambda(f)) = \mathsf{apply}(\lambda\,(f), \mathsf{treerec}(d, e, \mathsf{node}(v, l, r)))$$

as a value of type:

$$D(\mathsf{treerec}(d, e, \mathsf{node}(v, l, r)))$$

The induction step licences the use of following rule when evaluating $\mathsf{treetail}(d, e, \mathsf{node}(v, l, r), \lambda(f))$, where $s$ is either $l$ or $r$:

$$\frac{\mathsf{apply}(\lambda\,(g), \mathsf{treerec}(d, e, s)) \longrightarrow v}{\mathsf{treetail}(d, e, s, \lambda(g)) \longrightarrow v}$$

Rule 6.27: Induction step

On the left-hand side, we use the apply computation rule, the second treetail computation rule (Rule 6.34 on page 152) and the induction step we have just presented to obtain:

$$\dfrac{\dfrac{\mathsf{apply}(\lambda((x)\mathsf{treetail}(d, e, r, \lambda((y)f(e(v, l, r, x, y))))), \mathsf{treerec}(d, e, l)) \longrightarrow f'}{\mathsf{treetail}(d, e, l, \lambda((x)\mathsf{treetail}(d, e, r, \lambda((y)f(e(v, l, r, x, y)))))) \longrightarrow f'}\;\text{Ind.}}{\mathsf{treetail}(d, e, \mathsf{node}(v, l, r), \lambda(f)) \longrightarrow f'}\;\mathsf{treetail}$$

where the uppermost premiss is derived by

$$\dfrac{\dfrac{\dfrac{\dfrac{e(v, l, r, \mathsf{treerec}(d, e, l), \mathsf{treerec}(d, e, r)) \longrightarrow e' \qquad f(e') \longrightarrow f'}{f(e(v, l, r, \mathsf{treerec}(d, e, l), \mathsf{treerec}(d, e, r))) \longrightarrow f'}}{\mathsf{apply}(\lambda\,((y)f(e(v, l, r, \mathsf{treerec}(d, e, l), y))), \mathsf{treerec}(d, e, r)) \longrightarrow f'}\;\mathsf{apply}}{\mathsf{treetail}(d, e, r, \lambda((y)f(e(v, l, r, \mathsf{treerec}(d, e, l), y)))) \longrightarrow f'}\;\text{Ind.}}{\mathsf{apply}(\lambda((x)\mathsf{treetail}(d, e, r, \lambda((y)f(e(v, l, r, x, y))))), \mathsf{treerec}(d, e, l)) \longrightarrow f'}$$

Rule 6.28: Evaluating $\mathsf{treetail}(d, e, \mathsf{node}(v, l, r), \lambda(f))$

On the right-hand side we use the apply computation rule and the second treerec computation rule (Rule 2.59 on page 64) to obtain:

$$\dfrac{\dfrac{\dfrac{e(v, l. r, \mathsf{treerec}(d, e, l), \mathsf{treerec}(d, e, r)) \longrightarrow e'}{\mathsf{treerec}(d, e, \mathsf{node}(v, l, r)) \longrightarrow e'}\ \mathsf{treerec}\ \text{Comp} \qquad f(e') \longrightarrow f'}{f(\mathsf{treerec}(d, e, \mathsf{node}(v, l, r))) \longrightarrow f'}}{\mathsf{apply}(\lambda(f), \mathsf{treerec}(d, e, \mathsf{node}(v, l, r))) \longrightarrow f'}\ \mathsf{apply}\ \text{Comp}$$

Rule 6.29: Evaluating $\mathsf{apply}(\lambda(f), \mathsf{treerec}(d, \epsilon.\ \mathsf{node}(v, l, r)))$

The premisses of Rule 6.44 allow us to infer that both left- and right-hand sides are values of $D(\mathsf{treerec}(d, e, \mathsf{node}(v, l, r)))$.

So the inductive case has been established.

So both cases have been established and we have justified Rule 6.44.

### Internalising the relationship between treetail and treerec

Just as we could internalise the relationship between nattail and natrec, so we can internalise the relationship between treetail and treerec. The proof follows the same pattern as 6.14 on page 137, and, in the interests of brevity we omit it.

### Section summary

In this section we have shown that any value we can compute using treetail we can also compute using treerec. So, as with the natural numbers and polymorphic lists, we retain 'simple-minded' consistency.

Just as with the natural numbers, if the continuation is the identity function, we can construct a tail-recursive function from a structurally recursive one, as shown in Rule 6.45 on page 163.

Also as before, we can replace:

$$D(\mathsf{treerec}(d, e, t))$$

with:

$$D(\mathsf{treetail}(d, e, t, \lambda((x)x)))$$

in Rule 6.35.

## 6.5 Chapter Summary

In this chapter we have:

- presented CPS non-canonical constants for common inductively defined types;
- presented elimination rules which use these non-canonical constants;
- shown the relationship between the CPS non-canonical constants and the structurally recursive non-canonical constants, and consequently that:
    - the new rules preserve 'simple-minded consistency';
    - there is a simple way to construct a tail-recursive function, given a structurally recursive one;
    - although our initial justification of the new elimination rules made use of the structurally-recursive non-canonical constant, this can be removed.
- shown the relationship between our nattail and tprim, another tail-recursive operator for the natural numbers.

$$\dfrac{n : \mathsf{Nat} \qquad \begin{array}{c}[x : \mathsf{Nat}]\\ \vdots \\ C(x)\,\mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(n)]\\ \vdots \\ D(y)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[v : \mathsf{Nat}]\\ \vdots \\ k : \Pi(C(v), D)\end{array} \qquad d : C(\mathsf{zero}) \qquad \begin{array}{c}\begin{bmatrix} p : \mathsf{Nat} \\ z : C(p) \end{bmatrix}\\ \vdots \\ e(p,z) : C(\mathsf{succ}(p))\end{array}}{\mathsf{nattail}(d, e, n, k) : D(\mathsf{natrec}(d, e, n))}\ \text{Nat Elim.}$$

Rule 6.30: Natural number elimination with tail recursion

$$\dfrac{n : \mathsf{Nat} \quad \begin{array}{c}[x : \mathsf{Nat}]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \quad \begin{array}{c}[w : C(n)]\\ \vdots \\ D(w)\ \mathsf{type}\end{array} \quad \begin{array}{c}[y : C(n)]\\ \vdots \\ f(y) : D(y)\end{array} \quad d : C(\mathsf{zero}) \quad \begin{array}{c}\begin{bmatrix} p : \mathsf{Nat} \\ z : C(p)\end{bmatrix}\\ \vdots \\ e(p,z) : C(\mathsf{succ}(p))\end{array}}{\mathsf{nattail}(d, e, \mathsf{succ}(n), \lambda(f)) = \mathsf{nattail}(d, e, n, \lambda((u)f(e(n,u)))) : D(\mathsf{natrec}(d, e, \mathsf{succ}(n)))}\ \text{Nat Elim.} =$$

Rule 6.31: Equality of expressions involving nattail 2

$$\frac{l : \mathsf{List}(A) \qquad \begin{array}{c}[x : \mathsf{List}(A)]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(l)]\\ \vdots \\ D(y)\ \mathsf{type}\end{array} \qquad k : \Pi(C(l), D) \qquad d : C(\mathsf{nil}) \qquad \begin{array}{c}\left[\begin{array}{c} p : A \\ q : \mathsf{List}(A) \\ r : C(q) \end{array}\right]\\ \vdots \\ e(p,q,r) : C(\mathsf{cons}(p,q))\end{array}}{\mathsf{listtail}(d, e, l, k) : D(\mathsf{listrec}(d, e, l))}$$

Rule 6.32: List elimination with tail recursion

$$
\frac{
a : A \quad as : \mathsf{List}(A) \quad
\begin{array}{c}[x : \mathit{List}(A)] \\ \vdots \\ C(x)\ \mathsf{type}\end{array} \quad
\begin{array}{c}[y : C(\mathsf{cons}(a, as))] \\ \vdots \\ D(y)\ \mathsf{type}\end{array} \quad
\begin{array}{c}[z : C(\mathsf{cons}(a, as))] \\ \vdots \\ b(z) : D(z)\end{array} \quad
d : C(\mathsf{nil}) \quad
\begin{array}{c}\begin{bmatrix} p : A \\ q : \mathsf{List}(A) \\ r : C(q) \end{bmatrix} \\ \vdots \\ e(p, q, r) : C(\mathsf{cons}(p, q))\end{array}
}{
\mathsf{listtail}(d, e, \mathsf{cons}(a, as), \lambda(f)) = \mathsf{listtail}(d, e, as, \lambda((x)(f(e(a, as, x))))) : D(\mathsf{listrec}(d, e, \mathsf{cons}(a, as)))
}
$$

Rule 6.33: Equality of expressions involving listtail 2

$$\frac{t \longrightarrow \mathsf{node}(v, l, r) \qquad k \longrightarrow \lambda(f) \qquad \mathsf{treetail}(d, e, l, \lambda((x)\mathsf{treetail}(d, e, r, \lambda((y)f(v, l, r, x, y))))) \longrightarrow f'}{\mathsf{treetail}(d, e, t, k) \longrightarrow f'}\ \mathsf{treetail}\,\mathrm{Comp}$$

Rule 6.34: treetail computation 2

$$
\frac{
t : \mathsf{BinTree}(A)
\qquad
\begin{array}{c}[x : \mathsf{BinTree}(A)]\\ \vdots \\ C(x)\ \mathsf{type}\end{array}
\qquad
\begin{array}{c}[y : C(t)]\\ \vdots \\ D(y)\ \mathsf{type}\end{array}
\qquad
k : \Pi(C(t), D)
\qquad
d : C(\mathsf{leaf})
\qquad
\begin{array}{c}\left[\begin{array}{c} z : A \\ l : \mathsf{BinTree}(A) \\ r : \mathsf{BinTree}(A) \\ u : C(l) \\ v : C(r) \end{array}\right]\\ \vdots \\ e(z, l, r, u, v) : C(\mathsf{node}(z, l, r))\end{array}
}{
\mathsf{treetail}(d, e, t, k) : D(\mathsf{treerec}(d, e, t))
}
$$

Rule 6.35: Binary tree elimination with tail recursion

$$\dfrac{v : A \quad l : \mathsf{BinTree}(A) \quad r : \mathsf{BinTree}(A) \quad \begin{array}{c}[x : \mathsf{BinTree}(A)] \\ \vdots \\ C(x)\ \mathsf{type}\end{array} \quad \begin{array}{c}[y : C(\mathsf{node}(v,l,r))] \\ \vdots \\ D(y)\ \mathsf{type}\end{array} \quad \begin{array}{c}[z : C(\mathsf{node}(v,l,r))] \\ \vdots \\ f(z) : D(z)\end{array} \quad d : C(\mathsf{leaf}) \quad \begin{array}{c}\left[\begin{array}{c} w : A \\ ll : \mathsf{BinTree}(A) \\ rr : \mathsf{BinTree}(A) \\ u : C(l) \\ v : C(r)\end{array}\right] \\ \vdots \\ \epsilon(w,ll,rr,u,v) : C(\mathsf{node}(w,ll,rr))\end{array}}{\mathsf{treetail}(d,\epsilon,\mathsf{node}(v,l,r),\lambda(f)) = \mathsf{treetail}(d,\epsilon,l,\lambda((x)\mathsf{treetail}(d,\epsilon,r,\lambda((y)f(v,x,y))))) : D(\mathsf{treerec}(d,\epsilon,t))}$$

Rule 6.36: treetail equality 2

$$\frac{n : \mathsf{Nat} \qquad \begin{array}{c}[x : \mathsf{Nat}]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(n)]\\ \vdots \\ D(y)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(n)]\\ \vdots \\ f(y) : D(y)\end{array} \qquad d : C(\mathsf{zero}) \qquad \begin{array}{c}\begin{bmatrix} p : \mathsf{Nat} \\ z : C(p) \end{bmatrix}\\ \vdots \\ e(p,z) : C(\mathsf{succ}(p))\end{array}}{\mathsf{nattail}(d, e, n, \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, n)) : D(\mathsf{natrec}(d, e, n))}$$

Rule 6.37: When nattail and natrec are the same

$$\dfrac{\dfrac{d : C(\mathsf{zero}) \quad g : \Pi(C, D)}{\mathsf{nattail}(d, e, \mathsf{zero}, g) = g(d) : D'(\mathsf{zero})} \quad \dfrac{\dfrac{g : \Pi(C, D) \quad \dfrac{d : C(\mathsf{zero})}{\mathsf{natrec}(d, e, \mathsf{zero}) = d : C(\mathsf{zero})}}{g(d) = g(\mathsf{natrec}(d, e, \mathsf{zero})) : D'(\mathsf{zero})} \quad \dfrac{g : \Pi(C, D) \quad \dfrac{d : C(\mathsf{zero})}{\mathsf{natrec}(d, e, \mathsf{zero}) : C(\mathsf{zero})}}{g(\mathsf{natrec}(d, e, \mathsf{zero})) = \mathsf{apply}(g, \mathsf{natrec}(d, e, \mathsf{zero})) : D'(\mathsf{zero})}}{g(d) = \mathsf{apply}(g, \mathsf{natrec}(d, e, \mathsf{zero})) : D'(\mathsf{zero})}}{\dfrac{\mathsf{nattail}(d, e, \mathsf{zero}, g) = \mathsf{apply}(g, \mathsf{natrec}(d, e, \mathsf{zero})) : D'(\mathsf{zero})}{\mathsf{eq} : \mathsf{EQ}(\mathsf{nattail}(d, e, \mathsf{zero}, g), \mathsf{apply}(g, \mathsf{natrec}(d, e, \mathsf{zero})), D'(\mathsf{zero}))}}$$

Rule 6.38: Sub-proof $\pi_1$

$$\dfrac{\dfrac{\dfrac{[m:\mathsf{Nat}] \quad d:C(\mathsf{zero}) \quad \begin{matrix}\begin{bmatrix}p:\mathsf{Nat}\\ q:C(p)\end{bmatrix}\\ \vdots \\ e(p,q):C(\mathsf{succ}(p))\end{matrix} \quad g:\Pi(C,D)}{\mathsf{nattail}(d,e,\mathsf{succ}(m),g)=\mathsf{nattail}(d,e,m,\lambda((x)(g(e(m,x)))))):D'(\mathsf{succ}(m))} \qquad \dfrac{\pi_{2.1} \quad \pi_{2.2}}{\mathsf{nattail}(d,e,m,\lambda((x)(g(e(m,x)))))=\mathsf{apply}(g,\mathsf{natrec}(d,e,\mathsf{succ}(m))):D'(\mathsf{succ}(m))}}{\mathsf{nattail}(d,e,\mathsf{succ}(m),g)=\mathsf{apply}(g,\mathsf{natrec}(d,e,\mathsf{succ}(m))):D'(\mathsf{succ}(m))}}{\mathsf{eq}:\mathsf{EQ}(\mathsf{nattail}(d,e,\mathsf{succ}(m),g),\mathsf{apply}(g,\mathsf{natrec}(d,e,\mathsf{succ}(m))),D'(\mathsf{succ}(m)))}$$

Rule 6.39: Sub-proof $\pi_2$

$$\dfrac{[q : \Pi(\Pi(C, D), (f)\mathsf{EQ}(\mathsf{nattail}(d, c, m, f), \mathsf{apply}(f, \mathsf{natrec}(d, c, m)), D'(\mathsf{succ}(m))))] \qquad \dfrac{g : \Pi(C, D) \quad [m : \mathsf{Nat}] \quad \begin{array}{c} \begin{bmatrix} p : \mathsf{Nat} \\ q : C(p) \end{bmatrix} \\ \vdots \\ c(p, q) : C(\mathsf{succ}(p)) \end{array}}{\lambda((x)g(c(m, x))) : \Pi(C, D)}}{\dfrac{\mathsf{apply}(q, \lambda((x)g(c(m, x)))) : EQ(\mathsf{nattail}(d, c, m, \lambda((x)g(c(m, x)))), \mathsf{apply}(\lambda((x)g(c(m, x))), \mathsf{natrec}(d, c, m)), D'(\mathsf{succ}(m))))}{\mathsf{nattail}(d, c, m, \lambda((x)g(c(m, x)))) = \mathsf{apply}(\lambda((x)g(c(m, x))), \mathsf{natrec}(d, c, m)) : D'(\mathsf{succ}(m))}}$$

Rule 6.40: Sub-proof $\pi_{2.1}$

$$\dfrac{\dfrac{[m : \mathsf{Nat}] \quad g : \Pi(C, D) \qquad \begin{matrix} \begin{bmatrix} p : \mathsf{Nat} \\ q : C(p) \end{bmatrix} \\ \vdots \\ e(p, q) : C(\mathsf{succ}(p)) \end{matrix} \qquad d : C(\mathsf{zero})}{\mathsf{apply}(\lambda((x)g(e(m, x))), \mathsf{natrec}(d, e, m)) = g(e(m, \mathsf{natrec}(d, e, m))) : D'(\mathsf{succ}(m))} \quad \pi_{2.2.1}}{\mathsf{apply}(\lambda((x)g(e(m, x))), \mathsf{natrec}(d, e, m)) = \mathsf{apply}(g, \mathsf{natrec}(d, e, \mathsf{succ}(m))) : D'(\mathsf{succ}(m))}$$

Rule 6.41: Sub-proof $\pi_{2.2}$

$$
\dfrac{
\dfrac{
\dfrac{[m : \mathsf{Nat}] \qquad \begin{matrix}\begin{bmatrix} p : \mathsf{Nat} \\ q : C'(p) \end{bmatrix} \\ \vdots \\ e(p,q) : C'(\mathsf{succ}(p))\end{matrix} \qquad d : C'(\mathsf{zero})}{e(m, \mathsf{natrec}(d, e, m)) = \mathsf{natrec}(d, e, \mathsf{succ}(m)) : C'(\mathsf{succ}(m))} \qquad g : \Pi(C', D)
}{g(e(m, \mathsf{natrec}(d, e, m))) = g(\mathsf{natrec}(d, e, \mathsf{succ}(m))) : D'(\mathsf{succ}(m))}
\qquad
\dfrac{
\dfrac{[m : \mathsf{Nat}] \qquad d : C'(\mathsf{zero}) \qquad \begin{matrix}\begin{bmatrix} p : \mathsf{Nat} \\ q : C'(p) \end{bmatrix} \\ \vdots \\ e(p,q) : C'(\mathsf{succ}(p))\end{matrix}}{\mathsf{natrec}(d, e, \mathsf{succ}(m)) : C'(\mathsf{succ}(m))} \qquad g : \Pi(C', D)
}{g(\mathsf{natrec}(d, e, \mathsf{succ}(m))) = \mathsf{apply}(g, \mathsf{natrec}(d, e, \mathsf{succ}(m))) : D'(\mathsf{succ}(m))}
}{g(e(m, \mathsf{natrec}(d, e, m))) = \mathsf{apply}(g, \mathsf{natrec}(d, e, \mathsf{succ}(m))) : D'(\mathsf{succ}(m))}
$$

Rule 6.42: Sub-proof $\pi_{2.2.1}$

$$\frac{l : \mathsf{List}(A) \qquad \begin{array}{c}[x : \mathsf{List}(A)]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(l)]\\ \vdots \\ D(y)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(l)]\\ \vdots \\ f(y) : D(y)\end{array} \qquad d : C(\mathsf{nil}) \qquad \begin{array}{c}\left[\begin{array}{c}p : A\\ q : \mathsf{List}(A)\\ r : C(q)\end{array}\right]\\ \vdots \\ e(p,q,r) : C(\mathsf{cons}(p,q))\end{array}}{\mathsf{listtail}(d,e,l,\lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{listrec}(d,e,l)) : D(\mathsf{listrec}(d,e,l))}$$

Rule 6.43: When listtail and listrec are the same

$$\frac{t : \mathsf{BinTree}(A) \quad \begin{array}{c}[x : \mathsf{BinTree}(A)]\\ \vdots \\ C(x)\ \mathsf{type}\end{array} \quad \begin{array}{c}[y : C(t)]\\ \vdots \\ D(y)\ \mathsf{type}\end{array} \quad \begin{array}{c}[z : C(\mathsf{node}(v, l, r))]\\ \vdots \\ f(z) : D(z)\end{array} \quad d : C(\mathsf{leaf}) \quad \begin{array}{c}\left[\begin{array}{c} z : A \\ l : \mathsf{BinTree}(A) \\ r : \mathsf{BinTree}(A) \\ u : C(l) \\ v : C(r) \end{array}\right]\\ \vdots \\ e(z, l, r, u, v) : C(\mathsf{node}(z, l, r))\end{array}}{\mathsf{treetail}(d, e, t, \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{treerec}(d, e, t)) : D(\mathsf{treerec}(d, e, t))}$$

Rule 6.44: When treetail and treerec are the same

$$
\frac{
t : \mathsf{BinTree}(A)
\qquad
\begin{array}{c}
[x : \mathsf{BinTree}(A)] \\
\vdots \\
C(x)\ \mathsf{type}
\end{array}
\qquad
d : C(\mathsf{leaf})
\qquad
\begin{array}{c}
\left[\begin{array}{c}
y : A \\
l : \mathsf{BinTree}(A) \\
r : \mathsf{BinTree}(A) \\
u : C(ll) \\
v : C(rr)
\end{array}\right] \\
\vdots \\
e(y, l, r, u, v) : C(\mathsf{node}(y, l, r))
\end{array}
}{
\mathsf{treetail}(d, e, t, \lambda((x)x)) = \mathsf{treerec}(d, e, t) : C(t)
}
$$

Rule 6.45: A corollary of Rule 6.44

# 7. Implementing CPS using well-order types

We introduced the $W$ types in Chapter 2. The $W$ types provide us with a way to encode all the inductively defined types, and we showed how we could use them to represent binary trees. The non-canonical constant associated with the $W$ types, wrec allows us to encode structural induction for the inductively defined types. In this chapter we will provide a non-canonical constant, wtail, which allows us to encode tail-recursion for the inductively defined types.

Rule 2.62 is the $W$ introduction rule.

## 7.1 Computation rule

The computation rule for wtail looks rather different from the computation rules for the CPS non-canonical constants associated with the inductively defined types. This is reasonable because the $W$ elimination rule corresponds to bar induction, whereas the elimination rules for the inductively defined types correspond to structural induction. As we can use wrec to implement any of the structurally recursive non-canonical constants, so wtail will allow us to implement any of the tail-recursive non-canonical constants. As we expect wtail takes an extra argument, the continuation function.

$$\frac{a \longrightarrow \mathsf{sup}(d, e) \qquad k \longrightarrow \lambda(f) \qquad b(d, e, (x.g)(\mathsf{wtail}(e(x), b, g)), f) \longrightarrow b'}{\mathsf{wtail}(a, b, k) \longrightarrow b'}$$

Rule 7.1: wtail computation

Using the definition of binary trees given in §2.5.1 we can use wtail to define treetail$(d, e, t, \lambda(f))$ as:

$$\begin{array}{l}
\mathsf{wtail}(t, \\
\qquad (x, y, z, f)\mathsf{when}((q)f(d), \\
\qquad\qquad\qquad\qquad (q)z(\mathsf{left}, \\
\qquad\qquad\qquad\qquad\qquad \lambda((u)(z(\mathsf{right}, \\
\qquad\qquad\qquad\qquad\qquad\quad \lambda((v)(f(e(q, y(\mathsf{left}), y(\mathsf{right}), v, u))))))), \\
\qquad\qquad\qquad\qquad x), \\
\qquad \lambda(f))
\end{array}$$

This mirrors the definition of treerec using wrec that we gave in §2.5.1.

## 7.2 Elimination rule

When we presented the CPS non-canonical constants for the inductively defined types discussed in Chapter 6 we found that the elimination rule that we constructed had as a conclusion:

$$\mathsf{nattail}(d, e, n, k) : D(\mathsf{natrec}(d, e, n))$$

and that we could justify a rule like Rule 6.37 which has as a conclusion:

$$\mathsf{nattail}(d, e, n, \lambda(f)) = \mathsf{apply}(\lambda(f), \mathsf{natrec}(d, e, n)) : D(\mathsf{natrec}(d, e, n))$$

We might hope to do the same for the $W$ types. Unfortunately the arities of the arguments to wtail and wrec are different. The second argument to wtail has arity:

$$\circ \otimes (\circ \ominus\!\!\!\!\rightarrow \circ) \otimes (\circ \otimes \circ \ominus\!\!\!\!\rightarrow \circ) \otimes (\circ \ominus\!\!\!\!\rightarrow \circ) \ominus\!\!\!\!\rightarrow \circ$$

and the second argument to wrec has arity:

$$0 \otimes (0 \ominus\!\!\!\longrightarrow 0) \otimes (0 \ominus\!\!\!\longrightarrow 0) \ominus\!\!\!\longrightarrow 0$$

However, if the second argument to wtail is called $b$ then we can define $b'$, of the correct arity, as:

$$b'(x, y, z) =_{\mathsf{def}} b(x, y, (u, v)z(u), (i)i)$$

We can think of $b'$ as an un-CPS version of $b$. $\mathsf{apply}(k, \mathsf{wrec}(a, b'))$ is then equal to $\mathsf{wtail}(a, b, k)$ and Rule 7.2 on the following page is an appropriate elimination rule. The justification for this rule follows the same pattern as the justifications that we presented for the rules in Chapter 6.

## 7.3 Chapter Summary

In this chapter we have provided a computation rule for wtail, a non-canonical constant for the $W$ types which will allow us to implement CPS non-canonical constants for the inductively defined types, in the same way as wrec allows us to construct structurally recursive non-canonical constants. We have given a definition of treetail using wtail, and the encoding of trees that we used in §2.5.1. Finally, we have presented an elimination rule which uses wtail.

$$\frac{a : W(A,B) \qquad \begin{array}{c}[v : W(A,B)]\\ \vdots \\ C(v)\ \mathsf{type}\end{array} \qquad \begin{array}{c}[y : C(a)]\\ \vdots \\ D(y)\ \mathsf{type}\end{array} \qquad k : \Pi(C(A), D) \qquad \begin{array}{c}\left[\begin{array}{c} y : A\\ z(x) : W(A,B)[x : B(y)]\\ u(w) : C(z(w))[w : B(y)]\end{array}\right]\\ \vdots \\ b'(y,z,u) : C(\mathsf{sup}(y,z))\end{array}}{\mathsf{wtail}(a,b,k) : D(\mathsf{wrec}(a,b'))}$$

Rule 7.2: $W$ elimination using wtail

# 8. Examples

We can now proceed to exploit tail recursion and program with continuations in a natural way within M-LTT. In this chapter we present some very simple functions to illustrate both the structure of the functions that we construct in continuation-passing style, and the structure of the proofs that we construct to use them. We focus on such simple functions because we are interested in the gross structure of the proofs, not in the details of any particular proof. We also outline a derivation of two simple programs.

## 8.1 Program structure

In §6.4 we saw that we could take a structurally-recursive function and produce an equivalent tail-recursive one. We look at some simple examples of this.

In M-LTT we can define addition and subtraction as:

$$\begin{aligned} \mathsf{plus}(n,m) &=_{\mathsf{def}} \mathsf{natrec}(m,(x,y)\mathsf{succ}(y),n) \\ \mathsf{times}(n,m) &=_{\mathsf{def}} \mathsf{natrec}(\mathsf{zero},(x,y)\mathsf{plus}(m,y),n) \end{aligned}$$

From these examples we can read off CPS versions:

$$\begin{aligned} \mathsf{plustail}(n,m,k) &=_{\mathsf{def}} \mathsf{nattail}(m,(x,y)\mathsf{succ}(y),n,k) \\ \mathsf{timestail}(n,m,k) &=_{\mathsf{def}} \mathsf{nattail}(\mathsf{zero},(x,y)\mathsf{plus}(m,y),n,k) \end{aligned}$$

We could, of, course use plustail in the definition of timestail:

$$\mathsf{timestail}(n,m,k) =_{\mathsf{def}} \mathsf{nattail}(\mathsf{zero},(x,y)\mathsf{plustail}(m,y,(\lambda(x)x)),n,k)$$

We can define the function to compute the length of a list as:

$$\mathsf{length}(l) =_{\mathsf{def}} \mathsf{listrec}(\mathsf{zero}, (x, y, z)\mathsf{succ}(z), l)$$

Recall that, when we use M-LTT for program derivation we are typically trying to construct a program p, such that we can make a judgement of the form:

$$\mathsf{p} : \mathsf{Specification}$$

Clearly in any real derivation Specification will be a rather complicated expression. Even in the very simple case of sorting a list (as discussed in § 8.3 on page 171) the type of the specification must state that the output list is an ordered permutation of the input list. Thus it must also tell us what a permutation is, and explain what it means for a list to be ordered. As we are only attempting to highlight some general properties of using CPS we will be in danger of not seeing the wood for the trees, so we focus on a very simple judgement, the judgement that $\mathsf{length}(l) : \mathsf{Nat}$ if $l : \mathsf{List}(A)$. The proof is:

$$\dfrac{\dfrac{l : \mathsf{List}(A) \quad \dfrac{}{\mathsf{Nat\ type}}\,\mathrm{Nat\ F} \quad \dfrac{}{\mathsf{zero} : \mathsf{Nat}}\,\mathrm{Nat\ I} \quad \dfrac{[n : \mathsf{Nat}]}{\mathsf{succ}(n) : \mathsf{Nat}}\,\mathrm{Nat\ I}}{\mathsf{listrec}(\mathsf{zero}, (x, y, z)\mathsf{succ}(z), l) : \mathsf{Nat}}\,\mathrm{List\ E}}{\mathsf{length}(l) : \mathsf{Nat}}\,=_{\mathsf{def}}$$

Rule 8.1: Typing $\mathsf{length}(l)$

Now we define a CPS function $\mathsf{lengthtail}(d, e, l, k)$:

$$\mathsf{lengthtail}(l, k) =_{\mathsf{def}} \mathsf{listtail}(\mathsf{zero}, (x, y, z)\mathsf{succ}(z), l, k)$$

The equivalent of Rule 8.1 is:

$$\dfrac{\dfrac{l : \mathsf{List}(A) \qquad \dfrac{}{\mathsf{Nat\ type}} \qquad k : \Pi(\mathsf{Nat}, D) \qquad \dfrac{}{\mathsf{zero} : \mathsf{Nat}} \qquad \dfrac{[n : \mathsf{Nat}]}{\mathsf{succ}(n) : \mathsf{Nat}}}{\mathsf{listtail}(\mathsf{zero}, (x, y, z)\mathsf{succ}(z), l, k) : D(\mathsf{listtail}(\mathsf{zero}, (x, y, z)\mathsf{succ}(z), l, \lambda((w)w)))}}{\mathsf{lengthtail}(l, k) : D(\mathsf{lengthtail}(l, \lambda((w)w)))}$$

Rule 8.2: Typing $\mathsf{lengthtail}(l, k)$

The two judgements involved in these proofs are slightly different, of course. Notice that in Rule 8.2 we have judgement about the type of the continuation left to establish. In this way we have moved the termination proof of $\mathsf{lengthtail}(l, k)$ onto whatever comes next. This is a clue to how CPS functions let us use M-LTT conveniently to discuss functions whose termination we are not really concerned about. For example, when were are performing I/O we usually want to show that the current I/O operation will pass control on to the next operation, and we are not *usually* concerned with whether the *total* sequence of I/O operations will terminate. Operating systems are similar. Although, in general, we want to be able to shut an operating system down gracefully, and therefore we *must* be able to discuss its termination properties, we normally think of an operating system as an on-going process. We usually want to discuss the local behaviour of a part of the operating system, and to show that, within the context of the operating system, the part behaves as we wish. M-LTT is not, with its strong emphasis on termination, the most natural setting in which to discuss on-going computations, but we have shown that we can use CPS functions to enable us to do this.

## 8.2 Proof structure

This phenomenon of passing the termination proof on to the next function also affects the structure of proofs that we construct. Again we stick with a very simple problem. We can define a function to add the lengths of two lists as:

$$\begin{aligned}\mathsf{ls}(l,m) &=_{\mathsf{def}} \mathsf{plus}(\mathsf{length}(l),\mathsf{length}(m))\\ &=_{\mathsf{def}} \mathsf{natrec}(\mathsf{length}(l),(x,y)\mathsf{succ}(y),\mathsf{length}(m))\end{aligned}$$

Showing that $\mathsf{ls}(l,m)$ has type $\mathsf{Nat}$ if $l,m$ are lists starts off as:

$$\cfrac{\cfrac{\cfrac{\begin{array}{c}\vdots\\ \mathsf{length}(m):\mathsf{Nat}\end{array} \qquad \cfrac{}{\mathsf{Nat\ type}} \qquad \begin{array}{c}\vdots\\ \mathsf{length}(m):\mathsf{Nat}\end{array} \qquad \cfrac{[y:\mathsf{Nat}]}{(x,y)\mathsf{succ}(y):\mathsf{Nat}}}{\mathsf{natrec}(\mathsf{length}(l),(x,y)\mathsf{succ}(y),\mathsf{length}(m)):\mathsf{Nat}}}{\mathsf{plus}(\mathsf{length}(l),\mathsf{length}(m)):\mathsf{Nat}}}{\mathsf{ls}(l,m):\mathsf{Nat}}$$

Rule 8.3: Typing $\mathsf{ls}(l,m)$

To use the CPS non-canonical constants to add the lengths of two lists we define:

$$\mathsf{lstail}(l,m,k) =_{\mathsf{def}} \mathsf{lengthtail}(m,\lambda((x)(\mathsf{lengthtail}(l,\lambda((y)(\mathsf{plustail}(m,l,k)))))))$$

Rule 8.5 on page 176 is the beginning of the equivalent derivation to Rule 8.3. These two proofs have rather different structures, reflecting the difference in the computation rules. Although we can think of the order imposed by Rule 8.5 as a restriction, we can also think of this as a way to assist in the structuring of proofs.

## 8.3 Example of program derivation

In this section we *outline* the derivation of two CPS programs:

- an algorithm to sort a list of Nat;
- an algorithm to find the least item in a list of Nat.

### 8.3.1 A CPS sorting algorithm

First, we will explain what a non-CPS sorting algorithm will look like, and then we will explain what the CPS version is.

#### Types for a non-CPS sorting algorithm

A non-CPS sorting algorithm will take a list of Nat and will return an ordered permutation of this list (along with a proof that this is an ordered permutation of the input list) .

One list of natural numbers is a permutation of another if all the numbers which occur in one occur the same number of times in the other. So we can define:

$$\mathsf{Perm}(l, m) =_{\mathsf{def}} \Pi(\mathsf{Nat}, (x)(\mathsf{EQ}(\mathsf{occurrences}(x, l), \mathsf{occurrences}(x, m), \mathsf{Nat})))$$

A list is ordered if the $i$th member is less than or equal to the $j$th when $i$ is less than or equal to $j$. So we can define:

$$\mathsf{Ordered}(l) =_{\mathsf{def}} \Pi(\mathsf{Nat}, (x)(\Pi(\mathsf{Nat}, (y)(\mathsf{index}(x, l) \leq \mathsf{index}(y, l) \supset x \leq y))))$$

One natural number is less than or equal to another if there is some natural number which can be added to the first to give the second. So we can define:

$$x \leq y =_{\mathsf{def}} \Sigma(\mathsf{Nat}, (z)(\mathsf{EQ}(\mathsf{plus}(x, z), y, \mathsf{Nat})))$$

As is usual, from a programming point of view, we are not interested in the formal proof that we have actually constructed a sorted permutation. For expository purposes we will suppress these details. We also suppress the definitions of occurrences and index.

We can define:

$$\mathsf{Sorted}(l) =_{\mathsf{def}} \Sigma(\mathsf{List}(\mathsf{Nat}), (m)(\mathsf{Perm}(l, m) \,\&\, \mathsf{Ordered}(m)))$$

So, a suitable type will be:

$$\Pi(\mathsf{List}(\mathsf{Nat}), \mathsf{Sorted})$$

If we were to construct a non-CPS program $p$ to sort a list we would construct some $p$ with this type.

### Using CPS

We are aiming to produce a CPS program, so we are actually thinking about producing a program which satisfies *this* specification:

$$\Pi(\mathsf{List}(\mathsf{Nat}), (l)\Pi(\Pi(\mathsf{Sorted}(l), D), D))$$

This specification may strike us as rather odd: in particular Sorted seems to be in the wrong place. We expect it to be the *implicate* rather than the *implicans*. In § 8.3.2 on the next page we will resolve this.

The search for an object with this type proceeds by using Π introduction twice, and our new list elimination rule to get (as usual suppressing the well-formedness judgements):

$$\dfrac{[l : \mathsf{List}(\mathsf{Nat})] \qquad [k : \Pi(\mathsf{Sorted}(l), D)] \qquad d : \mathsf{Sorted}(\mathsf{nil}) \qquad \begin{matrix} \begin{bmatrix} p : \mathsf{Nat} \\ q : \mathsf{List}(\mathsf{Nat}) \\ r : \mathsf{Sorted}(q) \end{bmatrix} \\ \vdots \\ e(p, q, r) : \mathsf{Sorted}(\mathsf{cons}(p, q)) \end{matrix}}{\lambda((l)\lambda((k)\mathsf{listtail}(d, e, l, k))) : \Pi(\mathsf{List}(\mathsf{Nat}), \Pi(\Pi(\mathsf{Sorted}, D), D))}$$

Rule 8.4: Sorting a list

The details of the rest of the proof proceeds just as in the non-CPS case.

### 8.3.2 Finding a minimum

A function to find the minimum in a list of Nat will take a list of Nat and will return either an indication that the list is empty, or will return the least item in the list (along with a proof that this is the least item in the list).

We can define:

$$\mathsf{Empty}(l) =_{\mathsf{def}} \neg\Sigma(\mathsf{Nat}, (m)(\mathsf{member}(m, l)))$$

and:

$$\mathsf{Least}(l, m) =_{\mathsf{def}} \mathsf{member}(m, l) \,\&\, \Pi(\mathsf{Nat}, (n)(\mathsf{member}(n, l) \supset m \leq n))))$$

We can define:

$$\mathsf{Min}(l) =_{\mathsf{def}} \mathsf{Empty}(l) + \Sigma(\mathsf{Nat}, \mathsf{Least}(l))$$

Now we are trying to find an object with type:

$$\Pi(\mathsf{List}(\mathsf{Nat}), \mathsf{Min})$$

In §8.3.1 we have outlined the construction of a function which takes a continuation which makes use of a sorted list. One of the properties which we can prove of sorted lists of natural numbers is that they are either empty, or that their head is their least item. A formal proof object can therefore be supplied as a continuation to our sorting algorithm to yield an algorithm to find a minimum. This also resolves our puzzle about the location of Sorted: when we supply a continuation the implicans will depend on a value of the type $\mathsf{Sorted}(l)$. In the trivial case the continuation will be the identity function of type $\mathsf{Sorted}(l) \to \mathsf{Sorted}(l)$, and in the case of an algorithm to find a minimum the continuation will be of type $\Pi(\mathsf{Sorted}(l), \mathsf{Min})$.

## 8.4 Chapter Summary

In this chapter we have given a simple examples which show how the use of the CPS non-canonical constants affects both the functions that we write in M-LTT, and the proofs that we construct. We have also shown how the use of continuations allows us to structure proofs, so that we can discuss local properties of sub-parts of larger programs, particularly programs whose global properties are not of interest when considering local fragments.

$$\dfrac{\dfrac{\dfrac{m : \mathsf{List}(A) \qquad \dfrac{}{\mathsf{Nat\ type}} \qquad \begin{matrix}\vdots\\ \lambda((x)(\mathsf{lengthtail}(l, \lambda((y)(\mathsf{plustail}(x, y, k)))))) : \Pi(\mathsf{Nat}, D)\end{matrix} \qquad \dfrac{}{\mathsf{zero} : \mathsf{Nat}} \qquad \dfrac{[n : \mathsf{Nat}]}{\mathsf{succ}(n) : \mathsf{Nat}}}{\mathsf{listtail}(\mathsf{zero}, (x, y, z)\mathsf{succ}(z), m, \lambda((x)(\mathsf{lengthtail}(l, \lambda((y)(\mathsf{plustail}(x, y, k))))))) : D(\mathsf{lstail}(l, m, \lambda((w)w)))}}{\mathsf{lengthtail}(m, \lambda((x)(\mathsf{lengthtail}(l, \lambda((y)(\mathsf{plustail}(x, y, k))))))) : D(\mathsf{lstail}(l, m, \lambda((w)w)))}}{\mathsf{lstail}(l, m, k) : D(\mathsf{lstail}(l, m, \lambda((w)w)))}$$

Rule 8.5: Typing $\mathsf{lstail}(l, m, k)$

# Part IV

# Conclusions

# 9. Conclusions and future work

In this chapter we outline some areas for possible future development, and draw some conclusions.

## 9.1 Topics for further investigation

The work presented in Chapter 4 on using continuations in categorial grammar is worth further development. The interpretation of type-lifting as continuation-passing is, to our knowledge, quite novel. Further investigations in this area should prove fruitful.

Only small examples were presented: developing larger programming examples would prove a useful exercise.

Continuations provide us with a way to discuss continuing computations. One of the great strengths of M-LTT is that typing implies termination, so even allowing for lazy evaluation, continuing computations are not really one of the strengths of M-LTT. We saw in Chapter 8 that we can use continuations to reason about continuing computations. Here we make a subtle distinction between constructive and intuitionistic mathematics. In constructive mathematics everything is lawlike: intuitionistic mathematics also allows lawless objects [112]. When we briefly discussed I/O in §3.3.1 we suggested a model where the world was just two lists. If we think instead of the input as being generated by a process, as it might be in a concurrent system, where the 'world' is the other processes, then we could think of the input as being generated, not by a lawlike sequence, but by, for example, a toss of a coin or the roll of a die. Thus a model of I/O can be the basis for a model of choice sequences. It would be interesting to see if this observation can be extended.

Despite earlier predictions state has not 'withered away' in functional programming. Haskell, however, uses monads to handle many state-based computations (see, for example [117]). Investigating the addition of monads to M-LTT might also be an interesting challenge.

Programming in M-LTT 'by hand' can be tedious. Adding CPS non-canonical constants to a proof-assistant would help to alleviate this problem.

## 9.2 Summary

We presented M-LTT, paying particular attention to the proof-theoretic arguments presented by Michael Dummett on how inference rules are (or should be) justified. We did this because we intended to present an extension to M-LTT which would allow us to program with continuations, and which would require us to construct elimination rules of an unusual form.

We gave an description of the continuation-passing style of functional programming and highlighted some desirable properties of CPS functions. We illustrated an interpretation of type-lifting in categorial grammar as continuation passing.

We showed that we can extend M-LTT to allow us to program with continuations. We presented CPS non-canonical constants for types which are commonly used in programming, and presented and justified rules for using these CPS non-canonical constants. We showed how the new constants related to the usual non-canonical constants for the types involved, and consequently showed that we had not introduced any new absurdities into the theory. We also presented a CPS non-canonical constant for the $W$ types, and showed that this allows us to express the CPS non-canonical constant for binary trees.

Finally we presented some very simple examples.

## 9.3 Conclusions

Our first conclusion is that it *is* possible to extend M-LTT to allow ourselves to program with continuations in a natural way. We can present CPS non-canonical constants which follow a uniform pattern for the inductively defined types.

We also conclude that the informal semantics of M-LTT can be a double-edged sword. By following the informal semantics carefully we can extend the theory in various ways, however the informality of the semantics can act against us believing that our extension is *really* justifiable.

As a programming environment M-LTT suffers from the relative poverty of the programs we can write. Effort spent on the left of the : is as valuable as that spent to the right.

# Bibliography

[1] Ackermann, W. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928. In German.

[2] Ajdukiewicz, K. Syntactic Connexion. In S. McCall, editor, *Polish Logic 1920–1939*, pages 207–231. Clarendon Press, Oxford, England, 1967. First appeared as Die syntaktische Konnexität, *Studia Philosophica*, vol. 1, 1–27, 1935. Original in German.

[3] Appel, A. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, 1992.

[4] Backhouse, R., P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself Type Theory. Computing Science Notes CS 8811, Wiskunde en Informatica, Rijksuniversiteit Groningen, 19 September 1988.

[5] Bar-Hillel, Y. On Categorial and Phrase Structure Grammars. In *Language and Information Selected Essays on their Theory and Application*, pages 99–115. 1964. First appeared in *The Bulletin of the Research Council of Israel*, vol. 9F, 1–16, 1960.

[6] Barendregt, H. *The Lambda Calculus Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, The Netherlands, revised edition, 1984.

[7] —. The Impact of the Lambda Calculus in Logic and Computer Science. *Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[8] Barwise, J. *The Situation in Logic*, volume 17 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford, California, USA, 1989.

[9] Barwise, J. and J. Perry. *Situations and Attitudes.* MIT Press, Cambridge, Massachusetts, USA, 1983.

[10] Beeson, M. J. *Foundations of Constructive Mathematics.* Springer, Berlin, Germany, 1985.

[11] van Benthem, J. and K. Does. Higher-order Logic. In Gabbay and Guenthner [40], pages 275–329.

[12] Bishop, E. *Foundations of Constructive Analysis.* McGraw-Hill, New York, New York, USA, 1967.

[13] Bridges, D. and F. Richman. *Varieties of Constructive Mathematics*, volume 97 of *London Mathematical Society Lecture Note Series.* Cambridge University Press, Cambridge, England, 1987.

[14] Brouwer, L. E. J. *Brouwer's Cambridge Lectures on Intuitionism.* Cambridge University Press, Cambridge, England, 1981. Edited by Dirk van Dalen.

[15] Carpenter, B. *Type-Logical Semantics.* The MIT Press, Cambridge, Massachusetts, USA, 1997.

[16] Church, A. *Introduction to Mathematical Logic*, volume 1. Princeton University Press, Princeton, New Jersey, USA, second, enlarged edition, 1956.

[17] Coquand, T., B. Nordström, J. M. Smith, and B. von Sydow. Type Theory and Programming. *EATCS Bulletin*, 52, 1994.

[18] Danvy, O. and A. Filinski. Representing Control A Study of CPS Transformation. *Mathematical Structures in Computer Science*, 1992. Also Tech Report CIS-91-2, Kansas State University.

[19] Davie, A. J. T. *An Introduction to Functional Programming Systems Using Haskell*, volume 27 of *Cambridge Computer Science Texts*. Cambridge University Press, Cambridge, England, 1992.

[20] Dijkstra, E. W. *A Primer of ALGOL 60 Programming*. APIC Studies in Data Processing. Academic Press, London, England, 1962.

[21] —. *A Discipline of Programming*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1976.

[22] Dowty, D. R. Type Raising, Functional Composition, and Non-Constituent Conjunction. In R. T. Oehrle, E. Bach, and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures*, Studies in Linguistics and Philosophy, pages 153–198. D. Reidel, Dordrecht, The Netherlands, 1988.

[23] Dowty, D. R., R. E. Wall, and S. Peters. *Introduction to Montague Semantics*. Studies in Linguistics and Philosophy. D. Reidel, Dordrecht, The Netherlands, 1981.

[24] Draves, R. P., B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *13th ACM Symposium on Operating Systems Principles*, pages 122–136. ACM Press, 1991.

[25] Dummett, M. *Elements of Intuitionism*. Oxford Logic Guides. Clarendon Press, Oxford, England, 1977. A second edition was published in 2000.

[26] —. The Justification of Deduction. In *Truth and Other Enigmas* [28], pages 290–318. First published in 1973.

[27] —. The Philosophical Basis of Intuitionistic Logic. In *Truth and Other Enigmas* [28], pages 215–247. First published in H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, North-Holland, Amsterdam, The Netherlands,1975.

[28] —. *Truth and Other Enigmas*. Duckworth, London, England, 1978.

[29] —. *The Interpretation of Frege's Philosophy*. Duckworth, London, England, 1981.

[30] —. *The Logical Basis of Metaphysics*. Duckworth, London, England, 1991.

[31] —. *The Seas of Language*. Oxford University Press, Oxford, England, 1993.

[32] —. What is a theory of meaning? (I). In *The Seas of Language* [31], pages 1–33. First published in S. Guttenplan, editor, *Mind and Language*, Oxford University Press, Oxford, England, 1975.

[33] —. What is a theory of meaning? (II). In *The Seas of Language* [31], pages 34–105. First published in G. Evans and J. McDowell, editors, *Truth and Meaning*, Oxford University Press, Oxford, England, 1976.

[34] van Eijck, J. and F.-J. de Vries. Reasoning About Update Logic. *Journal of Philosophical Logic*, 24(1):19–47, 1995.

[35] Fischer, M. J. Lambda Calculus Schemata. *Sigplan Notices*, 7:104–109, 1972.

[36] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *Conference on Programming Language Design and Implementation*. 1993.

[37] Frege, G. *The Foundations of Arithmetic*. Basil Blackwell, Oxford, England, 1950. Originally published in German as *Die Grundlagen der Arithmetik*, 1884.

[38] —. Function and Concept. In P. Geach and M. Black, editors, *Translations from the Philosophical Writings of Gottlob Frege*, pages 42–55. Barnes and Noble, Totawa, New Jersey, USA, 3rd edition, 1952. Translation of an address given to the *Jenaische Gessellschaft für Medicin und Naturwissenschaft*, 1891.

[39] Friedman, H. Classically and Intuitionistically Provably Recursive Functions. In G. Müller and D. S. Scott, editors, *Higher Set Theory*, pages 21–27. Springer, 1977.

[40] Gabbay, D. and F. Guenthner, editors. *Handbook of Philosophical Logic. Volume I. Elements of Classical Logic*, volume 164 of *Synthese Library*. D. Reidel, Dordrecht, The Netherlands, 1983.

[41] Geach, P. T. *Reference and Generality An Examination of Some Medieval and Modern Theories*. Cornell University Press, Ithaca, New York, USA, 3rd edition, 1980. First edition published in 1962.

[42] Gentzen, G. Investigations Into Logical Deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, The Netherlands, 1969.

[43] Girard, J.-Y. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des copures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Scandinavian Logic Symposium*, pages 63–69. North-Holland, Amsterdam, The Netherlands, 1971. In French.

[44] —. *Interprétation fonctionelle et élimination des coupures de l'arithmétic d'ordre supérieur*. Thèse de doctorat d'état, Université Paris VII, 1972. In French.

[45] —. The system **F** of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[46] —. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[47] —. *Proof Theory and Logical Complexity*, volume 1 of *Studies in Proof Theory Monographs*. Bibliopolis, Napoli, Italy, 1987.

[48] Girard, J.-Y., Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1989.

[49] Green, J. S. *ALGOL Programming for KDF9.* An English Electric Leo mini-manual. English Electric-Leo Computers Ltd, Stoke-on-Trent, England, 1963.

[50] Griffin, T. G. A Formulae-as-Types Notion of Control. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL 17)*, pages 47–58. ACM Press, 1990.

[51] de Groote, P. *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de Logique*. Academia, Louvain-la-Neuve, Belgium, 1995.

[52] Hamilton, A. G. The Pict System. Department of Computer Science Technical Report 57, Stirling University, 1990.

[53] Harel, D. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic II Extension of Classical Logic*, volume 164 of *Synthese Library*, pages 497–604. D. Reidel, Dordrecht, The Netherlands, 1984.

[54] Hatcliff, J. and O. Danvy. A Generic Account of Continuation-Passing Styles. In *Popl 94 : ACM Symposium on Principles of Programming Languages*, pages 458–471. Association for Computing Machinery, 1994.

[55] Hayes, I. J. and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989. Also available as Technical Report 148, Key Centre for Software Technology, Department of Computer Science, University of Queensland.

[56] Hilbert, D. and W. Ackermann. *Principles of Mathematical Logic.* Chelsea Publishing Company, New York, USA, 1950. A translation, with corrections, of the second German edition of *Grundzüge der Theoretishcen logik* of 1938.

[57] Hilbert, D. and P. Bernays. *Grundlagen der Mathematik*, volume 1. Springer, Berlin, Germany, 1934. In German.

[58] Horn, C. The NuPRL Proof Development System. Department of Artificial Intelligence Working Paper 214, University of Edinburgh, 1988. The Edinburgh version of NuPRL has been renamed Oyster.

[59] Kamp, H. and U. Reyle. *From Discourse to Logic Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*, volume 42 of *Studies in Logic and Philosophy*. Kluwer, Dordrecht, The Netherlands, 1993.

[60] —. A Calculus for First Order Discourse Representation Structures. *Journal of Logic, Language, and Information*, 5(3–4):297–348, 1996.

[61] Kelsey, R., W. Clinger, and J. Rees. Revised$^5$ Report on the Algorithmic Language Scheme. *Journal of Higher Order and Symbolic Computation*, 11(1):7 –105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[62] Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2nd edition, 1988.

[63] Kleene, S. C. General Recursive Functions of Natural Numbers. In M. Davis, editor, *The Undecidable*, pages 237–253. Raven Press, Hewlett, New York, USA, 1965. Originally published in *Mathematische Annalen* 112(5):727–742, 1936.

[64] Kolmogorov, A. On the Interpretation of Intuitionistic Logic. In P. Mancosu, editor, *From Brouwer to Hilbert The Debate on the Foundations of Mathematics in the 1920s*, pages 324–334. Oxford University Press, Oxford, England, 1998. Originally published in German as Zur Deutung der intuitionistichen Logik, *Mathematische Zeitschrift* 35:58–65, 1932.

[65] König, E. Parsing as natural deduction. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 272–279. ACL Publications, Morristown, New Jersey, 1989.

[66] Kreisel, G. Informal rigour and completeness proofs. In I. Lakatos, editor, *Problems in the philosophy of mathematics. Proceedings of the International Colloquium in the Philosophy of Science, Volume 1*, pages 138–186. North-Holland, Amsterdam, The Netherlands, 1965.

[67] Lambek, J. The Mathematics of Sentence Structure. *The American Mathematical Monthly*, 65:154–170, 1958. Also published in W. Buszkowski, W. Marciszewski, and J. van Benthem, editors, *Categorial Grammar*, John Benjamin, Amsterdam, The Netherlands, 1988.

[68] Leonardo of Pisa (Fibonacci). *Liber Abacci.* 2nd edition, 1228. In Latin.

[69] Leslie, N. *Specification And Implementation Of A Unification Algorithm In Martin-Löf's Type Theory.* MSc, St Andrews, 1993.

[70] —. Tail-recursive Non-canonical Constants and Continuations in Martin-Löf's Type Theory. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop & Formal Methods Pacific '98*, pages 213–229. Springer Singapore, 1998.

[71] Louden, K. C. *Programming Languages Principles and Practice.* PWS-KENT Series in Computer Science. PWS Publishing Company, Boston, Massachusetts, USA, 1993.

[72] Luo, Z. and R. Pollack. The LEGO Proof Development System: A User's Manual. Technical Report ECS-LFCS-92-211, Edinburgh University, 1992.

[73] McCarthy, J. History of LISP. In Wexelblat [118], pages 173–197.

[74] Martin-Löf, P. A theory of types. Technical report 71-3, Mathematics Department, University of Stockholm, 1971.

[75] —. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics, pages 81–109. North-Holland, 1975.

[76] —. Constructive Mathematics and Computer Programming. In L. J. Cohen, J. Łoś, H. Pfeiffer, and K.-P. Podewski, editors, *Logic, methodology, and philosophy of science VI : proceedings of the Sixth International Congress of Logic, Methodology, and Philosophy of Science*, Studies in Logic and the Foundations of Mathematics, pages 153–175. North-Holland, Amsterdam, The Netherlands, 1982.

[77] —. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory Lecture Notes*. Bibliopolis, Napoli, Italy, 1984. Notes taken by Giovanni Sambin from a series of lectures given in Padua, June 1980.

[78] —. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. Originally presented at the meeting *Teoria della Dimostrazione e Filosofia della Logica*, Siena 6–9 April, 1983.

[79] —. An intuitionistic theory of types. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. Clarendon Press, Oxford, England, 1998. Originally circulated as a University of Stockholm Mathematics Department pre-print, 1972.

[80] Montague, R. *Formal Philosophy Selected Papers of Richard Montague*. Yale University Press, New Haven, Connecticut, USA, 1974. Edited and with an introduction by Richmond H. Tomason.

[81] —. The Proper Treatment of Quantification in Ordinary English. In *Formal Philosophy Selected Papers of Richard Montague* [80], pages 247–270. Originally published in J. Hintikka, J. Moravcsik and P. Suppes (eds.) *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*. D. Reidel Publishing Company. Dordrecht, The Netherlands. 1970.

[82] Moortgat, M. *Categorial Investigations Logical and Linguistic Aspects of the Lambek Calculus*, volume 9 of *Groningen-Amsterdam Studies in Semantics*. Foris Publications, Dordrecht, The Netherlands, 1988.

[83] Morrill, G. V. *Type Logical Grammar*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1994.

[84] —. Grammar and logic. *Theoria*, LXII:260–293, 1996. An earlier version appeared as *Grammar and Logical Types* in Guy Barry and Glyn V Morrill, editors, *Studies in Categorial Grammar*, pages 127–148, Edinburgh Working Papers in Cognitive Science Volume 5, Centre for Cognitive Science, University of Edinburgh, 1990; and in Martin Stokhof and Leen Torenvliet, editors, *Proceedings of the Seventh Amsterdam Colloquium*, pages 429-450, Institute for Language, Logic and Information, Universiteit van Amsterdam.

[85] Murthy, C. R. *Extracting Constructive Content From Classical Proofs*. PhD, Cornell University, 1990. Also available as TR 89-1151 from Dept. of Computer Science, Cornell University.

[86] —. Classical Proofs as Programs. How, What, When and Why. Extended abstract, Department of Computer Science, Cornell University, Feb 18 1991.

[87] Nordström, B. Terminating General Recursion. *BIT*, 28:605–619, 1988.

[88] Nordström, B., K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory An Introduction*. Clarendon Press, Oxford, England, 1990.

[89] Paulson, L. C. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2, 1986. Also appeared as Technical Report 57, University of Cambridge Computer Laboratory, 1984.

[90] Perlis, A. J. The American Side of the Development of ALGOL. In Wexelblat [118], pages 75–91.

[91] Petersson, K. A Programming System for Type Theory. Programming Methodology Group Report 9, Chalmers University of Technology, 1982.

[92] Peyton Jones, S. L. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series on Computer Science. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1987.

[93] Peyton Jones, S. L., J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. The Haskell 98 Report, 1999. Available from `http://www.haskell.org/onlinereport/`.

[94] Plotkin, G. D. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[95] Prawitz, D. *Natural Deduction A Proof-Theoretical Study*, volume 3 of *Stockholm Studies in Philosophy*. Almquist and Wiksells, Uppsala, Sweden, 1965.

[96] —. Meaning and proofs: on the conflict between classical and intuitionistic logic. *Theoria*, 77(1):1–40, 1977.

[97] Prior, A. N. The runabout inference-ticket. *Analysis*, 21:38–39, 1960.

[98] Ranta, A. *Type-Theoretical Grammar*, volume 1 of *Indices*. Clarendon Press, Oxford, England, 1994.

[99] Restall, G. *An Introduction to Substructural Logics*. Routledge, London, England, 2000.

[100] Russell, B. *The Principles of Mathematics*. Cambridge University Press., Cambridge, England, 1903. A re-issued edition was published in 1996 by WW Norton and Co.

[101] Rybakov, V. V. *Admissibility of Logical Inference Rules*, volume 136 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, The Netherlands, 1997.

[102] Schmidt, D. A. *Denotational Semantics A Methodology for Language Development*. Wm. C. Brown, Dubuque, Iowa, USA, 1986.

[103] Schwichtenberg, H. Proofs, Lambda Terms and Control Operators. In H. Schwichtenberg, editor, *Logic of Computation*, pages 309–348. Springer, Heidelberg, Germany, 1997. Proceedings of the NATO Advanced Study Institute on Logic of Computation, held at Marktoberdorf, Germany, July 25 – August 6, 1995.

[104] Shieh, S. Some Senses of Holism: An Anti-Realist's Guide to Quine. In R. G. Heck, Jnr, editor, *Language Thought and Logic Essays in Honour of Michael Dummett*, pages 71–103. Oxford University Press, Oxford, England, 1997.

[105] Strachey, C. and C. P. Wadsworth. Continuations A Mathematical Semantics for Handling Full Jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory Programming Research Group, 1974.

[106] Sundholm, G. Proof Theory and Meaning. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic. Volume III. Alternatives to Classical Logic*, volume 166 of *Synthese Library*, pages 471–506. D. Reidel, Dordrecht, The Netherlands, 1986.

[107] —. Systems of Deduction. In Gabbay and Guenthner [40], pages 133–188.

[108] Tarski, A. The Semantic Conception of Truth and the Foundation of Semantics. *Philosophy and Phenomenological Research*, 4, 1944. This paper has appeared in a number of other places (sometimes under the truncated title *The Semantic Conception of Truth*), including Volume II of *Collected papers of Alfred Tarski* S.R. Givant and R.N. McKenzie (eds), Birkhauser, Basel, Switzerland, 1986.

[109] Thielecke, H. *Categorical Structure of Continuation Passing Style*. PhD, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

[110] Thompson, S. *Type Theory and Functional Programming*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1991.

[111] —. *Haskell The Craft of Functional Programming*. International Computer Science Series. Addison-Wesley, Wokingham, England, 2nd edition, 1999.

[112] Troelstra, A. S. *Choice Sequences A Chapter of Intuitionistic Mathematics*. Oxford Logic Guides. Oxford University Press, Oxford, England, 1977.

[113] —. *Lectures on Linear Logic*, volume 29 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, California, USA, 1992.

[114] Troelstra, A. S. and D. van Dalen. *Constructivism in Mathematics An Introduction Volume 1*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, revised edition, 1988.

[115] —. *Constructivism in Mathematics An Introduction Volume 2*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, 1988.

[116] Vorobev, N. N. *The Fibonacci Numbers*. Heath, Boston, Massachusetts, USA, 1963. Translation of *Chisla Fibonachchi*, published in Russian, 1951.

[117] Wadler, P. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*.

Springer, Berlin, Germany, 1992. This paper is available from `http://www.cs.bell-labs.com/who/wadler/topics/monads.html`.

[118] Wexelblat, R. L., editor. *History of Programming Languages*. ACM Monograph. Academic Press, New York, New York, USA, 1981.

[119] Wittgenstein, L. *Philosophical Investigations*. Basil Blackwell, Oxford, England, 2nd edition, 1958. Translated by G. E. M. Anscombe.