Cristina Matache

Formalisation of the $\lambda\mu^{\mathbf{T}}$-calculus in Isabelle/HOL

Computer Science Tripos – Part II

Fitzwilliam College

May 16, 2017

Proforma

Name:	**Cristina Matache**
College:	**Fitzwilliam College**
Project Title:	**Formalisation of the $\lambda\mu^{\mathbf{T}}$-calculus in Isabelle/HOL**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**11,957**[1]
Project Originator:	Dr V. B. F. Gomes
Supervisor:	Dr V. B. F. Gomes

Original Aims of the Project

The project was concerned with the $\lambda\mu^{\mathbf{T}}$-calculus, an extension of the λ-calculus isomorphic to classical logic. The first main goal was to formalise the $\lambda\mu^{\mathbf{T}}$-calculus in the proof assistant Isabelle/HOL, and use this formalisation to prove type preservation and progress theorems for $\lambda\mu^{\mathbf{T}}$. The second main goal was to define a language, μ**ML**, based on the calculus, and implement an interpreter for it in OCaml. None of these have been explored in the literature before, although pen-and-paper proofs about metatheoretical properties of $\lambda\mu^{\mathbf{T}}$ exist.

Work Completed

Both main goals have been achieved. The $\lambda\mu^{\mathbf{T}}$ formalisation and proofs were completed using de Bruijn notation. The interpreter was implemented by extracting code from the Isabelle formalisation and writing a front-end in OCaml. As a result, the evaluation function of the interpreter has the desirable safety properties: type preservation and progress. Tests were carried out for the remaining components. Moreover, I extended the $\lambda\mu^{\mathbf{T}}$-calculus to $\lambda\mu^{\mathbf{T}}_{top}$ and added a range of new datatypes. The formalisation, proofs, and the μ**ML** interpreter were extended accordingly. Finally, I showed how classical propositions can be proved using the interpreter.

Special Difficulties

None.

[1]This word count was computed by `texcount -sum diss.tex`

Declaration

I, Cristina Matache of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr Victor Gomes, for proposing this project to me, and for patiently providing assistance and answering my many questions throughout the project. Moreover, I am grateful to him and to Dr Dominic Mulligan for their initiative to prepare a paper based on this work, *Programming with classical types*, and submit it to ITP. I appreciate the help of my Director of Studies, Dr Robert Harle, of Dr Tim Griffin, and of my friends Sebastian Borgeaud and George Ash, who read earlier versions of this dissertation and provided useful comments. Last but not least, I want to thank my parents for supporting me during my time in Cambridge, and for being an inspiration.

Chapter 1

Introduction

This chapter begins with an outline of the project. It presents the main motivation for undertaking this project, and gives a brief account of previous related work.

1.1 Project Structure

My project concerns the $\lambda\mu^{\mathbf{T}}$-calculus. The aim was to formalise $\lambda\mu^{\mathbf{T}}$ in the proof assistant Isabelle/HOL, and to prove Type Preservation and Progress for the formalised calculus. This is described in Section 3.1.

Additionally, the project involved building an interpreter in OCaml for a language based on $\lambda\mu^{\mathbf{T}}$, called μ**ML** (Section 3.3). The type-checking and evaluation functions used in the interpreter were automatically extracted from the Isabelle formalisation. This means that the Isabelle proofs still apply to them. Therefore, the core of the interpreter is proved correct: evaluation obeys type preservation and progress. A diagram of the project is shown in Figure 1.1.1.

These components constitute the core of the project, and have all been implemented successfully. As time allowed, I worked on several extensions. I used the μ**ML** interpreter to prove classical propositions, illustrated in Section 4.2. Moreover, I extended the $\lambda\mu^{\mathbf{T}}$-calculus to $\lambda\mu^{\mathbf{T}}_{top}$, and added booleans, products and sums (Section 3.2).

Given the work completed, the success criteria established in the project proposal have been met. Moreover, a paper based on this work has been submitted to the *International Conference on Interactive Theorem Proving (ITP)*, and is currently under review.

1.2 Motivation

This section briefly discusses the rationale behind creating the $\lambda\mu^{\mathbf{T}}$-calculus. It then outlines the reasons why my project is interesting, explains how the scope of the project was established, and motivates the choice of tools.

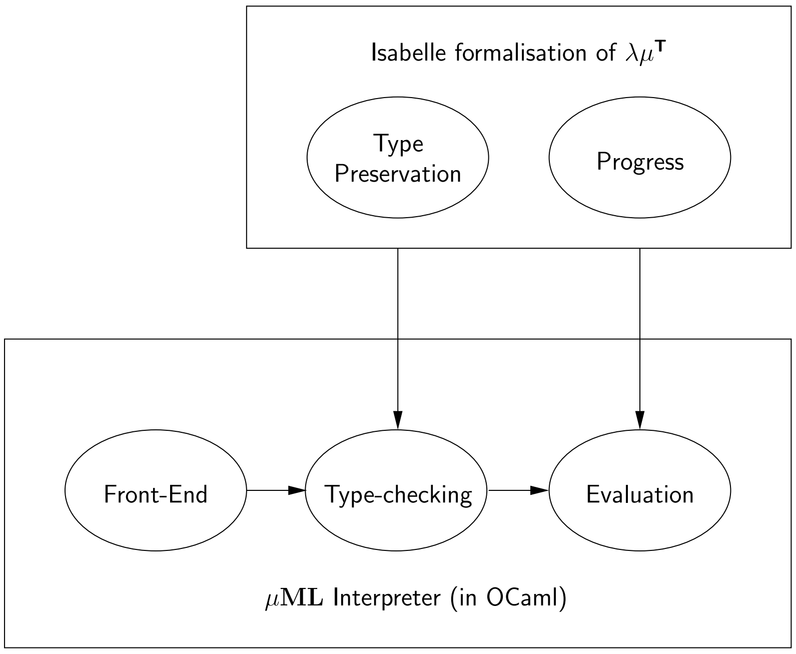

Isabelle formalisation of $\lambda\mu^{\mathbf{T}}$
Type Preservation
Progress
Front-End
Type-checking
Evaluation
μ**ML** Interpreter (in OCaml)

Figure 1.1.1: The structure of the project.

1.2.1 The $\lambda\mu^{\mathbf{T}}$-calculus

The $\lambda\mu^{\mathbf{T}}$-calculus was introduced by Geuvers *et al.* [14]. It is a combination between Parigot's $\lambda\mu$-calculus [30] and Gödel's System **T** [41].

The purpose of developing $\lambda\mu$ is to interpret the computational content of classical proofs. This is still an open area of research, which generated considerable interest. The $\lambda\mu$-calculus can be viewed as an extension of the λ-calculus with control operators. It is isomorphic to classical logic, under the Curry-Howard, or propositions-as-types, correspondence [19].

System **T** is an extension of the simply-typed λ-calculus, adding a datatype for natural numbers and primitive recursion on them. Gödel used it to prove consistency of arithmetic [41].

The reason for combining $\lambda\mu$ and System **T** is that Geuvers *et al.* [14] noticed a lack of calculi with control that also incorporate datatypes. They motivate this lack saying that, when combining control operators with datatypes, it is not straightforward to prove standard results such as confluence and strong normalisation. Consequently, they develop $\lambda\mu^{\mathbf{T}}$ and prove these results.

1.2.2 Contribution of my Project

The value of formally verifying mathematical or computer science theories is well-known. For example, in the paper *Winskel is (Almost) Right* [28], Nipkow describes a formalisation in Isabelle/HOL of the first 100 pages of Winskel's semantics textbook [43]. This formalisation uncovered one serious mistake in a proof found in the textbook.

The $\lambda\mu^{\mathbf{T}}$ definitions and proofs outlined by Geuvers *et al.* [14] are all done manually, apart from part of the strong normalisation proof. Therefore, a formalisation of $\lambda\mu^{\mathbf{T}}$ in a proof assistant would increase the confidence in the correctness of these proofs.

However, formalising theories in an interactive theorem prover takes a lot of time. The proofs are "extremely laborious" [35], which is probably why they are not more widespread. Consequently, I only chose to include Type Preservation and Progress proofs for $\lambda\mu^{\mathbf{T}}$ in the core of my project. Although in the beginning this seemed too little, they turned out to occupy most of the time spent on the project.

There are several reasons for this. I had never used a proof assistant before, so first, I had to learn Isabelle. Moreover, there is no previous formalisation of the $\lambda\mu^{\mathbf{T}}$-, or the $\lambda\mu$-calculus as far as I know. Type Preservation and Progress proofs for simple languages are described in the Part IB Semantics of Programming Languages course [40]. However, the particularities of $\lambda\mu^{\mathbf{T}}$ require additional niceties to make the standard proofs work. These were sometimes omitted in Geuvers *et al.* [14].

What is more, the original $\lambda\mu^{\mathbf{T}}$ formulation [14] does not use de Bruijn indices, and I am not aware of any other work that does. As a result, I had to adapt all the definitions and proofs myself to use de Bruijn indices. In practice, I had to prove a significant amount of additional lemmas to handle them.

The μ**ML** interpreter is of interest because μ**ML**, as $\lambda\mu^{\mathbf{T}}$, is isomorphic to classical logic. This means that classical propositions can be proved in μ**ML** by encoding them as a type, finding a term of that type, and type-checking it. This way of proving propositions is similar to how the proof assistants Agda [1] and Coq [5] work for constructive logic. There is no other implementation of a programming language based on the $\lambda\mu^{\mathbf{T}}$-calculus, as far as I know.

1.2.3 The Choice of Proof Assistant and Programming Language

I chose to use Isabelle/HOL mainly because I could get appropriate supervision for the project if I did so. As a beginner, learning Isabelle was made easier by the tutorials and up-to-date documentation that the Isabelle/HOL maintainers provide [18, 21, 26, 27, 42].

The Isabelle code generation facility [18] proved very useful in this project. This feature allows executable specifications, such as functions, to be exported from Isabelle theories into Standard ML, OCaml, Haskell or Scala. It allowed me to automatically generate code for the core of the μ**ML** interpreter, which is therefore verified. Among the languages that Isabelle exports code to, I am familiar with SML and OCaml, from the Part IA Foundations of Computer Science course [33], and the Part IB Compiler Construction course [17], respectively. I chose to use OCaml because it is more widely used for developing applications.

1.3 Related Work

Extracting computational content from classical proofs, and finding calculi that are isomorphic to classical logic, under the propositions-as-types correspondence, has attracted a great deal of research. This interest began with an influential paper by Griffin [16]. Here, he types the $\mathcal{C}$ operator, from Felleisen's λ-$\mathcal{C}$ [10], using the Double Negation Law,

$\neg\neg A \to A$. Thus, the propositions-as-types correspondence is extended to classical logic. The operator $\mathcal{C}$ is similar to `call/cc` in Scheme (call-with-current-continuation), so λ-$\mathcal{C}$ includes a notion of control.

Following this development, a series of other calculi with control have emerged. Some of them are Parigot's $\lambda\mu$ [30], the λ_Δ-calculus [39], and Crolard's $\lambda_{\mathtt{ct}}$ [7], a λ-calculus with a `catch/trow` mechanism. There has been considerable research into the metatheoretical properties of $\lambda\mu$, for example [9, 31, 32], and into different variants of $\lambda\mu$, for example [11, 29], to name a few. The paper by Ariola and Herbelin [6] is an excellent study of calculi with control and the logics that are isomorphic to them.

Chapter 2

Preparation

The main purpose of this chapter is to introduce the reader to the $\lambda\mu^{\mathbf{T}}$-calculus and to the proof assistant Isabelle/HOL. It also gives a brief account of the starting point of the project, the methodology followed, and the tools used.

2.1 Starting Point

To the best of my knowledge, there is no previous formalisation of the $\lambda\mu^{\mathbf{T}}$-calculus in any proof assistant. Similarly, I am not aware of any implementation of a language based on this calculus.

Prior to arranging this project, I had no experience working with proof assistants. I started learning Isabelle in October 2016, when it became apparent that I was going to use it for my project. For this purpose, I studied some of the material in the textbook *Concrete Semantics* [26], and attempted the associated exercises. While formalising the $\lambda\mu^{\mathbf{T}}$-calculus, it proved useful to refer to a similar formalisation of the λ-calculus that is part of the Isabelle distribution [2]. At the start of the project, my knowledge of OCaml was limited to that gained in the Part IB Compiler Construction course [17].

2.2 The $\lambda\mu^{\mathbf{T}}$-calculus

The $\lambda\mu^{\mathbf{T}}$-calculus is an extension of the simply-typed λ-calculus. It incorporates a new kind of abstraction (the μ-abstraction), a datatype of natural numbers, and a construct for primitive recursion on this datatype. It has been shown that $\lambda\mu^{\mathbf{T}}$ obeys type preservation, has a normal-form theorem, is confluent and strongly-normalising [14]. The calculus in presented in the next six subsections.

2.2.1 Syntax

The syntax of the $\lambda\mu^{\mathbf{T}}$-calculus is defined as:

Definition 2.2.1 (Syntax).

$$\begin{aligned} t, r, s &::= x \mid \lambda x : \rho.r \mid ts \mid \mu\alpha : \rho.c \mid 0 \mid \mathtt{S}t \mid \mathtt{nrec}_{\rho}\ r\ s\ t \\ c, d &::= [\alpha]t \end{aligned}$$

$$\frac{x : \rho \in \Gamma}{\Gamma; \Delta \vdash x : \rho} \qquad \frac{\Gamma, x : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda x : \sigma . t : \sigma \to \tau} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash ts : \tau}$$

(a) axiom (b) lambda (c) app

$$\Gamma; \Delta \vdash 0 : \mathtt{N} \qquad \frac{\Gamma; \Delta \vdash t : \mathtt{N}}{\Gamma; \Delta \vdash \mathtt{S}t : \mathtt{N}}$$

(d) zero (e) suc

$$\frac{\Gamma; \Delta \vdash r : \rho \qquad \Gamma; \Delta \vdash s : \mathtt{N} \to \rho \to \rho \qquad \Gamma; \Delta \vdash t : \mathtt{N}}{\Gamma; \Delta \vdash \mathtt{nrec}_\rho\ r\ s\ t : \rho}$$

(f) nrec

$$\frac{\Gamma; \Delta, \alpha : \rho \vdash c : \perp\!\!\!\perp}{\Gamma; \Delta \vdash \mu\alpha : \rho . c : \rho} \qquad \frac{\Gamma; \Delta \vdash t : \rho \qquad \alpha : \rho \in \Delta}{\Gamma; \Delta \vdash [\alpha]t : \perp\!\!\!\perp}$$

(g) activate (h) passivate

Figure 2.2.1: The rules for typing judgments in $\lambda\mu^{\mathbf{T}}$ [14].

Terms of the form $[\alpha]t$ are called *commands* or *named terms*, while all other terms are unnamed. There are two kinds of variables: the usual λ-variables, and μ-variables, denoted by lower-case Greek letters. A μ-variable labels, or *names*, the term inside a command. Similarly to the λ-abstraction, there is a μ-abstraction which binds μ-variables. The terms 0 and $\mathtt{S}t$ are used to represent the natural numbers, while `nrec` is the primitive recursor. Both kinds of abstraction and the `nrec` construct carry type annotations, ρ. Terms that are α-equivalent are considered equal.

Definition 2.2.2 (Grammar of types).

$$\rho, \sigma, \tau ::= \mathtt{N} \mid \perp\!\!\!\perp \mid \sigma \to \tau$$

Here, $\mathtt{N}$ is the type of natural numbers and $\perp\!\!\!\perp$ is a special type for commands.

2.2.2 The Typing Relation

There are two kinds of typing environments, one for λ-variables, denoted by Γ, and one for μ-variables, denoted by Δ. These are sets of pairs of the form `free variable:type`. The notation $\Gamma; \Delta \vdash t : \tau$ means that term t has type τ, given the typing environments Γ and Δ. Similarly for commands.

Definition 2.2.3. The typing relation for $\lambda\mu^{\mathbf{T}}$ is defined in Figure 2.2.1.

Most typing rules are similar to the λ-calculus. A few interesting cases are described below:

(nrec) This is the usual rule encountered in System **T**. The term t, which is a natural number, keeps track of how many recursion steps still need to be unfolded; if t is 0, then the expression evaluates to r; otherwise, s is applied to t and the result of the next recursion step. The `nrec` construct is annotated with its own type.

(activate) This rule comes from $\lambda\mu$. It says that a μ-abstraction is typeable if the command inside it is typeable. The type annotation of the μ-variable matches the type of the μ-abstraction.

(passivate) This also comes from $\lambda\mu$. It reads as follows: a command $[\alpha]t$ is typeable if the type of the term t agrees with the type that α has in the environment.

2.2.3 Logical Substitution and Structural Substitution

In order to define the reduction relation, logical substitution and structural substitution are defined.

Definition 2.2.4. Logical substitution is defined recursively on the structure of terms and named terms similarly to substitution in the λ-calculus. The notation $t[x := s]$ means substitute s for all free occurrences of x in t. Logical substitution is capture avoiding for both λ- and μ-variables.

Structural substitution is specific to the $\lambda\mu^{\mathbf{T}}$-calculus. It can be defined by introducing the notion of a $\lambda\mu^{\mathbf{T}}$ context:

Definition 2.2.5. Contexts in $\lambda\mu^{\mathbf{T}}$ are given by the grammar:

$$E ::= \Box \mid Et \mid \mathtt{S}E \mid \mathtt{nrec}_\rho\ r\ s\ E$$

Definition 2.2.6. Substituting a term for the hole in a context is defined as follows:

$$\begin{aligned}
\Box[t] &:= t \\
(Es)[t] &:= E[t]s \\
(\mathtt{S}E)[t] &:= \mathtt{S}E[t] \\
(\mathtt{nrec}_\rho\ r\ s\ E)[t] &:= \mathtt{nrec}_\rho\ r\ s\ E[t]
\end{aligned}$$

Definition 2.2.7. Structural substitution is defined recursively on the structure of terms and commands:

$$\begin{aligned}
x[\alpha := \beta E] &:= x \\
(\lambda x : \rho.r)[\alpha := \beta E] &:= \lambda x : \rho.(r[\alpha := \beta E]) \\
(ts)[\alpha := \beta E] &:= (t[\alpha := \beta E])(s[\alpha := \beta E]) \\
0[\alpha := \beta E] &:= 0 \\
(\mathtt{S}t)[\alpha := \beta E] &:= \mathtt{S}(t[\alpha := \beta E]) \\
(\mathtt{nrec}_\rho\ r\ s\ t)[\alpha := \beta E] &:= \mathtt{nrec}_\rho\ (r[\alpha := \beta E])\ (s[\alpha := \beta E])\ (t[\alpha := \beta E]) \\
(\mu\gamma : \rho.c)[\alpha := \beta E] &:= \mu\gamma : \rho.(c[\alpha := \beta E]) \\
([\alpha]t)[\alpha := \beta E] &:= [\beta]E[t[\alpha := \beta E]] \\
([\gamma]t)[\alpha := \beta E] &:= [\gamma]t[\alpha := \beta E] \qquad \textit{provided that } \gamma \neq \alpha
\end{aligned}$$

Structural substitution is capture avoiding for both λ- and μ-variables.

The notation $t[\alpha := \beta E]$ means that when a named term of the form $[\alpha]s$ (where α is free) is encountered inside t, the μ-variable α is replaced by β, the context E is placed around the term s, and structural substitution is applied recursively to s. So overall, the command $[\alpha]s$ is replaced with the command $[\beta]E[s[\alpha := \beta E]]$.

Structural substitution is a key component of the $\lambda\mu^{\mathbf{T}}$-calculus. The way it operates highlights the role of commands. The μ-variable in a command $[\alpha]s$ tags the term s. This tag allows structural substitution to directly access s, when it is part of a bigger term, and apply an operation to it, given by the context E. This feature is specific to $\lambda\mu^{\mathbf{T}}$; in the λ-calculus there is no mechanism to access subterms directly, any operation has to be applied to a whole term.

2.2.4 The Reduction Relation

Given two more auxiliary definitions, the $\lambda\mu^{\mathbf{T}}$ reduction relation can be defined:

Definition 2.2.8. The expression $\mathrm{FCV}(t)$ represents the set of free μ-variables of t. Analogously, $\mathrm{FV}(t)$ is the set of free λ-variables of t.

Definition 2.2.9. A term or command is λ-*closed* if it has no free λ-variables. Similarly, a term is μ-*closed* if it has no free μ-variables. A term is *closed* if it has no free λ- or μ-variables.

Notation 2.2.10. The notation $\underline{n}$ represent the term $\mathtt{S}^n 0$.

Definition 2.2.11. The one-step reduction relation for $\lambda\mu^{\mathbf{T}}$ is defined as below:

$$
\begin{aligned}
(\lambda x : \rho.t)r &\to t[x := r] && (\beta)\\
\mathtt{S}(\mu\alpha : \rho.c) &\to \mu\alpha : \rho.(c[\alpha := \alpha\ (\mathtt{S}\square)]) && (\mu\mathtt{S})\\
(\mu\alpha : \sigma \to \tau.c)s &\to \mu\alpha : \tau.(c[\alpha := \alpha\ (\square s)]) && (\mu R)\\
\mu\alpha : \rho.[\alpha]t &\to t \qquad \textit{provided that}\ \ \alpha \notin \mathrm{FCV}(t) && (\mu\eta)\\
[\alpha]\mu\beta : \rho.c &\to c[\beta := \alpha\ \square] && (\mu i)\\
\mathtt{nrec}_\rho\ r\ s\ 0 &\to r && (0)\\
\mathtt{nrec}_\rho\ r\ s\ (\mathtt{S}\underline{n}) &\to s\ \underline{n}\ (\mathtt{nrec}_\rho\ r\ s\ \underline{n}) && (\mathtt{S})\\
\mathtt{nrec}_\rho\ r\ s\ (\mu\alpha : \mathtt{N}.c) &\to \mu\alpha : \rho.(c[\alpha := \alpha\ (\mathtt{nrec}_\rho\ r\ s\ \square)]) && (\mu\mathtt{N})
\end{aligned}
$$

Reduction is also allowed inside any $\lambda\mu^{\mathbf{T}}$ term, so these rules can be applied to any subterm. This leads to an evaluation strategy similar to full β-reduction in the λ-calculus.

The reduction rules are described below, paying particular attention to the use of structural substitution:

- (β) This is the usual β-reduction of the λ-calculus.
- (μR) This rule is part of $\lambda\mu$, and describes the case when a μ-abstraction serves as the function in a function application. The use of the structural substitution means that all free occurrences of $[\alpha]t$ inside c are replaced by $[\alpha](ts)$. There are similar rules for successor $(\mu\mathtt{S})$, and `nrec` $(\mu\mathtt{N})$.

$$\frac{}{\Gamma, A \vdash_{MC} A; \Delta}\ (\text{id}) \qquad \frac{\Gamma \vdash_{MC}; A, \Delta}{\Gamma \vdash_{MC} A; \Delta}\ (\text{activate}) \qquad \frac{\Gamma \vdash_{MC} A; A, \Delta}{\Gamma \vdash_{MC}; A, \Delta}\ (\text{passivate})$$

$$\frac{\Gamma, A \vdash_{MC} B; \Delta}{\Gamma \vdash_{MC} A \to B; \Delta}\ (\to_i) \qquad \frac{\Gamma \vdash_{MC} A \to B; \Delta \qquad \Gamma \vdash_{MC} A; \Delta}{\Gamma \vdash_{MC} B; \Delta}\ (\to_e)$$

Figure 2.2.2: Minimal Classical Natural Deduction [6].

(μi) All free occurrences of β in c are replaced by α. This rule also comes from $\lambda\mu$. In Parigot's original presentation [30], it is called a renaming rule because all the named terms of the form $\beta[t]$ change their name to α.

(0)/(`S`) These are the usual rules for primitive recursion from System **T**. The last argument of `nrec` is only allowed to be a natural number. Primitive recursion cannot be performed on arbitrary terms because this would break confluence [14].

Notation 2.2.12. The reflexive-transitive closure of $\to$ is denoted by $\to^*$.

2.2.5 A *Propositions-as-Types* Correspondence for $\lambda\mu^{\mathbf{T}}$

The propositions-as-types correspondence, also known as the Curry-Howard isomorphism [19], relates the simply-typed λ-calculus to Intuitionistic Natural Deduction. In this context, types are viewed as propositions, terms as proofs, and β-reduction as proof simplification, namely cut-elimination [38]. This correspondence has subsequently been extended to other calculi and logics.

In this section, I present a similar correspondence: a mapping from a system called Minimal Classical Natural Deduction [6] to $\lambda\mu^{\mathbf{T}}$. This was originally presented by Parigot [30] for the $\lambda\mu$-calculus.

Definition 2.2.13. Minimal Classical Natural Deduction [6] is defined in Figure 2.2.2. It is presented in the style of the sequent calculus, where sequents have the usual meaning. The symbols Γ and Δ stand for sets of formulae. The only logical connective is implication, $\to$.

In general, a comma in the antecedent of a sequent is used to denote conjunction, and a comma in the succedent, disjunction. Apart from this, Parigot [30] introduces by convention a semicolon on the right-hand-side. This also denotes disjunction, but it is used to isolate a distinguished formula, called the *active* formula, on the left of the semicolon. This is a formula explicitly mentioned in the rule. The rules (activate) and (passivate) cause a formula from Δ to become active or inactive, respectively.

Minimal Classical Natural Deduction implements *minimal classical logic* [20]. The inference rules are the same as in an *intuitionistic* natural deduction system. However, allowing multiple conclusions on the right-hand-side of a sequent makes this system *classical*, as Gentzen showed [12, 13].

Minimal Classical Natural Deduction validates Peirce's Law, $((A \to B) \to A) \to A$, but not Ex Falso Quodlibet, $\bot \to A$, nor the Double Negation Law, $\neg\neg A \to A$ [6]. In this

$$\frac{}{\Gamma, A \vdash_{MC} A; \Delta}\,(\text{id}) \quad \mapsto \quad \frac{x : A \in \Gamma}{\Gamma; \Delta \vdash x : A}\,(\text{axiom})$$

$$\frac{\Gamma, A \vdash_{MC} B; \Delta}{\Gamma \vdash_{MC} A \to B; \Delta}\,(\to_i) \quad \mapsto \quad \frac{\Gamma, x : A; \Delta \vdash t : B}{\Gamma; \Delta \vdash \lambda x : A.t : A \to B}\,(\text{lambda})$$

$$\frac{\Gamma \vdash_{MC} A \to B; \Delta \qquad \Gamma \vdash_{MC} A; \Delta}{\Gamma \vdash_{MC} B; \Delta}\,(\to_e) \quad \mapsto \quad \frac{\Gamma; \Delta \vdash t : A \to B \qquad \Gamma; \Delta \vdash s : A}{\Gamma; \Delta \vdash ts : B}\,(\text{app})$$

$$\frac{\Gamma \vdash_{MC}; A, \Delta}{\Gamma \vdash_{MC} A; \Delta}\,(\text{activate}) \quad \mapsto \quad \frac{\Gamma; \Delta, \alpha : A \vdash c : \perp\!\!\!\perp}{\Gamma; \Delta \vdash \mu\alpha : A.c : A}\,(\text{activate})$$

$$\frac{\Gamma \vdash_{MC} A; A, \Delta}{\Gamma \vdash_{MC}; A, \Delta}\,(\text{passivate}) \quad \mapsto \quad \frac{\Gamma; \Delta \vdash t : A \qquad \alpha : A \in \Delta}{\Gamma; \Delta \vdash [\alpha]t : \perp\!\!\!\perp}\,(\text{passivate})$$

Figure 2.2.3: A *Propositions-as-Types* correspondence between Minimal Classical Natural Deduction and $\lambda\mu^{\mathbf{T}}$.

sense, it does not implement *full* classical logic, hence the adjective *minimal*. However, this can be repaired by adding an elimination rule for $\perp$ (Section 3.2.1).

Proposition 2.2.14. (Ariola and Herbelin [6]) A formula A is provable in minimal classical logic if and only if there exists a closed $\lambda\mu$ term t such that $\vdash t : A$.

The propositions-as-types correspondence between $\lambda\mu^{\mathbf{T}}$ and Minimal Classical Natural Deduction is given in Figure 2.2.3. One can see that the natural numbers in $\lambda\mu^{\mathbf{T}}$ do not have a proposition associated with them. This is because, although propositions can be viewed as types, there is no natural way of interpreting some datatypes, such as natural numbers, as propositions [23].

Making the correspondence between Minimal Classical Natural Deduction and $\lambda\mu$ explicit reveals the role of μ-variables. As in the λ-calculus, λ-variables are used to index formulae in the antecedent of a sequent. Similarly, μ-variables are needed to index *inactive* formulae in the succedent. The (passivate) rule causes a formula to become inactive by giving it an index, while the μ-binding mechanism makes a formula active. The type $\perp\!\!\!\perp$, for commands, is not associated with any formula in the logic; it is a placeholder for the case when the sequent has no active formula.

2.2.6 Computational Content of $\lambda\mu^{\mathbf{T}}$

In the $\lambda\mu^{\mathbf{T}}$-calculus, commands and μ-abstractions introduce a control mechanism. There are two ways of interpreting it [6,14]. The first one is to consider the command $[\alpha]t$, where $\Gamma; \Delta \vdash [\alpha]t : \perp\!\!\!\perp$, as a continuation α that is waiting for a term t, of type $\Delta(\alpha)$. In this interpretation, the role of the μ-abstraction $\mu\alpha : \rho.c$ is to choose a particular result by capturing a continuation that returns a result of type ρ.

The second interpretation regards μ-abstractions as forming a catch-throw mechanism:

Definition 2.2.15. Catch and throw can be defined as follows:

$$\texttt{catch}_\alpha\ t := \mu\alpha : \rho.[\alpha]t$$
$$\texttt{throw}_\beta\ s := \mu\_ : \rho.[\beta]s$$

where the notation _ represents any μ-variable different from β that does not appear free in s.

The `catch` expression defined above catches an exception labelled α, from t. Similarly, `throw` throws the result of s to β. Crolard [7] showed that a calculus with `catch` and `throw` as primitives is equivalent to the $\lambda\mu$-calculus. Moreover, the reduction properties of `catch` and `throw` are shown to be the expected ones [14].

2.3 Isabelle/HOL

Isabelle/HOL is an instantiation of the generic proof assistant Isabelle. Isabelle is part of the LCF (Logic for Computable Functions) family of proof assistants [24]. It aims to provide a common implementation for all systems in this family [15], hence the attribute "generic". Isabelle provides a metalogic in which different object logics can be declared, the one used in Isabelle/HOL being higher-order logic. Roughly speaking, Isabelle works by performing resolution and higher-order unification [34].

2.3.1 Proving in Isabelle/HOL

Isabelle allows the user to define datatypes and functions in a very similar fashion to Standard ML, although only total functions that terminate are permitted. In addition, one can, for example, define relations inductively. Given such definitions, Isabelle infers introduction and elimination rules, similar to the rules of Natural Deduction. These rules, together with induction and simplification (equational reasoning), can be used to prove propositions about the objects defined.

There are two ways of writing proofs in Isabelle: one is the so called *apply*-style, and the other involves using Isabelle's structured proof language, Isar. In apply-style, the user specifies proof methods or rules that are applied to the goal, in order to refine it to a form that is known to be true. Isar is designed to make proofs more similar to pen-and-paper ones, by structuring them as a human would do. This means that Isar proofs are more easily understood by readers than apply-style proofs, but they take longer to write. In my project, I combined these two styles of proofs. I used Isar to structure complicated proofs with a lot of cases, and then apply-style to solve each case.

```
inductive ev :: nat ⇒ bool where
ev0: ev 0 |
evSS: ev n ⟹ ev (Suc(Suc n))

fun evn :: nat ⇒ bool where
evn 0 = True |
evn (Suc 0) = False |
evn (Suc(Suc n)) = evn n

lemma ev(Suc(Suc(Suc(Suc 0))))
apply(rule evSS)
apply(rule evSS)
apply(rule ev0)
done

lemma ev m ⟹ evn m
apply(induction rule: ev.induct)
by simp-all
```

Figure 2.3.1: Isabelle code that defines an inductive predicate and a function that decide whether a number is even. Two simple lemmas follow.

2.3.2 Example Definitions and Proofs

To help the reader form an idea about what proving in Isabelle is like, Figure 2.3.1 shows an example. It contains two definitions and lemmas from *Concrete Semantics* [26].

First, an inductive predicate, *ev*, for even numbers is defined. It includes inductive rules that exactly specify the set of even natural numbers. This is followed by the definition of a function, *evn*, that decides whether a natural number is even. Intuitively, this function computes the same set of natural numbers as the inductive predicate.

The first lemma proves that the number 4 is even. Initially, the goal is the statement of the lemma itself. Rule *evSS*, from the definition of *ev*, is applied to the goal *backwards*, as an introduction rule. This means that the premise of this rule becomes the new goal. That is, the new goal is to prove 2 is even. Rule *evSS* is applied again in the same way. The goal is now to prove that 0 is even. Rule *ev0* is used to establish that this goal is true.

The second lemma proves that if a number satisfies the inductive predicate *ev*, the function *evn* returns *True*. The proof proceeds by rule induction on the hypothesis *ev m*. This results in two subgoals, one for each rule in the definition of *ev*. These are both proved using the automated proof method *simp-all*.

2.4 Tools and Design

The current release of Isabelle includes a user interface based on jEdit, which acts as an IDE. I used this to carry out the $\lambda\mu^{\mathbf{T}}$ formalisation and proofs. Another useful feature of

Isabelle, which I used to prepare this dissertation, is the ability to typeset Latex documents automatically from theories.

I implemented the μ**ML** interpreter in OCaml, using the libraries OCamlLex and Menhir to do lexing and parsing respectively. All the code and documents associated with the project are part of a Git repository hosted on Bitbucket. Using a version control system has proved an efficient way to back-up my project and communicate with my supervisor.

The Isabelle formalisation of $\lambda\mu^{\mathbf{T}}$, which forms a big part of my project, is an atypical software component because it does not require testing. The proofs I completed ensure that the definitions in the formalisation are correct, and Isabelle ensures that the proofs are correct. Nevertheless, after writing all the necessary definitions and before proving the main results about $\lambda\mu^{\mathbf{T}}$, I wrote simple lemmas, similar to unit tests, as a sanity-check of my definitions.

The implementation of the interpreter did require testing. This was to ensure that the parser functions correctly, and that the transformation that the interpreter performs on the parsed abstract syntax tree is correct. I automated these tests using OUnit, which is a unit-test library for OCaml [4].

Chapter 3

Implementation

This chapter is divided in three main sections. First, I present the $\lambda\mu^{\mathbf{T}}$ Isabelle formalisation, including outlines of the Type Preservation and Progress proofs, as well as other important proofs. The next section is dedicated to extensions of $\lambda\mu^{\mathbf{T}}$. Finally, I describe the μ**ML** language and the implementation of its OCaml interpreter.

3.1 Formalising the $\lambda\mu^{\mathbf{T}}$-calculus in Isabelle/HOL

This section presents the most important definitions, lemmas and proofs in the Isabelle formalisation of $\lambda\mu^{\mathbf{T}}$, and draws attention to those that posed difficulties. Most of these follow the formulations in the paper that introduced $\lambda\mu^{\mathbf{T}}$ [14]. However, in the paper, some proofs are only sketched or omitted altogether. As a result, I had to fill in the gaps in my formalisation.

3.1.1 Defining the $\lambda\mu^{\mathbf{T}}$ Syntax Using de Bruijn Notation

The $\lambda\mu^{\mathbf{T}}$ terms and commands can be defined in Isabelle as two mutually recursive datatypes, Figure 3.1.2. Similarly, the $\lambda\mu^{\mathbf{T}}$ types become another Isabelle datatype, Figure 3.1.1.

To deal with α-equivalence, I chose to use de Bruijn notation [8]. Another option is to use Nominal Isabelle [3], a library for Isabelle/HOL that uses Nominal techniques, based on the work of Pitts [37], to address this problem. Using de Bruijn notation is the standard approach, and it is easier to get started with, which is why I chose it.

In this scheme, each bound variable is represented by a number, its index, that indicates the number of abstractions that need to be traversed to arrive at the one that binds

```
datatype type =
    Nat
  | Command
  | Fun type type
```

Figure 3.1.1: The $\lambda\mu^{\mathbf{T}}$ types defined as a datatype in Isabelle.

```
datatype dBT =
      LVar nat
    | Lbd type dBT
    | App dBT dBT
    | Mu type dBC
    | Zero
    | S dBT
    | Nrec type dBT dBT dBT

and dBC =
      MVar nat dBT
```

Figure 3.1.2: The $\lambda\mu^{\mathbf{T}}$ terms and commands defined as two mutually recursive datatypes in Isabelle.

this variable. Each free variable has an index that represents it when the variable appears in the top-level context, not enclosed in any abstractions. If the free variable occurs inside n abstractions, its index is incremented by n. This is best illustrated with an example from the λ-calculus. If the index of the free variable x is 3 in the top-level context, the λ-terms $\lambda y.\lambda z.((z\ y)\ x)$ and $\lambda v.\lambda w.((w\ v)\ x)$ are both represented in de Bruijn notation as $\lambda.\lambda.((0\ 1)\ 5)$.

Applying this notation to $\lambda\mu^{\mathbf{T}}$ means that there will be two disjoint sets of indices: one for λ-variables, and one for μ-variables. A λ-abstraction is written as:

$$\lambda : \rho.t$$

where ρ is a type annotation. The binder is not specified anymore because it is implicitly 0. Similarly, μ-abstractions are written as:

$$\mu : \rho.c$$

3.1.2 The Typing Relation

In Isabelle, I chose to define typing environments, Γ and Δ, as functions from natural numbers to types. An empty typing environment can be expressed by any function whatsoever because it is never going to be accessed when a typing judgement is valid. This is the method used in the λ-calculus implementation included in the Isabelle/HOL distribution [2].

Consider the typing judgement:

$$\Gamma; \Delta \vdash \lambda : \rho.(3\ 0) : \rho \to \delta$$

Using de Bruijn notation, the free variable 3, that appears inside the λ, corresponds to variable 2 from the typing environment Γ. Therefore it must be the case that 2 has type $\rho \to \delta$ in Γ.

In order to prove the judgement above, one would need to prove:

$$\Gamma\langle 0 : \rho\rangle; \Delta \vdash (3\ 0) : \delta$$

```
definition
  shift :: (nat ⇒ 'a) ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a
where
  e⟨i:a⟩ = (λj. if j < i then e j else if j = i then a else e (j−1))
```

Figure 3.1.3: Definition of the environment update function in Isabelle.

For this to be provable, the free variable 3 in the typing environment $\Gamma\langle 0 : \rho\rangle$ must have the same type as 2 in Γ. To make sure that the typing environment is changed accordingly, the update operation $\Gamma\langle 0 : \rho\rangle$ is a *shifting* operation: the value of Γ at 0 is now ρ, and all other variables that were previously in the environment are shifted up by one. So, if before, 2 was associated with $\rho \to \delta$, now 3 is associated with this type instead.

Definition 3.1.1 (Updating a typing environment)**.** In general, the operation $\langle \_ : \_\rangle$, that is used to add a new variable to the typing environment, is defined as:

$$\Gamma\langle n : \rho\rangle(m) := \begin{cases} \Gamma(m) & \text{if } m < n \\ \rho & \text{if } m = n \\ \Gamma(m-1) & \text{if } m > n \end{cases}$$

where Γ is a function from natural numbers to $\lambda\mu^{\mathbf{T}}$ types.

In Isabelle, I defined environment update as a function, as in Figure 3.1.3. The keyword **definition** is used for functions that are not recursive.

Using this representation of variables and typing environments, I defined the $\lambda\mu^{\mathbf{T}}$ typing relation in Isabelle as an inductive predicate, Figure 3.1.4. Since terms and commands are two mutually recursive datatypes, I had to define two mutually recursive typing relations, one for each of them.

3.1.3 Logical Substitution

I defined logical substitution in Isabelle as two mutually recursive functions, one for terms and one for commands, Figure 3.1.5.

All the rules are as expected apart from those for λ-variables, λ-abstractions, and μ-abstractions:

1. The rule for λ-variables is:

$$x[y := s] := \begin{cases} s & \text{if } x = y \\ x-1 & \text{if } x > y \\ x & \text{if } x < y \end{cases} \tag{3.1.1}$$

   where x and y are natural numbers and s is a term. The need to decrement x if $x > y$ arises because the reduction rule (β):

$$(\lambda : \rho.t)r \to t[0 := r]$$

```
inductive typing-dBT :: (nat ⇒ type) ⇒ (nat ⇒ type) ⇒ dBT ⇒ type ⇒ bool
and typing-dBC :: (nat ⇒ type) ⇒ (nat ⇒ type) ⇒ dBC ⇒ type ⇒ bool
where
    var: Γ x = T ⟹ Γ, Δ ⊢_T ‘x : T
  | zero: Γ, Δ ⊢_T Zero : Nat
  | suc: Γ, Δ ⊢_T t : Nat ⟹ Γ, Δ ⊢_T (S t) : Nat
  | app: (Γ, Δ ⊢_T t : (T1→T2)) ⟹ (Γ, Δ ⊢_T s : T1)
      ⟹ Γ, Δ ⊢_T (t°s) : T2
  | lambda: Γ⟨0:T1⟩, Δ ⊢_T t : T2
      ⟹ Γ, Δ ⊢_T (λ T1 : t) : (T1→T2)
  | nrec: Γ, Δ ⊢_T r : T ⟹
        Γ, Δ ⊢_T s : (Nat→T→T) ⟹
        Γ, Δ ⊢_T t : Nat
        ⟹ Γ, Δ ⊢_T (Nrec T r s t) : T
  | activate: Γ, Δ⟨0:T⟩ ⊢_C c : Command ⟹ Γ, Δ ⊢_T (μ T : c) : T
  | passivate: Γ, Δ ⊢_T t : T ⟹ (Δ x = T)
        ⟹ Γ, Δ ⊢_C (<x> t) : Command
```

Figure 3.1.4: The $\lambda\mu^{\mathbf{T}}$ typing rules as an inductive predicate in Isabelle.

```
primrec
  subst-dBT :: [dBT, dBT, nat] ⇒ dBT and
  subst-dBC :: [dBC, dBT, nat] ⇒ dBC
where
    subst-LVar: (‘i)[s/k]^T =
      (if k < i then ‘(i−1) else if k = i then s else (‘i))
  | subst-Lbd: (λ T : t)[s/k]^T = λ T : (t[(liftL-dBT s 0) / k+1]^T)
  | subst-App: (t ° u)[s/k]^T = t[s/k]^T ° u[s/k]^T
  | subst-Mu: (μ T : c)[s/k]^T = μ T : (c[(liftM-dBT s 0) / k]^C)
  | subst-Zero: Zero[s/k]^T = Zero
  | subts-S: (S t)[s/k]^T = (S (t[s/k]^T))
  | subst-Nrec: (Nrec T t u v)[s/k]^T = Nrec T (t[s/k]^T) (u[s/k]^T) (v[s/k]^T)
  | subst-MVar: (<i> t)[s/k]^C = <i> (t[s/k]^T)
```

Figure 3.1.5: Logical substitution defined as a function in Isabelle. The function $\uparrow_\lambda$ is denoted by *liftL*, and $\uparrow_\mu$ by *liftM*.

has to preserve typing. So if

$$\Gamma; \Delta \vdash (\lambda : \rho.t)r : \delta \tag{3.1.2}$$

holds, I wanted to ensure that

$$\Gamma; \Delta \vdash t[0 := r] : \delta \tag{3.1.3}$$

also holds. According to the typing rules, judgement 3.1.2 holds if

$$\Gamma\langle 0 : \rho\rangle; \Delta \vdash t : \delta$$

is true. So, in order for judgement 3.1.3 to be provable, the free variables in t that are greater than 0 need to be shifted down by 1.

2. The second case in the definition of logical substitution that needed to be adapted is that of λ-abstractions:

   $$(\lambda : \rho.t)[x := s] := \lambda : \rho.(t[x + 1 := \uparrow^0_\lambda (s)])$$

   The value of x needs to be incremented because the index of a free variable is incremented when the variable is inside a λ-abstraction. The function $\uparrow^0_\lambda (\_)$ is needed to ensure logical substitution is capture avoiding.

   Definition 3.1.2 (Lifting functions)**.** The function $\uparrow^n_\lambda (t)$ takes as an argument a term or command, t, and increases by 1 the index of all free λ-variables in t that are greater or equal to n. The function $\uparrow^n_\mu (t)$ performs the same operation but for free μ-variables.

   Being capture avoiding is a standard property of substitution in the λ-calculus. Using the previous example, it means that no free variable in s should have the same name as the binding variable of the λ. Otherwise, such a free variable would become bound as a result of the substitution.

3. When defining logical substitution for μ-abstractions, the function, $\uparrow^0_\mu (\_)$, needs to be used:

   $$(\mu : \rho.c)[x := s] := \mu : \rho.(c[x := \uparrow^0_\mu (s)])$$

3.1.4 Structural Substitution

I defined the $\lambda\mu^{\mathbf{T}}$ contexts as a datatype in Isabelle without any modifications. Context substitution is defined as a function from contexts to terms.

The definition of structural substitution appears in Figure 3.1.6. The cases that needed adaptation are described below:

1. $$(\lambda : \rho.r)[\alpha := \beta E] := \lambda : \rho.(r[\alpha := \beta \ \uparrow^0_\lambda (E)])$$

   The function $\uparrow^n_\lambda (\_)$ is the lifting function for free λ-variables applied to contexts. It increments the indices of all free λ-variables in E by 1. This is needed to make structural substitution capture avoiding for λ-variables.

```
primrec
  struct-subst-dBT :: [dBT, nat, nat, ctxt] ⇒ dBT
and
  struct-subst-dBC :: [dBC, nat, nat, ctxt] ⇒ dBC
where
    struct-LVar: (‘i)[j=k E]^T = (‘i)
  | struct-Lbd: (λ T : t)[j=k E]^T = (λ T : (t[j=k (liftL-ctxt E 0)]^T))
  | struct-App: (t°s)[j=k E]^T = (t[j=k E]^T)°(s[j=k E]^T)
  | struct-Zero: Zero[j=k E]^T = Zero
  | struct-Suc: (S t)[j=k E]^T = S (t[j=k E]^T)
  | struct-Nrec: (Nrec T r s t)[j=k E]^T = Nrec T (r[j=k E]^T) (s[j=k E]^T) (t[j=k E]^T)
  | struct-Mu: (μ T : c)[j=k E]^T = μ T : (c[(j+1)=(k+1) (liftM-ctxt E 0)]^C)
  | struct-MVar: (<i> t)[j=k E]^C =
      (if i=j then (<k> (ctxt-subst E (t[j=k E]^T)))
       else (if j<i ∧ i≤k then (<i−1> (t[j=k E]^T))
            else (if k≤i ∧ i<j then (<i+1> (t[j=k E]^T))
                 else (<i> (t[j=k E]^T)))))
```

Figure 3.1.6: Structural substitution as a primitive recursive function in Isabelle.

2. $$(\mu:\rho.c)[\alpha \coloneqq \beta E] \coloneqq \mu:\rho.(c[(\alpha+1) \coloneqq (\beta+1)\ \uparrow^0_\mu (E)])$$

   The context E is lifted to make structural substitution capture avoiding for μ-variables. The free μ-variables α and β are incremented because they are now inside a μ-abstraction.

3. In the rule for μ-variables, I had to carefully adjust the de Bruijn indices of the free μ-variables encountered:

$$([\gamma]t)[\alpha \coloneqq \beta E] \coloneqq \begin{cases} [\beta](E[t[\alpha \coloneqq \beta E]]) & \text{if } \gamma = \alpha \\ [\gamma-1](t[\alpha \coloneqq \beta E]) & \text{if } \alpha < \gamma \leq \beta \\ [\gamma+1](t[\alpha \coloneqq \beta E]) & \text{if } \beta \leq \gamma < \alpha \\ [\gamma](t[\alpha \coloneqq \beta E]) & \text{otherwise} \end{cases} \tag{3.1.4}$$

   The case split above is needed to ensure that typing in preserved under structural substitution. An informal explanation would be as follows: after the substitution, the free variable α is replaced in the typing environment by β. First, examine the case $\alpha < \gamma \leq \beta$. If α has been added to the typing environment using the environment update operation $\langle\_ : \_\rangle$, γ actually represents the variable $\gamma - 1$ shifted up by 1. However, if β is added instead, $\gamma - 1$ is not shifted up. Hence the need to decrement γ by 1 when α is replaced by β. The case $\beta \leq \gamma < \alpha$ is similar.

   The definition of this case has been a source of difficulties and required experimentation. I only realised a case split is needed when I was unable to prove the Structural Substitution lemma, needed for Type Preservation of $\lambda\mu^{\mathbf{T}}$.

inductive $\mathit{beta\text{-}terms} :: [\mathit{dBT},\ \mathit{dBT}] \Rightarrow \mathit{bool}$
  and $\mathit{beta\text{-}command} :: [\mathit{dBC},\ \mathit{dBC}] \Rightarrow \mathit{bool}$
where
  beta: $(\lambda\ T : t)^{\circ}r \rightarrow_{\beta} t[r/0]^{T}$

| muSuc: $S\ (\mu\ T : c) \rightarrow_{\beta} \mu\ T : (c[0 = 0\ (\mathit{CSuc}\ \Diamond)]^{C})$

| muApp: $(\mu\ (T1 \rightarrow T2) : c)^{\circ}s \rightarrow_{\beta} \mu\ T2 : (c[0 = 0\ (\Diamond\ ^{\bullet}\ (\mathit{liftM\text{-}dBT}\ s\ 0))]^{C})$

| $\mathit{muRename}$: $(0 \notin (\mathit{fmv\text{-}dBT}\ t\ 0)) \Longrightarrow (\mu\ T : (<0>\ t)) \rightarrow_{\beta} \mathit{dropM\text{-}dBT}\ t\ 0$

| mVar: $<i>\ (\mu\ T : c)\ {}_{C}\!\!\rightarrow_{\beta} (\mathit{dropM\text{-}dBC}\ (c[0 = i\ \Diamond]^{C})\ i)$

| $\mathit{nrecZero}$: $\mathit{Nrec}\ T\ r\ s\ \mathit{Zero} \rightarrow_{\beta} r$
| $\mathit{nrecSuc}$: $\mathit{is\text{-}natval}\ n \Longrightarrow \mathit{Nrec}\ T\ r\ s\ (S\ n) \rightarrow_{\beta} s^{\circ}n^{\circ}(\mathit{Nrec}\ T\ r\ s\ n)$
| nrecMu: $\mathit{Nrec}\ T\ r\ s\ (\mu\ T1 : c)$
  $\rightarrow_{\beta} \mu\ T : (c[0 = 0\ (\mathit{CNrec}\ T\ (\mathit{liftM\text{-}dBT}\ r\ 0)\ (\mathit{liftM\text{-}dBT}\ s\ 0)\ \Diamond)]^{C})$

| lambda: $s \rightarrow_{\beta} t \Longrightarrow (\lambda\ T : s) \rightarrow_{\beta} (\lambda\ T : t)$
| appL: $s \rightarrow_{\beta} u \Longrightarrow (s^{\circ}t) \rightarrow_{\beta} (u^{\circ}t)$
| appR: $t \rightarrow_{\beta} u \Longrightarrow (s^{\circ}t) \rightarrow_{\beta} (s^{\circ}u)$
| mu: $c\ {}_{C}\!\!\rightarrow_{\beta} d \Longrightarrow (\mu\ T : c) \rightarrow_{\beta} (\mu\ T : d)$
| suc: $s \rightarrow_{\beta} t \Longrightarrow (S\ s) \rightarrow_{\beta} (S\ t)$

| nrecL: $r \rightarrow_{\beta} u \Longrightarrow (\mathit{Nrec}\ T\ r\ s\ t) \rightarrow_{\beta} (\mathit{Nrec}\ T\ u\ s\ t)$
| nrecM: $s \rightarrow_{\beta} u \Longrightarrow (\mathit{Nrec}\ T\ r\ s\ t) \rightarrow_{\beta} (\mathit{Nrec}\ T\ r\ u\ t)$
| nrecR: $t \rightarrow_{\beta} u \Longrightarrow (\mathit{Nrec}\ T\ r\ s\ t) \rightarrow_{\beta} (\mathit{Nrec}\ T\ r\ s\ u)$
| $\mathit{mVar2}$: $t \rightarrow_{\beta} s \Longrightarrow (<i>\ t)\ {}_{C}\!\!\rightarrow_{\beta} (<i>\ s)$

Figure 3.1.7: The $\lambda\mu^{\mathbf{T}}$ reduction relation as an inductive predicate in Isabelle.

3.1.5 The Reduction Relation

Using the definitions of substitution, I formalised the $\lambda\mu^{\mathbf{T}}$ reduction relation as an inductive predicate. All the rules appear in Figure 3.1.7. Below, I draw attention to some of the rules that needed adjustments:

1. In rules (μR) and $(\mu \mathtt{N})$, a structural substitution has to be performed inside a μ-abstraction, as a result of the reduction. The context used for the structural substitution is either $(\Box\ s)$ or $(\mathtt{nrec}_{\rho}\ r\ s\ \Box)$. To avoid capture of the free μ-variables in these contexts, their indices need to be incremented by 1.

$$(\mu : \sigma \rightarrow \tau.c)s \rightarrow \mu : \tau.(c[0 := 0\ (\Box\ \uparrow^{0}_{\mu}(s))]) \qquad (\mu R)$$

$$\mathtt{nrec}_{\rho}\ r\ s\ (\mu : \mathtt{N}.c) \rightarrow \mu : \rho.(c[0 := 0\ (\mathtt{nrec}_{\rho}\ \uparrow^{0}_{\mu}(r)\ \uparrow^{0}_{\mu}(s)\ \Box)]) \qquad (\mu \mathtt{N})$$

2. In rule $(\mu\eta)$, the indices of the free μ-variables in t need to be adjusted when t is no longer inside a μ-abstraction:

$$\mu : \rho.[0]t \rightarrow \downarrow_{\mu}^{0} (t) \qquad \textit{provided that } 0 \notin \mathrm{FCV}(t) \tag{$\mu\eta$}$$

Definition 3.1.3 (Drop function). The function $\downarrow_{\mu}^{n} (t)$ takes as an argument a term or command, t, and decreases by 1 the index of the free μ-variables in t that are strictly greater than n. It can be seen as the opposite of $\uparrow_{\mu}^{n} (t)$.

To check the side condition of this reduction rule, I defined functions that calculate the set of free μ-variables of a term and command respectively.

3. One thing to note is that, in rule (β), the free λ-variables in t do not need any further adjustment, although t loses an enclosing λ-abstraction:

$$(\lambda : \rho.t)r \rightarrow t[0 := r] \tag{β}$$

thanks to the way logical substitution is implemented.

4. All the reduction rules which are explicitly stated can be used inside any term or command. Therefore, in my Isabelle formalisation of the reduction relation, I added new rules, known as congruence rules, such as:

$$t \rightarrow u \implies (ts) \rightarrow (us)$$

Definition 3.1.4. The reflexive-transitive closure of the reduction relation is defined as an inductive predicate with rules:

$$t \rightarrow^* t$$
$$t \rightarrow s \ \wedge \ s \rightarrow^* u \implies t \rightarrow^* u$$

3.1.6 Logical Substitution Lemma

This section begins the presentation of lemmas used in the proof of Type Preservation. For convenience, I only state them for terms, but they can be formulated in exactly the same way for commands.

Lemma 3.1.5. For any typing environment Γ, variable index n, and types ρ and δ:

$$\Gamma\langle n : \rho\rangle\langle 0 : \delta\rangle = \Gamma\langle 0 : \delta\rangle\langle n+1 : \rho\rangle$$

Proof. By the definition of $\langle \_ : \_\rangle$. □

Lemma 3.1.6 (Lifting preserves typing).

$$\Gamma; \Delta \vdash t : \rho \implies \Gamma\langle x : \delta\rangle; \Delta \vdash \uparrow_{\lambda}^{x} (t) : \rho$$
$$\Gamma; \Delta \vdash t : \rho \implies \Gamma; \Delta\langle \alpha : \delta\rangle \vdash \uparrow_{\mu}^{\alpha} (t) : \rho$$

Proof. Each implication is proved separately by rule induction on the typing judgement $\Gamma; \Delta \vdash t : \rho$. $\square$

I used these two results to prove a substitution lemma for logical substitution, similar to the one in the typed λ-calculus. This shows that typing is preserved under logical substitution:

Lemma 3.1.7 (Logical Substitution lemma)**.** If

$$\Gamma\langle x : \delta\rangle; \Delta \vdash t : \rho$$

and

$$\Gamma; \Delta \vdash r : \delta$$

then

$$\Gamma; \Delta \vdash t[x \coloneqq r] : \rho$$

Proof. The proof is done by rule induction on the first typing judgement. An interesting case is (lambda):

(lambda) Assume that

$$\begin{gathered}\Gamma\langle x : \delta\rangle; \Delta \vdash (\lambda : \rho.t) : \rho \to \sigma \\ \Gamma; \Delta \vdash r : \delta\end{gathered}$$

Therefore,

$$\Gamma\langle x : \delta\rangle\langle 0 : \rho\rangle; \Delta \vdash t : \sigma$$

We need to prove that:

$$\Gamma; \Delta \vdash (\lambda : \rho.t)[x \coloneqq r] : \rho \to \sigma$$

By the definition of logical substitution, this is equivalent to proving:

$$\Gamma\langle 0 : \rho\rangle; \Delta \vdash t[(x+1) \coloneqq \uparrow^0_\lambda (r)] : \sigma$$

This follows from the induction hypothesis using Lemmas 3.1.5 and 3.1.6.

$\square$

Figure 3.1.8 shows how the Logical Substitution lemma is proved in Isabelle. This is an example of an apply-style proof, which is arguably not very readable. First, the lemma is stated for both terms and commands. The proof proceeds by mutual induction on the typing judgements of t and c. Some automated proof methods such as *auto* and *fastforce* are used, while the results in Lemma 3.1.6 are applied explicitly.

theorem *subst-type*:
$\Gamma 1,\ \Delta \vdash_T t : T \Longrightarrow \Gamma,\ \Delta \vdash_T r : T1 \Longrightarrow \Gamma 1 = \Gamma\langle k{:}T1\rangle \Longrightarrow \Gamma,\ \Delta \vdash_T t[r/k]^T : T$
$\Gamma 1,\ \Delta \vdash_C c : \mathit{Command} \Longrightarrow \forall\Gamma.\forall r.\forall T1.\ (\Gamma,\ \Delta \vdash_T r : T1 \longrightarrow$
$(\forall k.\ (\Gamma 1 = \Gamma\langle k{:}T1\rangle \longrightarrow \Gamma,\ \Delta \vdash_C c[r/k]^C : \mathit{Command})))$
apply(*induct arbitrary*: Γ *k T1 r rule*: *typing-dBT-typing-dBC.inducts*)
apply(*auto*)
apply(*rotate-tac 2*)
apply(*drule liftL-type*(*1*))
apply(*fastforce*)
apply(*rotate-tac 2*)
apply(*drule liftM-type*(*1*))
apply(*fastforce*)
done

Figure 3.1.8: The Logical Substitution lemma proved in Isabelle.

3.1.7 Contextual Typing Judgements

Structural substitution is formulated using $\lambda\mu^{\mathbf{T}}$ contexts. To express the fact that structural substitution preserves typing, contextual typing judgements are introduced [14]. I defined this in Isabelle as an inductive predicate.

Definition 3.1.8. The rules for contextual typing judgments are presented in Figure 3.1.9. The notation $\Gamma;\Delta \vdash E : \sigma \Leftarrow \rho$ means that $E[t]$ has type σ in the given environment if $\Gamma;\Delta \vdash t : \rho$.

Lemma 3.1.9. The typing judgement:

$$\Gamma;\Delta \vdash E[t] : \sigma$$

holds if and only if the following two judgements both hold:

$$\Gamma;\Delta \vdash E : \sigma \Leftarrow \rho$$
$$\Gamma;\Delta \vdash t : \rho$$

This expresses the fact that contextual typing judgements have the desired meaning.

Proof. The forward direction is proved by structural induction on the context E. The reverse direction, by rule induction on the contextual typing judgement. □

Introducing contextual typing judgements allowed me to formulate and prove the following property of the reflexive-transitive closure of the reduction relation:

Lemma 3.1.10. If

$$\Gamma;\Delta \vdash E : \sigma \Leftarrow \rho$$

then

$$E[\mu : \rho.c] \to^* \mu : \sigma.(c[0 := 0 \ \uparrow^0_\mu (E)])$$

Proof. By structural induction on the context E. □

$$\Gamma;\Delta \vdash \Box : \rho \Leftarrow \rho \;\text{(hole)} \qquad \frac{\Gamma;\Delta \vdash E : \mathtt{N} \Leftarrow \rho}{\Gamma;\Delta \vdash \mathtt{S}E : \mathtt{N} \Leftarrow \rho}\;\text{(suc)}$$

$$\frac{\Gamma;\Delta \vdash E : \sigma \to \delta \Leftarrow \rho \qquad \Gamma;\Delta \vdash t : \sigma}{\Gamma;\Delta \vdash Et : \delta \Leftarrow \rho}\;\text{(app)}$$

$$\frac{\Gamma;\Delta \vdash r : \sigma \qquad \Gamma;\Delta \vdash s : \mathtt{N} \to \sigma \to \sigma \qquad \Gamma;\Delta \vdash E : \mathtt{N} \Leftarrow \rho}{\Gamma;\Delta \vdash \mathtt{nrec}\; r\; s\; E : \sigma \Leftarrow \rho}\;\text{(nrec)}$$

Figure 3.1.9: Contextual typing judgements in $\lambda\mu^{\mathbf{T}}$ [14].

The assumption about the type of the context E is needed because the μ-abstraction that results from the reduction needs to have the same type, σ, as $E[\mu : \rho.c]$. Therefore, the type annotation of this μ-abstraction needs to be σ. This is a consequence of the fact that the one-step reduction relation preserves typing, which I will prove in Section 3.1.9.

The lemma above is an example of a proposition from the original $\lambda\mu^{\mathbf{T}}$ paper [14] which I had to adapt in order for it to be provable in my formalisation. The original formulation does not include the typing assumption for E. This is because type annotations on the μ- and λ-abstractions are omitted. In this sense, my formulation of the lemma is more precise.

3.1.8 Structural Substitution Lemma

Using contextual typing judgements, I introduce a lemma for contexts which states that lifting preserves typing of contexts:

Lemma 3.1.11 (Lifting preserves typing for contexts)**.**

$$\Gamma;\Delta \vdash E : \sigma \Leftarrow \rho \implies \Gamma\langle x : \delta\rangle;\Delta \vdash \uparrow^{x}_{\lambda}(E) : \sigma \Leftarrow \rho$$
$$\Gamma;\Delta \vdash E : \sigma \Leftarrow \rho \implies \Gamma;\Delta\langle \alpha : \delta\rangle \vdash \uparrow^{\alpha}_{\mu}(E) : \sigma \Leftarrow \rho$$

Proof. Each implication is proved by rule induction on the contextual typing judgement. □

Using the results presented so far, I proved a substitution lemma for structural substitution:

Lemma 3.1.12 (Structural Substitution lemma)**.** Given that

$$\Gamma;\Delta\langle \alpha : \delta\rangle \vdash t : \rho$$
$$\Gamma;\Delta \vdash E : \sigma \Leftarrow \delta$$

both hold, then

$$\Gamma;\Delta\langle \beta : \sigma\rangle \vdash t[\alpha := \beta\; \uparrow^{\beta}_{\mu}(E)] : \rho$$

Proof. By rule induction on the first typing judgement. I present the (activate) and (passivate) cases. The (lambda) case is similar to (activate) and all the other cases follow from the induction hypothesis.

(activate) By assumption, we know that:

$$\Gamma; \Delta\langle \alpha : \delta \rangle \vdash (\mu : \rho.c) : \rho$$
$$\Gamma; \Delta \vdash E : \sigma \Leftarrow \delta$$

Therefore, using Lemmas 3.1.5 and 3.1.11 respectively:

$$\Gamma; \Delta\langle 0 : \rho \rangle\langle \alpha + 1 : \delta \rangle \vdash c : \perp\!\!\!\perp \tag{3.1.5}$$
$$\Gamma; \Delta\langle 0 : \rho \rangle \vdash \uparrow^0_\mu (E) : \sigma \Leftarrow \delta \tag{3.1.6}$$

We need to prove:

$$\Gamma; \Delta\langle \beta : \sigma \rangle \vdash (\mu : \rho.c)[\alpha \coloneqq \beta \ \uparrow^\beta_\mu (E)] : \rho$$

which is equivalent to:

$$\Gamma; \Delta\langle 0 : \rho \rangle\langle \beta + 1 : \sigma \rangle \vdash c[(\alpha + 1) \coloneqq (\beta + 1) \ \uparrow^0_\mu (\uparrow^\beta_\mu (E))] : \perp\!\!\!\perp$$

again by Lemma 3.1.5 about commutativity of $\langle \_ : \_ \rangle$. It can be shown that:

$$\uparrow^0_\mu (\uparrow^\beta_\mu (E)) = \uparrow^{\beta+1}_\mu (\uparrow^0_\mu (E))$$

so we can use assumptions 3.1.5 and 3.1.6, together with the induction hypothesis, to deduce the conclusion.

(passivate) By assumption, we know that:

$$\Gamma; \Delta\langle \alpha : \delta \rangle \vdash [\gamma]t : \perp\!\!\!\perp \tag{3.1.7}$$
$$\Gamma; \Delta \vdash E : \sigma \Leftarrow \delta \tag{3.1.8}$$

and we need to prove:

$$\Gamma; \Delta\langle \beta : \sigma \rangle \vdash ([\gamma]t)[\alpha \coloneqq \beta \ \uparrow^\beta_\mu (E)] : \perp\!\!\!\perp \tag{3.1.9}$$

At this point, a case split needs to be done according to the value of γ relative to α and β. The cases are the ones in the definition of structural substitution for commands. In fact, I realised the need of doing a case split in this definition while trying to prove the current lemma. This need arises because the operation $\langle \_ : \_ \rangle$ shifts up the variables in the environment.

Cases $\alpha \neq \gamma$: the conclusion follows from the induction hypothesis and assumptions 3.1.7 and 3.1.8.

Case $\gamma = \alpha$: from the definition of structural substitution and the typing rules, the conclusion, 3.1.9, becomes equivalent to:

$$\Gamma; \Delta\langle \beta : \sigma \rangle \vdash (\uparrow^\beta_\mu (E))[t[\alpha \coloneqq \beta \ \uparrow^\beta_\mu (E)]] : \sigma$$

We can break this down into two statements using Lemma 3.1.9 about contextual typing judgements. The first one is:

$$\Gamma; \Delta\langle\beta : \sigma\rangle \vdash \uparrow^{\beta}_{\mu} (E) : \sigma \Leftarrow \delta$$

which follows from assumption 3.1.8 and Lemma 3.1.11, about context lifting. And the second one is:

$$\Gamma; \Delta\langle\beta : \sigma\rangle \vdash\ t[\alpha := \beta\ \uparrow^{\beta}_{\mu} (E)] : \delta$$

which follows from the induction hypothesis.

□

I had to make an adjustment in the statement of Lemma 3.1.12 above, compared to its original presentation [14], to cope with the use of de Bruijn indices. When the context E is used for structural substitution, all its free μ-variables greater or equal to β are lifted, $\uparrow^{\beta}_{\mu} (E)$. This is because E was previously typed in environment Δ, but in the conclusion the environment is $\Delta\langle\beta : \sigma\rangle$.

3.1.9 Type Preservation Theorem

Having proved the logical substitution and structural substitution lemmas, I was able to prove that $\lambda\mu^{\mathbf{T}}$ obeys type preservation. This is one of the main results I aimed to prove in my project.

Theorem 3.1.13 (Type Preservation)**.** Given any $\lambda\mu^{\mathbf{T}}$ terms (or commands) t and s, any type ρ, and typing environments Γ and Δ, if:

$$\Gamma; \Delta \vdash t : \rho$$

and

$$t \to s$$

then

$$\Gamma; \Delta \vdash s : \rho$$

Proof. By rule induction on the typing judgement $\Gamma; \Delta \vdash t : \rho$. Some of the cases are:

(passivate) Assume that:

$$\begin{aligned} &\Gamma; \Delta \vdash [\alpha]t : \perp\!\!\!\perp \\ &[\alpha]t \to d \end{aligned} \tag{3.1.10}$$

We can do a case split on the reduction in 3.1.10. An interesting case is:

– or $t = \mu : \rho.c$, where $\rho = \Delta(\alpha)$, and $d = \downarrow^{\alpha}_{\mu} (c[0 := \alpha\ \Box])$. We can deduce from the assumptions that:

$$\Gamma; \Delta\langle 0 : \rho\rangle \vdash c : \perp\!\!\!\perp$$

We can use the fact $\Gamma; \Delta \vdash \Box : \rho \Leftarrow \rho$ to apply the structural substitution lemma and obtain:

$$\Gamma; \Delta\langle \alpha : \rho \rangle \vdash c[0 \coloneqq \alpha\ \Box] : \perp\!\!\!\perp$$

To obtain the final result:

$$\Gamma; \Delta \vdash \downarrow_{\mu}^{\alpha} (c[0 \coloneqq \alpha\ \Box]) : \perp\!\!\!\perp$$

one can prove that decrementing the μ-variables in c using $\downarrow_{\mu}^{\alpha} (\_)$ preserves typing. This is a similar result to the one proved for lifting.

(app) The two interesting cases are:

- $t = (\lambda : \sigma.r)w$ and $s = r[0 \coloneqq w]$. For this case, we can use the logical substitution lemma together with the induction hypothesis.
- $t = (\mu : \sigma \to \tau.c)r$ and $s = \mu : \tau.(c[0 \coloneqq 0\ \uparrow_{\mu}^{0} (\Box\ r)])$. The structural substitution lemma is used in this case.

□

The formulation of the Type Preservation theorem, for terms and commands, in Isabelle appears in Figure 3.1.10. The proof is done by mutual induction on the typing judgements for t and c. The complete proof can be found in Appendix A.1.

This is an example of a structured proof, where I used the Isar proof language to separate the cases that arise from the induction. In this proof, it is convenient to do so because a lot of the cases need to be handled manually.

In Figure 3.1.10, I included the proof of the application case. First, the assumptions and the goal of this case are stated explicitly. Then, the proof proceeds in apply-style because this is less verbose. Notice that the logical substitution lemma, *subst-type*, is used. Then, a specific instantiation of the structural substitution lemma is applied. This is similar to what I described in the (app) case, in the proof above.

3.1.10 Progress Theorem

In order to formulate a progress theorem, which is the second main theorem in my project, I need to introduce the notion of values in $\lambda\mu^{\mathbf{T}}$. I defined them in Isabelle as an inductive predicate.

Definition 3.1.14. The set of values in $\lambda\mu^{\mathbf{T}}$ is defined inductively as:

$$v \coloneqq 0 \mid \mathtt{S}\underline{n} \mid \lambda : \rho.t$$

Lemma 3.1.15. For any λ-closed value v, such that $\Gamma; \Delta \vdash v : \rho$:

$$\rho = \mathtt{N} \implies v = \underline{n} \qquad \text{where } n \text{ is a natural number}$$
$$\rho = \sigma \to \tau \implies v = \lambda : \sigma.r \qquad \text{for some term } r$$

Proof. By structural induction on the value v. □

```
theorem type-preservation:
  Γ, Δ ⊢T t : T ⟹ t →β s ⟹ Γ, Δ ⊢T s : T
  Γ, Δ ⊢C c : Command ⟹ ∀ d. (c C→β d ⟶ Γ, Δ ⊢C d : Command)

proof(induct arbitrary: s rule: typing-dBT-typing-dBC.inducts)
  fix Γ Δ t T1 T2 s sa
  assume Γ , Δ ⊢T t : T1 → T2
        (⋀s. t →β s ⟹ Γ , Δ ⊢T s : T1 → T2)
        Γ , Δ ⊢T s : T1
        (⋀sa. s →β sa ⟹ Γ , Δ ⊢T sa : T1)
        t ° s →β sa
  thus Γ , Δ ⊢T sa : T2
    apply(safe)
    apply(clarsimp simp add: subst-type)
    apply(frule struct-subst-type-command
      [where ?Δ1.0 = Δ⟨0:T1 → T2⟩ and ?Δ = Δ and ?α = 0
       and ?T1.0 = T1 → T2 and ?E = ◊ • s and ?U = T2])
    apply(fastforce)+
  done
  ...
```

Figure 3.1.10: The Type Preservation theorem in Isabelle, and proof of the application case.

Using values, I can define the notion of normal form. Intuitively, terms that are in normal form are those that cannot reduce anymore.

Definition 3.1.16 (Normal Form)**.** A term t is in normal form if:

- t is a value or
- $t = \mu : \rho.[\alpha]v$ for some value v, and μ-variable α

To prove the progress theorem, I first proved a lemma about normal forms [14]. In Isabelle, I chose to write the proof of this lemma as a structured proof to enhance readability (Appendix A.2).

Lemma 3.1.17.

1. Consider a λ-closed term t, such that $\Gamma; \Delta \vdash t : \rho$. If there is no term s such that $t \to s$, then t is in normal form.

2. Consider a λ-closed commad c, such that $\Gamma; \Delta \vdash c : \perp\!\!\!\perp$. For any term t and μ-variable β such that $c = [\beta]t$, if t cannot reduce anymore, then t is in normal form.

Proof. By mutual rule induction on the typing judgements $\Gamma; \Delta \vdash t : \rho$ and $\Gamma; \Delta \vdash c : \perp\!\!\!\perp$. Most cases are easy to prove using the assumptions, induction hypothesis, and Lemma 3.1.15 about values. I describe two of them:

(app) Assume:

$$\begin{aligned} &\Gamma; \Delta \vdash rs : \rho \\ &\forall\ u. \quad rs \not\to u \end{aligned}$$

where r and s are λ-closed. This means that $\Gamma; \Delta \vdash r : \sigma \to \rho$, for some type σ, and $\forall\ u.\ r \not\to u$. So we can apply the induction hypothesis for r. In the case where r is a value, we know from Lemma 3.1.15 that $r = \lambda : \sigma.v$ for some value v. Otherwise, $r = \mu : \sigma \to \rho.[\alpha]v$. In both cases we arrive at a contradiction because rs would reduce.

(activate) So far, I have omitted stating results for both terms and commands because they have a very similar formulation. I also glossed over the fact that the inductions performed were actually mutual inductions over terms and commands. In this lemma, however, I gave the formulation for commands explicitly because it is not obvious. The need for this formulation appears when trying to prove the current case.

Assume:

$$\begin{aligned} &\Gamma; \Delta \vdash \mu : \rho.c : \rho \\ &\forall\ u. \quad \mu : \rho.c \not\to u \end{aligned} \tag{3.1.11}$$

where $\mu : \rho.c$ is λ-closed. Therefore, $\Gamma; \Delta\langle 0 : \rho\rangle \vdash c : \perp\!\!\!\perp$. There exist some β and t such that $c = [\beta]t$, where t cannot reduce anymore, according to assumption 3.1.11. We can now apply the induction hypothesis for c to deduce that either:

– t is a value, and, since $\mu : \rho.c = \mu : \rho.[\beta]t$, we are done or
– $t = \mu : \sigma.[\gamma]v$ for some μ-variable γ and value v, where $\sigma = \Delta\langle 0 : \rho\rangle(\beta)$. In this case, $\mu : \rho.c = \mu : \rho.([\beta]\mu : \sigma.[\gamma]v)$, and we obtain a contradiction because this term can reduce by rule (μi).

□

Theorem 3.1.18 (Progress)**.** For any λ-closed term t, if $\Gamma; \Delta \vdash t : \rho$ then either:
- there exists a term s such that $t \to s$ or
- t is in normal form

Proof. The conclusion follows directly from Lemma 3.1.17 about normal forms. □

3.2 Extending the $\lambda\mu^{\mathbf{T}}$-calculus

In this section, I present two additions to the $\lambda\mu^{\mathbf{T}}$-calculus that I implemented in Isabelle. These are both extensions to my original project.

The first one is adding a new type of commands, and a $\perp$ type to represent falsity [6]. The resulting calculus is called $\lambda\mu^{\mathbf{T}}_{top}$. This extension is motivated by the desire to make $\lambda\mu^{\mathbf{T}}$ isomorphic to *full* classical logic.

$$\frac{}{\Gamma, A \vdash_C A; \Delta}\ (\text{id}) \qquad \frac{\Gamma \vdash_C ; A, \Delta}{\Gamma \vdash_C A; \Delta}\ (\text{activate}) \qquad \frac{\Gamma \vdash_C A; A, \Delta}{\Gamma \vdash_C ; A, \Delta}\ (\text{passivate})$$

$$\frac{\Gamma, A \vdash_C B; \Delta}{\Gamma \vdash_C A \to B; \Delta}\ (\to_i) \qquad \frac{\Gamma \vdash_C A \to B; \Delta \qquad \Gamma \vdash_C A; \Delta}{\Gamma \vdash_C B; \Delta}\ (\to_e) \qquad \frac{\Gamma \vdash_C \bot; \Delta}{\Gamma \vdash_C ; \Delta}\ (\bot_e)$$

Figure 3.2.1: Classical Natural Deduction [6].

The second one is adding boolean, product and sum types to $\lambda\mu^{\mathbf{T}}_{top}$. This makes it easier to write propositions as $\lambda\mu^{\mathbf{T}}_{top}$ types, and is a step towards making the μ**ML** language more similar to ML.

3.2.1 From $\lambda\mu^{\mathbf{T}}$ to $\lambda\mu^{\mathbf{T}}_{top}$

In Section 2.2.5, I showed that the $\lambda\mu^{\mathbf{T}}$-calculus is isomorphic to Minimal Classical Natural Deduction, which implements minimal classical logic. This means that it validates Peirce's Law but not the Double Negation Law. Consequently, Parigot [30] shows that the Double Negation Law can be proved in $\lambda\mu$, and hence also in $\lambda\mu^{\mathbf{T}}$, by a term that is *not closed*, but has a free μ-variable.

In principle, it is not a problem that some classical proofs can only be represented in $\lambda\mu^{\mathbf{T}}$ by terms with free μ-variables. The Progress theorem holds for terms that are not closed under μ-variables, so these terms are well-behaved as well. However, the μ**ML** interpreter only accepts closed terms, as it is the case in other programming languages. Non-closed terms are undesirable from a programming perspective because substituting them in an arbitrary context may cause capture, leading to unexpected behaviour. Moreover, two non-closed terms that represent the same proof would not be α-equivalent. In order to prove classical propositions using the interpreter, I would like their proofs to be closed $\lambda\mu^{\mathbf{T}}$ terms. Consequently, I attempted to extend $\lambda\mu^{\mathbf{T}}$ to be isomorphic with full classical logic.

To transform Minimal Classical Natural Deduction to Classical Natural Deduction, which implements full classical logic, Figure 3.2.1, Ariola and Herbelin show that it is enough to add an elimination rule for falsity [6]. They extend $\lambda\mu$ accordingly, adding a typing rule for the type falsity, $\bot$, and a new type of commands, $[\top]t$, to obtain the $\lambda\mu_{top}$-calculus.

I adopted the same approach for the $\lambda\mu^{\mathbf{T}}$-calculus. The new grammars of types and commands appear in Figure 3.2.2, together with the collected typing rules for $\lambda\mu^{\mathbf{T}}$.

In $[\top]t$, $\top$ can be regarded as a distinguished μ-variable. It behaves the same as μ-variables under substitution but it cannot be bound by any abstraction or used as a μ-binder. Therefore, it is never a free μ-variable and it cannot appear in a typing environment.

The typing rule (top) closely matches the rule $(\bot_e)$ from Classical Natural Deduction. Also, it is very similar to (passivate), reinforcing the point that $\top$ is just a special μ-variable.

$$\rho, \sigma, \tau ::= \mathtt{N} \mid \perp\!\!\!\perp \mid \sigma \to \tau \mid \bot$$

$$c, d ::= [\alpha]t \mid [\top]t$$

$$\frac{x : \rho \in \Gamma}{\Gamma; \Delta \vdash x : \rho}\ \text{(a) axiom} \qquad \frac{\Gamma, x : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda x : \sigma.t : \sigma \to \tau}\ \text{(b) lambda} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash ts : \tau}\ \text{(c) app}$$

$$\Gamma; \Delta \vdash 0 : \mathtt{N}\ \text{(d) zero} \qquad \frac{\Gamma; \Delta \vdash t : \mathtt{N}}{\Gamma; \Delta \vdash \mathtt{S}t : \mathtt{N}}\ \text{(e) suc}$$

$$\frac{\Gamma; \Delta \vdash r : \rho \qquad \Gamma; \Delta \vdash s : \mathtt{N} \to \rho \to \rho \qquad \Gamma; \Delta \vdash t : \mathtt{N}}{\Gamma; \Delta \vdash \mathtt{nrec}_\rho\ r\ s\ t : \rho}\ \text{(f) nrec}$$

$$\frac{\Gamma; \Delta, \alpha : \rho \vdash c : \perp\!\!\!\perp}{\Gamma; \Delta \vdash \mu\alpha : \rho.c : \rho}\ \text{(g) activate} \qquad \frac{\Gamma; \Delta \vdash t : \rho \qquad \alpha : \rho \in \Delta}{\Gamma; \Delta \vdash [\alpha]t : \perp\!\!\!\perp}\ \text{(h) passivate} \qquad \frac{\Gamma; \Delta \vdash t : \bot}{\Gamma; \Delta \vdash [\top]t : \perp\!\!\!\perp}\ \text{(i) top}$$

Figure 3.2.2: The new syntax in $\lambda\mu^{\mathbf{T}}_{top}$ and all its typing judgements.

3.2.2 Extending the Isabelle Formalisation of $\lambda\mu^{\mathbf{T}}$ to $\lambda\mu^{\mathbf{T}}_{top}$

In this section, I present the modifications I made to the $\lambda\mu^{\mathbf{T}}$ formalisation in order to incorporate the new features of $\lambda\mu^{\mathbf{T}}_{top}$. The following significant changes were made to the $\lambda\mu^{\mathbf{T}}$ definitions:

1. In the structural substitution:
$$t[\alpha := \beta E]$$
α or β can now be $\top$. Therefore, I decided to change the types of these two arguments to be *nat option*, rather than *nat*. *None* is used to represent $\top$ and *Some* γ is used to represent the μ-variable γ.

2. With regards to reduction, $\top$ behaves like a μ-variable. I only needed to add two new reduction rules for $\top$. The first one resembles the (μi) rule for commands. The second one enables reduction inside commands.
$$[\top]\mu : \bot.c \to\downarrow^0_\mu (c[\mathit{Some}\ 0 := \mathit{None}\ \square])$$
$$s \to t \implies [\top]s \to [\top]t$$

Most of the proofs I described in Section 3.1 were updated without difficulty to deal with the addition of $\bot$. In the case where the proof was carried out by rule induction on a typing judgement, I added a new case for $\bot$. Similarly, I added a case for $[\top]t$ to all structural inductions on commands.

Below, I highlight the changes I had to make to the Type Preservation and Progress theorems.

1. In the Type Preservation proof, I had to add the case $\Gamma, \Delta \vdash [\top]t : \perp\!\!\!\perp$. If $t = \mu : \bot.c$ then t reduces to $\downarrow^0_\mu (c[\mathit{Some}\ 0 := \mathit{None}\ \square])$. To prove this term is well-typed, I had to prove a structural substitution lemma for $\top$:

$$\Gamma, \Delta\langle\alpha : \bot\rangle \vdash t : \rho \implies \Gamma, \Delta \vdash \downarrow^\alpha_\mu (t[\mathit{Some}\ \alpha := \mathit{None}\ \square])$$

   This follows easily by rule induction on the typing judgement of t.

2. The proof of the Progress theorem is modified because the definition of normal form is extended: a term $\mu : \rho.[\top]v$, where v is a value, is also in normal form.

3.2.3 Adding Booleans, Products and Sums to $\lambda\mu^{\mathbf{T}}_{top}$

Following the example by Pierce [36] for the simply-typed λ-calculus, I extended $\lambda\mu^{\mathbf{T}}_{top}$ with more datatypes. The main points that have to be considered are outlined below:

1. Syntax. The boolean type, the product type and the sum type are added to the grammar:

$$\rho, \sigma, \tau ::= \ldots\ \texttt{Bool} \mid \sigma \times \tau \mid \sigma + \tau$$

   There are new kinds of terms to populate each of these types:

$$\begin{aligned} t, r, s ::= \ldots\ & \texttt{true} \mid \texttt{false} \mid \texttt{if} : \rho\ t\ \texttt{then}\ r\ \texttt{else}\ s \mid \\ & (\!|t, s|\!) : \rho \mid \pi_1 t \mid \pi_2 t \mid \\ & \texttt{inl} : \rho\ t \mid \texttt{inr} : \rho\ t \mid \texttt{case} : \rho\ t\ \texttt{of}\ \ \texttt{inl}\ x \Rightarrow s \mid \texttt{inr}\ y \Rightarrow r \end{aligned}$$

   Below is an explanation of the new syntax. In all cases, the type annotation ρ is the type of the whole expression.

   - The notation $(\!|t, s|\!)$ represents the pair formed by terms t and s. The term $\pi_1 t$ is the projection of the left element of the pair t. Similarly, $\pi_2 t$ is the right-projection.
   - The term `inl` $: \rho\ t$ is a sum, where t is the left component. Similarly for `inr`. The `case` construct does a case split on term t, which can be either `inl` or `inr`.

2. Typing rules. Under propositions-as-types, the sum type is isomorphic to disjunction and the product type to conjunction. Therefore, the typing rule (pair), in Figure 3.2.3, is very similar to the introduction rule for conjunction in Natural Deduction, and (proj1) and (proj2) correspond to conjunction elimination. Similarly for disjunction.

In my Isabelle formalisation, I extended the datatypes for types and terms similarly. Then, I added the typing rules from Figure 3.2.3 to the inductive predicate for typing.

In the `case` $: \rho\ t$ `of` `inl` $x \Rightarrow s \mid$ `inr` $y \Rightarrow r$ expression, variables x and y are λ-binders. They are similar to the binder in the λ-abstraction. Therefore, they do not appear in the Isabelle *Case* constructor. They are implicitly represented by the de Bruijn index 0 inside s and r respectively. So, in Isabelle `case` is represented as:

$$\frac{}{\Gamma;\Delta \vdash \texttt{true} : \texttt{Bool}}\ \text{(true)} \qquad \frac{}{\Gamma;\Delta \vdash \texttt{false} : \texttt{Bool}}\ \text{(false)}$$

$$\frac{\Gamma;\Delta \vdash t : \texttt{Bool} \qquad \Gamma;\Delta \vdash s : \rho \qquad \Gamma;\Delta \vdash r : \rho}{\Gamma;\Delta \vdash \texttt{if} : \rho\ t\ \texttt{then}\ s\ \texttt{else}\ r : \rho}\ \text{(if)}$$

$$\frac{\Gamma;\Delta \vdash t : \sigma \qquad \Gamma;\Delta \vdash s : \tau}{\Gamma;\Delta \vdash (\llparenthesis t, s \rrparenthesis : \sigma \times \tau) : \sigma \times \tau}\ \text{(pair)} \qquad \frac{\Gamma;\Delta \vdash t : \sigma \times \tau}{\Gamma;\Delta \vdash \pi_1 t : \sigma}\ \text{(proj1)} \qquad \frac{\Gamma;\Delta \vdash t : \sigma \times \tau}{\Gamma;\Delta \vdash \pi_2 t : \tau}\ \text{(proj1)}$$

$$\frac{\Gamma;\Delta \vdash t : \sigma}{\Gamma;\Delta \vdash (\texttt{inl} : \sigma + \tau\ t) : \sigma + \tau}\ \text{(inl)} \qquad \frac{\Gamma;\Delta \vdash t : \tau}{\Gamma;\Delta \vdash (\texttt{inr} : \sigma + \tau\ t) : \sigma + \tau}\ \text{(inr)}$$

$$\frac{\Gamma;\Delta \vdash t : \sigma + \tau \qquad \Gamma, x : \sigma;\Delta \vdash s : \rho \qquad \Gamma, y : \tau;\Delta \vdash r : \rho}{\Gamma;\Delta \vdash (\texttt{case} : \rho\ t\ \texttt{of}\ \ \texttt{inl}\ x \Rightarrow s \mid \texttt{inr}\ y \Rightarrow r) : \rho}\ \text{(case)}$$

Figure 3.2.3: The typing rules for boolean, products, and sums in $\lambda\mu^{\mathbf{T}}_{top}$.

Case type dBT dBT dBT (Case - - Of Inl⇒ -|Inr⇒ -)

3. Reduction rules. The most important reduction rules that I added to the Isabelle inductive predicate can be found in Figure 3.2.4. Most of the rules are the expected ones. The congruence rules are omitted.

   Interesting rules arise from the interaction with the μ-abstraction. To understand these better, one can look at `if` more carefully. Without taking into consideration μ-abstractions, the reduction rules for `if` are:

$$\texttt{if} : \rho\ \texttt{true}\ \ \texttt{then}\ s\ \texttt{else}\ r \rightarrow s$$
$$\texttt{if} : \rho\ \texttt{false}\ \ \texttt{then}\ s\ \texttt{else}\ r \rightarrow r$$
$$t \rightarrow u \implies \texttt{if} : \rho\ t\ \texttt{then}\ s\ \texttt{else}\ r \rightarrow \texttt{if} : \rho\ u\ \texttt{then}\ s\ \texttt{else}\ r$$

   An expression such as $\texttt{if} : \rho\ (\mu : \texttt{Bool}.[\alpha]v)\ \texttt{then}\ s\ \texttt{else}\ r$, where v is a value, is well-typed but cannot reduce using the three rules above. However, it is not in normal form either, because it is not a value, nor of the form $\mu : \rho.[\alpha]v$, where v is a value. Therefore, using only these reduction rules, Lemma 3.1.17, which says that terms that cannot reduce anymore are in normal form, is not valid anymore. To address this problem, I added rule 3.2.1, from Figure 3.2.4:

$$\texttt{if} : \rho\ (\mu : \sigma.c)\ \texttt{then}\ s\ \texttt{else}\ r \rightarrow \mu : \rho.(c[\texttt{Some}\ 0 := \texttt{Some}\ 0\ (\texttt{if} : \rho\ \Box\ \uparrow^0_\mu(s)\ \uparrow^0_\mu(r))])$$

   This rule is similar to rule ($\mu\texttt{N}$), for `nrec`. The reduction rules 3.2.2 through 3.2.8 address the same problem of reduction in the presence of a μ-abstraction.

   What is more, these rules justify the need of annotating `nrec`, `if`, pairs and `case` with their own type. The μ-abstraction on the right-hand-side of these reduction

$$\begin{gathered}
\texttt{if} : \rho \ \texttt{true then } s \ \texttt{else} \ r \to s \\
\texttt{if} : \rho \ \texttt{false then } s \ \texttt{else} \ r \to r \\
\texttt{if} : \rho \ (\mu : \sigma.c) \ \texttt{then} \ s \ \texttt{else} \ r \to \mu : \rho.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ (\texttt{if} : \rho \ \square \ \uparrow^0_\mu (s) \ \uparrow^0_\mu (r))]) \qquad (3.2.1)
\end{gathered}$$

$$\begin{aligned}
\pi_1((\!| t, s |\!) : \sigma \times \tau) &\to t \\
\pi_2((\!| t, s |\!) : \sigma \times \tau) &\to s \\
\pi_1(\mu : \sigma \times \tau.c) &\to \mu : \sigma.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ (\pi_1 \square)]) \qquad (3.2.2) \\
\pi_2(\mu : \sigma \times \tau.c) &\to \mu : \tau.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ (\pi_2 \square)]) \qquad (3.2.3)
\end{aligned}$$

$$\begin{aligned}
(\!| \mu : \sigma.c, \ s |\!) : \sigma \times \tau &\to \mu : \sigma \times \tau.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ ((\!| \square, \ \uparrow^0_\mu (s) |\!) : \sigma \times \tau)]) \qquad (3.2.4) \\
(\!| t, \ \mu : \tau.c |\!) : \sigma \times \tau &\to \mu : \sigma \times \tau.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ ((\!| \uparrow^0_\mu (t), \ \square |\!) : \sigma \times \tau)]) \qquad (3.2.5)
\end{aligned}$$

$$\begin{gathered}
\texttt{case} : \rho \ (\texttt{inl} : \sigma + \tau \ u) \ \texttt{of inl} \ x \Rightarrow s \mid \texttt{inr} \ y \Rightarrow r \to s[x \coloneqq u] \\
\texttt{case} : \rho \ (\texttt{inr} : \sigma + \tau \ u) \ \texttt{of inl} \ x \Rightarrow s \mid \texttt{inr} \ y \Rightarrow r \to r[y \coloneqq u] \\
\texttt{case} : \rho \ (\mu : \sigma + \tau.c) \ \texttt{of inl} \ x \Rightarrow s \mid \texttt{inr} \ y \Rightarrow r \\
\to \mu : \rho.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ (\texttt{case} : \rho \ \square \ \texttt{of inl} \ x \Rightarrow \uparrow^0_\mu (s) \mid \texttt{inr} \ y \Rightarrow \uparrow^0_\mu (r))]) \qquad (3.2.6) \\
\texttt{inl} : \sigma + \tau \ (\mu : \sigma.c) \to \mu : \sigma + \tau.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ (\texttt{inl} : \sigma + \tau \ \square)]) \qquad (3.2.7) \\
\texttt{inr} : \sigma + \tau \ (\mu : \tau.c) \to \mu : \sigma + \tau.(c[\texttt{Some } 0 \coloneqq \texttt{Some } 0 \ (\texttt{inr} : \sigma + \tau \ \square)]) \qquad (3.2.8)
\end{gathered}$$

Figure 3.2.4: Reduction rules for booleans, products, and sums in $\lambda\mu^{\mathbf{T}}_{top}$.

rules needs to be annotated with the type of the expression on the left-hand-side. This is required to make reduction preserve typing. Therefore, the type on the left-hand side needs to be known when the reduction is performed. It is customary to annotate `inl` and `inr` with their types to obtain uniqueness of typing.

4. Extending the Isabelle proofs. This took a long time because a lot of new cases appeared. For example, 9 new typing rules were added. Therefore, I had to add 9 cases to every structured proof done by rule induction on a typing judgement, for example the Structural Substitution lemma, Type Preservation, and Lemma 3.1.17 about normal forms.

   Before updating the proof of the Progress Theorem, I had to update the definition of values. I added 5 new types of values, similar to those added to the simply-typed λ-calculus:

$$v \coloneqq \ldots\ \texttt{true} \mid \texttt{false} \mid (\!|v1, v2|\!) : \rho \mid \texttt{inl} : \rho\ v \mid \texttt{inr} : \rho\ v$$

   where v, $v1$, and $v2$ are values. As a result, Lemma 3.1.15, which specifies which values each type is inhabited by, had to be extended.

3.3 The μML Language

In this section, I describe the language μ**ML**, which is based on $\lambda\mu^{\mathbf{T}}_{top}$ with booleans, products and sums. Additionally, I explain how I implemented an interpreter for μ**ML** in OCaml. Over the course of the project, I incrementally added features to μ**ML**, as I was extending the $\lambda\mu^{\mathbf{T}}$ Isabelle formalisation. However, I only describe the final version of the language here.

The syntax of μ**ML** mirrors closely the syntax of the $\lambda\mu^{\mathbf{T}}_{top}$-calculus. It is given it Figure 3.3.1, where I also make explicit the correspondence between the syntax of the language and the syntax of the calculus.

3.3.1 Typing and Reduction in μML

The typing rules in μ**ML** are the same as in $\lambda\mu^{\mathbf{T}}_{top}$, and the reduction rules of the two are very similar. As a result, I was able to use my Isabelle formalisation to automatically generate OCaml code [18] for the μ**ML** type-inference and reduction functions. This means that the Type Preservation and Progress proofs, as well as all other proofs I did in Isabelle, transfer to μ**ML**, assuming that the code generation facility works correctly. Under this assumption, the interpreter is guaranteed to perform type inference and reduction correctly.

Given environments Γ and Δ, the type of any term or command in $\lambda\mu^{\mathbf{T}}_{top}$ is unique. Therefore, the Isabelle inductive predicate that defines the typing relation can be turned into a function from environments and terms to types. This process creates a naive type-inference function for the interpreter.

Generating code for the reduction function proved more complicated because the inductive predicate that defines the reduction relation does not have a functional form. At

$\mathit{type} ::=$	$\texttt{nat}$	$(\texttt{N})$
	$\mid \texttt{comm}$	$(\perp\!\!\!\perp)$
	$\mid \mathit{type} \to \mathit{type}$	(function type)
	$\mid \texttt{bot}$	$(\perp)$
	$\mid \texttt{bool}$	$(\texttt{Bool})$
	$\mid \mathit{type} * \mathit{type}$	(product type)
	$\mid \mathit{type} + \mathit{type}$	(sum type)
$\mathit{expr} ::=$	x	(λ-variables)
	$\mid n$	$(\texttt{S}\underline{n})$
	$\mid \texttt{suc}\ \mathit{expr}$	$(\texttt{S}t)$
	$\mid \texttt{fun}(x : \mathit{type}) \to \mathit{expr}\ \texttt{end}$	$(\lambda x : \rho.t)$
	$\mid (\mathit{expr})\ (\mathit{expr})$	(application)
	$\mid \texttt{bind}(a : \mathit{type}) \to \mathit{command}\ \texttt{end}$	$(\mu a : \rho.c)$
	$\mid \texttt{nrec} : (\mathit{type}) \to (\mathit{expr},\ \mathit{expr},\ \mathit{expr})\ \texttt{end}$	$(\texttt{nrec}_\rho\ r\ s\ t)$
	$\mid \texttt{true}$	
	$\mid \texttt{false}$	
	$\mid \texttt{if} : \mathit{type}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr}\ \texttt{end}$	$(\texttt{if} : \rho\ t\ \texttt{then}\ s\ \texttt{else}\ r)$
	$\mid \texttt{proj1}(\mathit{expr})$	$(\pi_1 t)$
	$\mid \texttt{proj2}(\mathit{expr})$	$(\pi_2 t)$
	$\mid \{\mathit{expr}, \mathit{expr}\} : \mathit{type}$	$((\!\lvert t, s \rvert\!) : \rho)$
	$\mid \texttt{inl} : \mathit{type}(\mathit{expr})$	$(\texttt{inl} : \rho\ t)$
	$\mid \texttt{inr} : \mathit{type}(\mathit{expr})$	$(\texttt{inr} : \rho\ t)$
	$\mid \texttt{case} : \mathit{type}\ \mathit{expr}\ \texttt{of}\ \ \texttt{inl}\ x \to \mathit{expr} \mid \texttt{inr}\ y \to \mathit{expr}\ \texttt{end}$	$(\texttt{case} : \rho\ t\ \texttt{of}\ \ \texttt{inl}\ x \Rightarrow s \mid \texttt{inr}\ y \Rightarrow r)$
$\mathit{command} ::=$	$[a].\mathit{expr}$	$([a]t)$
	$\mid [\texttt{abort}].\mathit{expr}$	$([\top]t)$

where x ranges over λ-variables and a over μ-variables, and n is a natural number.

Figure 3.3.1: The syntax of the μ**ML** language.

lemma *red-in-beta*:
$\textit{red-term } t = \textit{Some } u \Longrightarrow t \rightarrow_\beta u$
$\bigwedge d.\ \textit{red-command } c = \textit{Some } d \Longrightarrow c\ {}_C{\rightarrow_\beta}\ d$

lemma *beta-in-red*:
$\textit{red-term } t = \textit{None} \Longrightarrow \forall\ u.\ \neg(t \rightarrow_\beta u)$
$\textit{red-command } c = \textit{None} \Longrightarrow \forall\ d.\ \neg(c\ {}_C{\rightarrow_\beta}\ d)$

Figure 3.3.2: Isabelle lemmas that prove the correct behaviour of the reduction function.

some point, there may be different one-step reductions that a term can undergo, since the reduction strategy in $\lambda\mu^{\mathbf{T}}_{top}$ resembles full β-reduction

To choose a reduction order for the interpreter, while preserving the same rules, I had to define a separate reduction function in Isabelle that I then exported to OCaml (Appendix A.3). This is a rather long-winded solution, but I arrived at it after a considerable amount of experimentation with other methods that did not work.

This method itself posed some difficulties. These were due to an inefficiency in the Isabelle **function** package [21] when dealing with large function definitions. As a result, I was not able to use overlapping patterns in the definition, which complicated it considerably.

Additionally, I had to prove that this reduction function has the desired behaviour compared to the $\lambda\mu^{\mathbf{T}}_{top}$ reduction relation. To show this, I proved two lemmas which appear in Figure 3.3.2. Informally, the first one says that if a term t can reduce to u using the reduction function, then it can also reduce to u using the reduction relation. So the reduction function does not perform any reductions that are not permitted.

The second lemma says that if a term t cannot reduce anymore using the reduction function, then it cannot reduce using the reduction relation either. Coupled with Lemma 3.1.17, which says that a $\lambda\mu^{\mathbf{T}}_{top}$ term that cannot reduce anymore is in normal form, it follows that the reduction function reduces terms to a $\lambda\mu^{\mathbf{T}}_{top}$ normal form.

Proving these two lemmas was not included in the original plan of the project. I carried out both proofs as structured proofs, by mutual structural induction on t and c. The proofs were not especially difficult but each case required manual steps, making these two proofs long.

3.3.2 The Front-End of the μML Interpreter

I implemented the remaining components of the interpreter, lexer, parser, and pretty-printer, in OCaml. For this, I used OCamlLex and Menhir respectively. These are standard lexer and parser generators for OCaml [25].

The μ**ML** code is parsed into an abstract syntax tree (AST) represented by two OCaml variant types. One variant type is used for both terms and commands, and one for μ**ML** types. A function is used to attach a de Bruijn index to each variable in the AST. Then, two mutually recursive functions translate terms and commands to the $\lambda\mu^{\mathbf{T}}_{top}$ AST exported from Isabelle. There is a similar function for types.

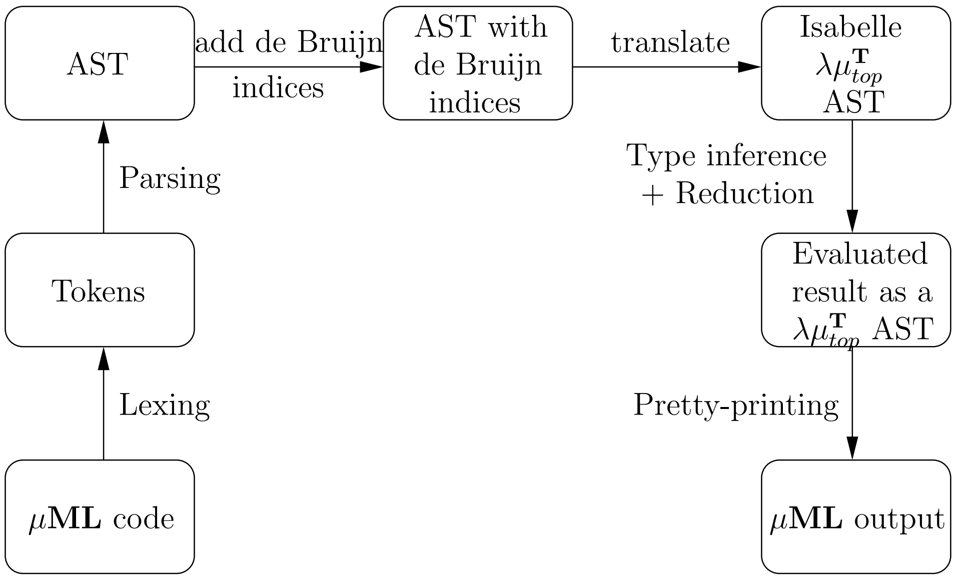
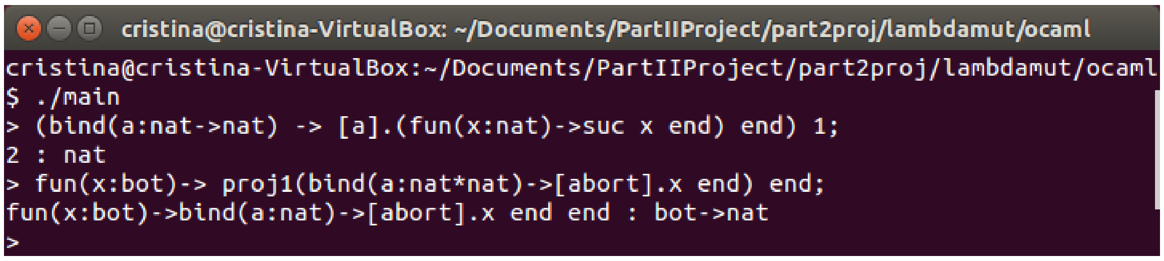

AST
add de Bruijn indices
AST with de Bruijn indices
translate
Isabelle $\lambda\mu^{\mathbf{T}}_{top}$ AST
Parsing
Type inference + Reduction
Tokens
Evaluated result as a $\lambda\mu^{\mathbf{T}}_{top}$ AST
Lexing
Pretty-printing
μ**ML** code
μ**ML** output

Figure 3.3.3: Diagram of the stages in the interpreter.


cristina@cristina-VirtualBox: ~/Documents/PartIIProject/part2proj/lambdamut/ocaml

```
cristina@cristina-VirtualBox:~/Documents/PartIIProject/part2proj/lambdamut/ocaml
$ ./main
> (bind(a:nat->nat) -> [a].(fun(x:nat)->suc x end) end) 1;
2 : nat
> fun(x:bot)-> proj1(bind(a:nat*nat)->[abort].x end) end;
fun(x:bot)->bind(a:nat)->[abort].x end end : bot->nat
>
```

Figure 3.3.4: Example usage of the μ**ML** REPL. In the first expression, the `bind` around the function is contracted, and then the function applied to argument 1. In the second expression, the `bind` is pulled out of the projection constructor.

This $\lambda\mu^{\mathbf{T}}_{top}$ AST can be given as an argument to the type-inference and reduction functions generated by Isabelle. The final result of the reduction, and the type of the term are then pretty-printed. This whole process is illustrated in Figure 3.3.3.

A user has two ways of interacting with the μ**ML** interpreter. A read-evaluate-print loop (REPL) can be started from the command line. The user types a term to be evaluated. The result is printed out together with its type. An example is given in Figure 3.3.4. To allow editing of keyboard input and persistent history, I wrapped my REPL into the `rlwrap` Unix utility [22].

The second interaction mode is by invoking the interpreter's top-level module with a μ**ML** source file as input. The results of the evaluation are written to a specified output file. The input file can contain multiple expressions to be evaluated. This facility is useful when a lot of expressions need to be evaluated.

Chapter 4

Evaluation

This chapter starts with an overview of the work undertaken. I then show how the μ**ML** interpreter can be used to prove classical propositions, which is the main motivation for implementing it. Next, I describe the testing I performed for the interpreter and the $\lambda\mu^{\mathbf{T}}_{top}$ formalisation. Finally, I compare the performance of the interpreter with that of the Poly/ML interpreter.

4.1 Work Completed

In addition to implementing the core of the project, I had time to work on several extensions. These are all described in the following two subsections.

4.1.1 Project Core

The first goal of the project was to become familiar with Isabelle. The material in *Concrete Semantics* [26] provided a useful introduction, but by no means an exhaustive one. Frequently during the project, I had to research new Isabelle features that would allow me to complete my task, usually in the Isabelle reference manual [42], or tutorials [18, 21, 27]. Overall, I acquired enough Isabelle knowledge to allow me to complete the rest of the project.

Most of my time was spent working on the Isabelle $\lambda\mu^{\mathbf{T}}$ formalisation and proofs. Understanding how to use de Bruijn indices correctly in $\lambda\mu^{\mathbf{T}}$ proved to be time-consuming. In the end, I managed to formalise all necessary definitions in Isabelle and complete the two main proofs that I set out to do, Type Preservation and Progress.

I defined the μ**ML** language based on $\lambda\mu^{\mathbf{T}}$, and implemented an interpreter in OCaml for it. The typing and reduction functions of the interpreter were automatically generated from the $\lambda\mu^{\mathbf{T}}$ Isabelle formalisation. Doing code generation in Isabelle for the reduction function proved more difficult than I expected. The lexer, parser, pretty-printer, and the code needed to combine these components was written in OCaml.

Given the results described so far, all the success criteria in the project proposal have been met.

4.1.2 Extensions

The extensions I implemented are as follows:

1. I extended the $\lambda\mu^{\mathbf{T}}$-calculus, to $\lambda\mu^{\mathbf{T}}_{top}$ to make it isomorphic to full classical logic, when considering only closed terms. This extension was not included in the project proposal because I only discovered later that this would be desirable.

2. I added more datatypes to $\lambda\mu^{\mathbf{T}}_{top}$, namely booleans, products and sums. This addition, similarly to the previous one, involved extending all my Isabelle definitions and proofs, and the μ**ML** interpreter accordingly.

3. The type annotations in $\lambda\mu^{\mathbf{T}}_{top}$ make type inference straightforward. Therefore, the typing function exported from Isabelle, and used in the interpreter, can already perform type inference, rather than type checking, as the proposal stipulated.

4. I used the interpreter to prove several classical propositions: Peirce's Law, the Double Negation Law, the equivalence between conjunction and disjunction respectively, and their classical definitions.

4.2 Proving Classical Propositions using the μML Interpreter

This section contains some examples of classical proofs given as μ**ML** terms, which can be type-checked by the interpreter. Since μ**ML** is not polymorphic, I replaced type variables with arbitrary concrete types. Their choice does not influence the validity of the proof.

1. A $\lambda\mu^{\mathbf{T}}_{top}$ term, whose type is the Double Negation Law, $((\tau \to \bot) \to \bot) \to \tau$, is given below. It is similar to the one proposed by Parigot in $\lambda\mu$ [30]:

   $$\lambda y : ((\tau \to \bot) \to \bot).\mu\alpha : \tau.[\top](y\ \lambda x : \tau.\mu\beta : \bot.[\alpha]x)$$

   The equivalent μ**ML** term is therefore:

   ```
   fun(y:(nat->bot)->bot)->
      bind(a:nat)->
          [abort].(y (fun(x:nat)->bind(b:bot)->[a].x end end))
      end
   end
   ```

   The interpreter gives this the correct type: `((nat->bot)->bot)->nat`.

2. A $\lambda\mu$ term that proves Peirce's Law, $((\sigma \to \tau) \to \sigma) \to \sigma$, is also given by Parigot [30]:

   $$\lambda x : (\sigma \to \tau) \to \sigma.\mu\alpha : \sigma.[\alpha](x\ \lambda y : \sigma.\mu\beta : \tau.[\alpha]y)$$

The corresponding μ**ML** term is:

```
fun(x:(nat->comm)->nat)->
   bind(a:nat)->
       [a].(x (fun(y:nat)->bind(b:comm)->[a].y end end))
   end
end
```

of type `((nat->comm)->nat)->nat`, as expected.

In classical logic, conjunction and disjunction can be defined in terms of implication and negation as:

$$A \wedge B \equiv \neg(A \to \neg B)$$
$$A \vee B \equiv \neg A \to B$$

I showed that the built-in conjunction operator, that can be added to Classical Natural Deduction, is equivalent to the definition above. To do this, I looked for $\lambda\mu^{\mathbf{T}}_{top}$ terms that prove the propositions:

$$A \wedge B \to \neg(A \to \neg B)$$
$$\neg(A \to \neg B) \to A \wedge B$$

I proved a similar result for disjunction. By definition, consider $\neg A$ to be the same as $A \to \bot$.

3. Conjunction:

- In $\lambda\mu^{\mathbf{T}}_{top}$ extended with booleans, products and sums, a term that has type $A \times B \to ((A \to (B \to \bot)) \to \bot)$ is:

$$\lambda x : A \times B.\lambda y : A \to B \to \bot.(y\ (\pi_1 x)\ (\pi_2 x))$$

  The corresponding μ**ML** term is:

```
fun(x:nat*comm)->
   fun(y:nat->comm->bot)-> (y (proj1(x)) (proj2(x)))
   end
end
```

  of type `nat*comm->(nat->comm->bot)->bot`, as expected.

- Finding a term of type $((A \to (B \to \bot)) \to \bot) \to A \times B$ is more involved:

$$\lambda x : (A \to B \to \bot) \to \bot.\mu\alpha : A \times B.[\top](x\ \lambda y : A.\lambda z : B.\mu\beta : \bot.[\alpha](\!|y, z|\!))$$

  Its typing derivation is given in Figure 4.2.1.

$$
\begin{prooftree}
\AxiomC{$\ldots$}
\AxiomC{}
\RightLabel{(axiom)}
\UnaryInfC{$\Gamma, y : A, z : B; \Delta, \beta : \bot \vdash y : A$}
\AxiomC{}
\RightLabel{(axiom)}
\UnaryInfC{$\Gamma, y : A, z : B; \Delta, \beta : \bot \vdash z : B$}
\RightLabel{(pair)}
\BinaryInfC{$\Gamma, y : A, z : B; \Delta, \beta : \bot \vdash (\!|y, z|\!) : A \times B$}
\RightLabel{(passivate)}
\UnaryInfC{$\Gamma, y : A, z : B; \Delta, \beta : \bot \vdash [\alpha](\!|y, z|\!) : \perp\!\!\!\perp$}
\RightLabel{(activate)}
\UnaryInfC{$\Gamma, y : A, z : B; \Delta \vdash \mu\beta : \bot.[\alpha](\!|y, z|\!) : \bot$}
\RightLabel{(lambda)}
\UnaryInfC{$\Gamma, y : A; \Delta \vdash \lambda z : B.\mu\beta : \bot.[\alpha](\!|y, z|\!) : B \to \bot$}
\RightLabel{(lambda)}
\UnaryInfC{$\Gamma; \Delta \vdash \lambda y : A.\lambda z : B.\mu\beta : \bot.[\alpha](\!|y, z|\!) : A \to B \to \bot$}
\RightLabel{(app)}
\BinaryInfC{$x : (A \to B \to \bot) \to \bot; \alpha : A \times B \vdash (x\ \lambda y : A.\lambda z : B.\mu\beta : \bot.[\alpha](\!|y, z|\!)) : \bot$}
\RightLabel{(top)}
\UnaryInfC{$x : (A \to B \to \bot) \to \bot; \alpha : A \times B \vdash [\top](x\ \lambda y : A.\lambda z : B.\mu\beta : \bot.[\alpha](\!|y, z|\!)) : \perp\!\!\!\perp$}
\RightLabel{(activate)}
\UnaryInfC{$x : (A \to B \to \bot) \to \bot; \vdash \mu\alpha : A \times B.[\top](x\ \lambda y : A.\lambda z : B.\mu\beta : \bot.[\alpha](\!|y, z|\!)) : A \times B$}
\RightLabel{(lambda)}
\UnaryInfC{$\vdash \lambda x : (A \to B \to \bot) \to \bot.\mu\alpha : A \times B.[\top](x\ \lambda y : A.\lambda z : B.\mu\beta : \bot.[\alpha](\!|y, z|\!)) : ((A \to B \to \bot) \to \bot) \to A \times B$}
\end{prooftree}
$$

where $\Gamma = \{x : (A \to B \to \bot) \to \bot\}$ and $\Delta = \{\alpha : A \times B\}$.

Figure 4.2.1: Proof of the proposition $((A \to B \to \bot) \to \bot) \to A \wedge B$ as a $\lambda\mu^{\mathbf{T}}_{top}$ term.

In μ**ML** this is:

```
fun(x:(nat->comm->bot)->bot)->
   bind(a:nat*comm)->
       [abort].(x
               (fun(y:nat)->
                   fun(z:comm)->
                      bind(b:bot)->
                          [a].{y, z}:nat*comm
                      end
                   end
                end))
   end
end
```

and has type `((nat->comm->bot)->bot)->nat*comm`.

4. Disjunction:

   - A $\lambda\mu^{\mathbf{T}}_{top}$ term of type $A + B \to ((A \to \bot) \to B)$ is:

     $\lambda x : A + B.\lambda y : A \to \bot.(\texttt{case} : B\ x\ \texttt{of}\ \texttt{inl}\ z \Rightarrow \mu\alpha : B.[\top](y\ z) \mid \texttt{inr}\ w \Rightarrow w)$

with the corresponding μ**ML** term:

```
fun(x:nat+comm)->
   fun(y:nat->bot)->
      case:comm x of inl z->bind(a:comm)-> [abort].(y z) end |
                     inr w->w
      end
   end
end
```

of type `nat+comm->(nat->bot)->comm`.

- For the inverse implication, $((A \to \bot) \to B) \to A + B$, a term of this type is:

$$\lambda x : (A \to \bot) \to B.\mu\alpha : A + B.[\top](\lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)) \\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A + B\ z)))$$

The typing derivation is given in Figure 4.2.2.
The corresponding μ**ML** term is:

```
fun(x:(nat->bot)->comm)->
   bind(a:nat+comm)->
       [abort].((fun(y:comm)->
                   bind(b:bot)->
                       [a].(inr:nat+comm(y))
                   end
                end)
               (x (fun(z:nat)->
                      bind(c:bot)->
                          [a].(inl:nat+comm(z))
                      end
                   end))
              )
   end
end
```

of type `((nat->bot)->comm)->nat+comm`.

4.3 Unit Tests for the Isabelle $\lambda\mu^{\mathbf{T}}$ Formalisation

The Isabelle formalisation of $\lambda\mu^{\mathbf{T}}$ is accompanied by proofs, the main ones being Type Preservation and Progress. As a result, unit tests are not necessary to ensure that the formalisation behaves correctly.

However, writing the definitions of the formalisation took a lot of time, and I could only do the proofs after this was complete. In the meantime, to get some assurance that

$$\dfrac{}{\Gamma, y : B; \Delta, \beta : \bot \vdash y : B}\,(\text{axiom})$$

$$\dfrac{\Gamma, y : B; \Delta, \beta : \bot \vdash y : B}{\Gamma, y : B; \Delta, \beta : \bot \vdash (\mathtt{inr}\!:\!A{+}B\ y)\!:\!A{+}B}\,(\text{inr})$$

$$\dfrac{\Gamma, y : B; \Delta, \beta : \bot \vdash (\mathtt{inr}\!:\!A{+}B\ y)\!:\!A{+}B}{\Gamma, y\!:\!B; \Delta, \beta\!:\!\bot \vdash [\alpha](\mathtt{inr}\!:\!A{+}B\ y)\!:\!\bot\!\!\!\bot}\,(\text{passivate})$$

$$\dfrac{\Gamma, y\!:\!B; \Delta, \beta\!:\!\bot \vdash [\alpha](\mathtt{inr}\!:\!A{+}B\ y)\!:\!\bot\!\!\!\bot}{\Gamma, y\!:\!B; \Delta \vdash \mu\beta\!:\!\bot.[\alpha](\mathtt{inr}\!:\!A{+}B\ y)\!:\!\bot}\,(\text{activate})$$

$$\dfrac{\Gamma, y\!:\!B; \Delta \vdash \mu\beta\!:\!\bot.[\alpha](\mathtt{inr}\!:\!A{+}B\ y)\!:\!\bot}{\Gamma; \Delta \vdash \lambda y\!:\!B.\mu\beta\!:\!\bot.[\alpha](\mathtt{inr}\!:\!A{+}B\ y)\!:\!B \to \bot}\,(\text{lambda})$$

$$\dfrac{}{\Gamma, z : A; \Delta, \gamma : \bot \vdash z : A}\,(\text{axiom})$$

$$\dfrac{\Gamma, z : A; \Delta, \gamma : \bot \vdash z : A}{\Gamma, z : A; \Delta, \gamma : \bot \vdash (\mathtt{inl}\!:\!A{+}B\ z)\!:\!A{+}B}\,(\text{inl})$$

$$\dfrac{\Gamma, z : A; \Delta, \gamma : \bot \vdash (\mathtt{inl}\!:\!A{+}B\ z)\!:\!A{+}B}{\Gamma, z : A; \Delta, \gamma\!:\!\bot \vdash [\alpha](\mathtt{inl}\!:\!A{+}B\ z)\!:\!\bot\!\!\!\bot}\,(\text{passivate})$$

$$\dfrac{\Gamma, z : A; \Delta, \gamma\!:\!\bot \vdash [\alpha](\mathtt{inl}\!:\!A{+}B\ z)\!:\!\bot\!\!\!\bot}{\Gamma, z\!:\!A; \Delta \vdash \mu\gamma\!:\!\bot.[\alpha](\mathtt{inl}\!:\!A{+}B\ z)\!:\!\bot}\,(\text{activate})$$

$$\dfrac{\Gamma, z\!:\!A; \Delta \vdash \mu\gamma\!:\!\bot.[\alpha](\mathtt{inl}\!:\!A{+}B\ z)\!:\!\bot}{\Gamma; \Delta \vdash \lambda z\!:\!A.\mu\gamma\!:\!\bot.[\alpha](\mathtt{inl}\!:\!A{+}B\ z)\!:\!A \to \bot}\,(\text{lambda})$$

$$\dfrac{\ldots \qquad \Gamma; \Delta \vdash \lambda z\!:\!A.\mu\gamma\!:\!\bot.[\alpha](\mathtt{inl}\!:\!A{+}B\ z)\!:\!A \to \bot}{\Gamma; \Delta \vdash x\ \lambda z\!:\!A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A + B\ z) : B}\,(\text{app})$$

$$\dfrac{\Gamma; \Delta \vdash \lambda y\!:\!B.\mu\beta\!:\!\bot.[\alpha](\mathtt{inr}\!:\!A{+}B\ y)\!:\!B \to \bot \qquad \Gamma; \Delta \vdash x\ \lambda z\!:\!A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A + B\ z) : B}{x : (A \to \bot) \to B; \alpha : A + B \vdash \lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A + B\ z)) : \bot}\,(\text{app})$$

$$\dfrac{x : (A \to \bot) \to B; \alpha : A + B \vdash \lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A + B\ z)) : \bot}{x : (A \to \bot) \to B; \alpha : A + B \vdash [\top](\lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A{+}B\ z))) : \bot\!\!\!\bot}\,(\text{top})$$

$$\dfrac{x : (A \to \bot) \to B; \alpha : A + B \vdash [\top](\lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A{+}B\ z))) : \bot\!\!\!\bot}{x : (A \to \bot) \to B \vdash \mu\alpha : A + B.[\top](\lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl}\!:\!A{+}B\ z))) : A{+}B}\,(\text{activate})$$

$$\dfrac{x : (A \to \bot) \to B \vdash \mu\alpha : A + B.[\top](\lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl}\!:\!A{+}B\ z))) : A{+}B}{\vdash \lambda x : (A \to \bot) \to B.\mu\alpha : A + B.[\top](\lambda y : B.\mu\beta : \bot.[\alpha](\mathtt{inr} : A + B\ y)\ (x\ \lambda z : A.\mu\gamma : \bot.[\alpha](\mathtt{inl} : A + B\ z))) : ((A \to \bot) \to B) \to A + B}\,(\text{lambda})$$

where $\Gamma = \{x : (A \to \bot) \to B\}$ and $\Delta = \{\alpha : A + B\}$

Figure 4.2.2: Proof of the proposition $((A \to \bot) \to B) \to A \vee B$ as a $\lambda\mu^{\mathbf{T}}_{top}$ term.

the definitions were correct, I proved small lemmas that check their behaviour. These serve the same purpose as unit tests do in a traditional software development project.

Below is an example that tests the inductive predicate for typing:

$$\textbf{schematic-goal}\ \Gamma,\ \Delta \vdash_T \mu\ \textit{?T1} : (\textit{<0>}\ (\lambda\ T : (\textit{`0'}))) : \textit{?T2}$$
$$\textbf{by}\ \textit{force}$$

Isabelle unifies the variables *?T1* and *?T2* with the type $T \rightarrow T$, and proves that the typing judgement is valid.

The following lemma tests that structural substitution is capture-avoiding for λ-variables:

$$\textbf{lemma}\ (\lambda\ \textit{T1} : (\mu\ \textit{T2} : (\textit{<2>}\ (\textit{`2'}))))[(\textit{Some 1})=(\textit{Some 2})\ (\Diamond\ ^\bullet\ (\textit{`0'}))]^T$$
$$= (\lambda\ \textit{T1} : (\mu\ \textit{T2} : (\textit{<3>}\ ((\textit{`2'})^\circ(\textit{`1'})))))$$
$$\textbf{by}\ \textit{simp}$$

The free λ-variable 0, from the context ($\Box$ 0), is incremented by 1 when placed inside a λ-abstraction: $\lambda : T_1.\mu : T_2.[3](2\ 1)$.

I proved similar lemmas to test other cases of interest in the definitions of typing and structural substitution. In addition, I wrote such tests for the logical substitution function, the lifting functions, the functions that calculate the set of free variables in a term, and the reduction relation.

Figure 4.3.1 shows a test for the reflexive-transitive closure of the reduction relation. It proves that the $\lambda\mu^{\mathbf{T}}$ term:

$$\mu\alpha : \mathtt{N}.[\alpha]\mathtt{S}((\lambda f : \mathtt{N} \rightarrow \mathtt{N}.\mu\beta : \mathtt{N}.[\beta](f\ 0))\ (\lambda x : \mathtt{N}.\mu\gamma : \mathtt{N}.[\alpha]x))$$

reduces in multiple steps to 0. The proof is done using several lemmas, one for each step in the reduction. This term is given as an example in the original $\lambda\mu^{\mathbf{T}}$ paper [14] to illustrate the reduction rules for μ-abstractions.

lemma
  (μ *Nat* : (*<0>* (*S*
    ((λ (*Nat*$\rightarrow$*Nat*) : (μ *Nat* : (*<0>* ((*`0'*)$^\circ$*Zero*))))$^\circ$
     (λ *Nat* : (μ *Nat* : (*<1>* (*`0'*))))))))
  $\rightarrow_\beta{}^*$ *Zero*
  using *ex1 ex2 ex3 ex4 ex5 ex6 ex7 step-term* **by** *auto*

Figure 4.3.1: Unit test for the reduction relation.

4.4 Testing the μML Interpreter

The front-end of the μ**ML** interpreter is not verified. As a result, I had to write tests to check its behaviour, all of which pass successfully. As I added more features to μ**ML**, the suite of tests increased. To manage these, I used the OUnit library [4], which is a unit test framework for OCaml, similar to JUnit. I automated the tests using a `Makefile`.

The tests are organised in three categories, each of them covering each piece of μ**ML** syntax:

```
"basic syntax" >:: (fun _ -> assert_equal [
                      "4";
                      "111";
                      "(((((x b) f) e) w) g)";
                      "(x y)";
                      "bind(a:comm) -> [a].x end";
                      "nrec:(nat) -> (x, y, z) end";
                      "fun(x:(nat->comm)) -> x end"]
                      (test_frontend_ounit (open_in parser_basicsyntax))
```

Figure 4.4.1: OUnit test for the parser.

```
"fun4" >:: (fun _ -> assert_equal
    "(Lbd Nat (Lbd Nat (App (Lbd Nat (S (LVar 0))) (LVar 1))))"
    (test_translate_ounit (Fun ("x", TNat, (Fun ("y", TNat,
           (App ((Fun ("x", TNat, (Suc (Var "x")))), (Var "x"))))))) ));
```

Figure 4.4.2: OUnit test for the translation between the AST produced by the parser and the $\lambda\mu^{\mathbf{T}}_{top}$ AST.

1. *Tests for the parser.* These check that the lexer and parser work correctly together. Figure 4.4.1 shows an example. The input μ**ML** expressions are read from a file. They are each transformed into an abstract syntax tree (AST) by the lexer and parser. Each AST is pretty-printed, and the results compared against the ones specified in the test.

2. *Tests for the translation of the AST produced by the parser to the $\lambda\mu^{\mathbf{T}}_{top}$ AST exported from Isabelle.* This translation involves replacing variables with de Bruijn indices. Therefore, I focused on testing function abstractions, μ-abstractions and `case` statements because they introduce binders, which need to be handled carefully (for example: Figure 4.4.2).

3. *Tests for the whole interpreter.* This is a series of tests that checks that μ**ML** expressions are interpreted correctly. There is a set of such tests for each feature of the language. Figure 4.4.3 gives a test for booleans. These tests work similarly to the ones for the parser.

4.5 Performance of the μML Interpreter

Simple arithmetic operations can be encoded in $\lambda\mu^{\mathbf{T}}$, and therefore in μ**ML**, using the datatype for natural numbers and primitive recursion. These encodings are the standard ones from System **T** [41]. I used this kind of operations, namely addition, predecessor and factorial, to compare the performance of the μ**ML** interpreter with the Poly/ML 5.2 interpreter.

```
let booleans_tests = "tests for booleans" >:::[
     "basic expressions" >::
        (fun _ -> assert_equal [
             "true : bool\n";
             "false : bool\n";
             "3 : nat\n";
             "false : bool\n";
             "7 : nat\n";
             "9 : nat\n";
             "fun(x:bot)->bind(a:nat)->[abort].x end end : bot->nat\n"]
             (process_ounit (open_in evaluation_booleans)))
]
```

Figure 4.4.3: OUnit test for interpreting expressions involving booleans.

I ran each operation in both interpreters and measured the *real* time taken by each, as reported by the Unix `time` utility. The programs were timed in a VirtualBox running 64-bit Ubuntu 14.04.1 LTS, on a machine with an Intel i5 1.6 GHz CPU, running Windows 10 natively.

Addition can be represented in $\lambda\mu^{\mathbf{T}}$ by the term [14]:

$$+ := \lambda m : \mathtt{N}.\lambda n : \mathtt{N}.(\mathtt{nrec}_{\mathtt{N}}\ n\ (\lambda x : \mathtt{N}.\lambda y : \mathtt{N}.\mathtt{S}y)\ m)$$

If m is 0, the term $m + n$ reduces to n as expected. Otherwise, if m is of the form $\mathtt{S}\underline{m'}$, it reduces to:

$$\mathtt{S}\ (\mathtt{nrec}_{\mathtt{N}}\ n\ (\lambda x : \mathtt{N}.\lambda y : \mathtt{N}.\mathtt{S}y)\ \underline{m'})$$

Therefore, the term $+$ computes the sum of m and n. Notice that the number of one-step reductions needed is linear in m. This is because `nrec` needs to be unfolded m times. Therefore, the time complexity is $\mathcal{O}(m)$.

The corresponding μ**ML** expression for addition is:

```
fun(m:nat)->
   fun(n:nat)->
       nrec:(nat)->(n, (fun(x:nat)->fun(y:nat)-> suc y end end), m)
       end
   end
end
```

In ML, I defined a datatype for natural numbers, and an addition function for it. This function takes linear time in the size of its first argument, like the μ**ML** function does:

```
datatype nat = Zero
             | Suc of nat;
fun add Zero n = n
  | add (Suc m) n = Suc (add m n);
```

	μ**ML** Interpreter		Poly/ML 5.2	
Expression	Mean	Standard deviation	Mean	Standard deviation
$20 + 5$	26.16	3.94	16.93	3.97
$40 + 5$	118.37	7.48	35.46	6.62
`pred` 20	16.47	2.66	12.86	3.75
`pred` 40	88.73	8.59	25.46	6.58
`fact` 6	879.77	26.61	133.00	16.37

Figure 4.5.1: Time taken to evaluate expressions using the μ**ML** interpreter and Poly/ML 5.2, respectively. The values are given in ms. The run-time for each expression is averaged over 30 runs, discarding outliers.

I chose to represent addition in ML in this way in order to obtain a meaningful comparison with μ**ML**.

The time taken to compute $20 + 5$ and $40 + 5$ using these expressions is given in Figure 4.5.1. The run-time for each expression is averaged over 30 runs, discarding outliers, and the standard deviation is calculated. The values are given in ms.

For the operation $20 + 5$, the time taken by the two interpreters is of the same order of magnitude. However, when computing $40 + 5$, the time taken by μ**ML** more than quadruples. This should not be the case, taking into consideration the time complexity of the operation. Instead, the run-time should double, as it does in the case of Poly/ML.

The predecessor operation in $\lambda\mu^{\mathbf{T}}$, and hence μ**ML**, takes constant time. It can be encoded as:

$$\texttt{pred} := \lambda m : \texttt{N}.(\texttt{nrec}_{\texttt{N}}\ 0\ (\lambda x : \texttt{N}.\lambda y : \texttt{N}.x)\ m)$$
$$\texttt{pred}\ \texttt{S}\underline{n} \to^* (\lambda x : \texttt{N}.\lambda y : \texttt{N}.x)\ \underline{n}\ (\texttt{nrec}_{\texttt{N}}\ 0\ (\lambda x : \texttt{N}.\lambda y : \texttt{N}.x)\ \underline{n}) \to^* \underline{n}$$

In ML, a function with the same time complexity is:

```
fun pred Zero = Zero
  | pred (Suc n) = n;
```

The time taken for the two interpreters to perform `pred` 20 is comparable, Figure 4.5.1. For `pred` 40, ML takes double the time because printing the result using `Suc` constructors is linear in the size of the result. The same should be true for μ**ML**. However, some inefficiencies manifest themselves here, as in the case of addition.

Factorial is another operation that is very inefficient in $\lambda\mu^{\mathbf{T}}$ and μ**ML**. To define it, multiplication needs to be defined first [14]:

$$* := \lambda m : \texttt{N}.\lambda n : \texttt{N}.(\texttt{nrec}_{\texttt{N}}\ 0\ (\lambda x : \texttt{N}.\lambda y : \texttt{N}.n + y)\ m)$$

Here, $+$ is the addition operation defined previously. Since each addition, $n + y$, takes $\mathcal{O}(n)$ time, it means that multiplication takes $\mathcal{O}(mn)$ time. Factorial can be defined in terms of multiplication as follows:

$$\texttt{fact} := \lambda m : \texttt{N}.(\texttt{nrec}_{\texttt{N}}\ 1\ (\lambda x : \texttt{N}.\lambda y : \texttt{N}.(\texttt{S}x) * y)\ m)$$

It requires m multiplications, $1 * 2 * 3 * \ldots * m$, which are performed left-to-right. So each multiplication has the form $k * (k-1)!$, and therefore takes $\mathcal{O}(k!)$ time. As a result, the factorial operation takes $\mathcal{O}(\sum_{k=1}^{m} k!)$ time, which is super-exponential.

In ML, I defined a factorial function with the same time complexity as follows:

```
fun times Zero n = Zero
  | times m Zero = Zero
  | times (Suc m) n = add n (times m n);

fun fact Zero = (Suc Zero)
  | fact (Suc n) = times (Suc n) (fact n);
```

Examining Figure 4.5.1, it is clear that computing factorial in this way is not feasible in general. Even for a small number like 6, the μ**ML** run-time is almost 1s.

In conclusion, μ**ML** suffers from inefficiencies when performing arithmetic operations. A possible explanation is that the code generated by Isabelle for the reduction function is not as efficient as reduction in Poly/ML. This is expected since Poly/ML is a dedicated interpreter, which contains specific optimisations, while the Isabelle code generation facility is general-purpose, and not targeted towards performance.

Chapter 5

Conclusion

This chapter briefly reviews the contents of the project and the work completed. It also outlines some of the possible ways this project can be extended in the future.

5.1 Outcome of the Project

My project concerned the $\lambda\mu^{\mathbf{T}}$-calculus, an extension of the λ-calculus with control operators and natural numbers, that is isomorphic to classical logic under the propositions-as-types correspondence. One of the aims was to formalise $\lambda\mu^{\mathbf{T}}$ in Isabelle/HOL, using de Bruijn indices, and to prove Type Preservation and Progress, which has never been done before in an interactive theorem prover.

One of the challenges I faced was learning Isabelle as a complete beginner, sufficiently well to allow me to implement the project. Another challenge was the use of de Bruijn notation. This proved more time-consuming than expected. Consequently, if I were to attempt this project again, I would seriously consider using techniques similar to Nominal Isabelle [3] instead, in order to deal with α-equivalence. Despite these difficulties, I managed to complete the $\lambda\mu^{\mathbf{T}}$ formalisation and all the necessary proofs. Moreover, I extended this formalisation to $\lambda\mu^{\mathbf{T}}_{top}$, added a range of new datatypes, and updated all the proofs to reflect these changes.

A further goal was to define the μ**ML** language, based on $\lambda\mu^{\mathbf{T}}$, and to implement an interpreter for it in OCaml. To do this, I used Isabelle's code generation mechanism to automatically export type-inference and reduction functions to OCaml, from the $\lambda\mu^{\mathbf{T}}$ formalisation. Later, I moved to using the formalisation of $\lambda\mu^{\mathbf{T}}_{top}$ with datatypes for this purpose. This posed some difficulties in the case of the reduction function. It required experimentation with Isabelle's **function** package [21], because of an inefficiency I encountered in it, and two additional proofs, not foreseen in the project proposal. These obstacles were overcome successfully. Although the μ**ML** interpreter suffers from some inefficiencies, it achieves its main purpose, that of proving classical propositions.

Given the work described above, I can conclude that the project was a success. All the success criteria established in the project proposal have been met, and extensions implemented.

5.2 Future Work

Not a great deal of research has been done into the $\lambda\mu^{\mathbf{T}}$-calculus. Therefore, there is potential for a lot of interesting future work.

One natural extension to my project would be to prove confluence of the $\lambda\mu^{\mathbf{T}}$-calculus in Isabelle. The pen-and-paper proof is outlined in the original presentation of $\lambda\mu^{\mathbf{T}}$ [14]. It would be interesting to extend this proof to $\lambda\mu^{\mathbf{T}}_{top}$ with booleans, products and sums, which has not been explored before in the literature. As Geuvers *et al.* [14] observe, a long-term goal would be to extend $\lambda\mu^{\mathbf{T}}$ with dependent types, to increase expressivity.

As far as the μ**ML** language is concerned, one could add recursion, polymorphism and `let`-bindings. This would make it more powerful and bring it closer to ML. Corresponding changes would need to be made to the underlying calculus.

Such changes would allow μ**ML** to be used as a target language for the Isabelle code generation facility. The underlying logic used in Isabelle/HOL is classical, whereas ML's type-system is based on intuitionistic logic. Therefore, more programs could potentially be extracted to the extended variant of μ**ML** mentioned above than to ML. This includes extracting classical proofs to μ**ML** programs, rather than only exporting executable specifications.

Bibliography

[1] *Agda.* http://wiki.portal.chalmers.se/agda/pmwiki.php. Accessed: 2017-03-24.

[2] *Isabelle.* https://isabelle.in.tum.de/index.html. Accessed: 2016-10-15.

[3] *Nominal Isabelle.* https://nms.kcl.ac.uk/christian.urban/Nominal/. Accessed: 2017-03-15.

[4] *OUnit.* http://ounit.forge.ocamlcore.org/index.html. Accessed: 2017-01-15.

[5] *The Coq Proof Assistant.* https://coq.inria.fr/. Accessed: 2017-03-24.

[6] Ariola, Z.M., and Herbelin, H., *Minimal Classical Logic and Control Operators.* ICALP '03 (Proceedings), pages 871–885. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[7] Crolard, T., *A Confluent λ-calculus with a Catch/Throw Mechanism.* Journal of Functional Programming, **9**(*6*):625–647, 1999.

[8] de Bruijn, N.G., *Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, With Application to the Church-Rosser Theorem.* Indagationes Mathematicae (Proceedings), **75**(*5*):381–392, 1972.

[9] de Groote, P., *A CPS-translation of the λμ-calculus.* CAAP '94 (Proceedings), pages 85–99. Springer-Verlag, London, UK, 1994.

[10] Felleisen, M., Friedman, D.P., Kohlbecker, E., and Bruce, D., *A Syntactic Theory of Sequential Control.* Theoretical Computer Science, **52**(*3*):205–237, 1987.

[11] Fujita, K., *Explicitly Typed λμ-calculus for Polymorphism and Call-by-Value.* TLCA '99 (Proceedings), pages 162–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[12] Gentzen, G., *Investigations into Logical Deduction.* American Philosophical Quarterly, **1**(*4*):288–306, 1964.

[13] Gentzen, G., *Investigations into Logical Deduction: II.* American Philosophical Quarterly, **2**(*3*):204–218, 1965.

[14] Geuvers, H., Krebbers, R., and McKinna, J., *The* $\lambda\mu^{\boldsymbol{T}}$*-calculus.* Annals of Pure and Applied Logic, **164**(*6*):676–701, 2013.

[15] Gordon, M., *Proof, Language, and Interaction.* Chapter: From LCF to HOL: A Short History, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[16] Griffin, T.G., *A Formulae-as-type Notion of Control.* POPL '90 (Proceedings), pages 47–58. New York, NY, USA, 1990. ACM.

[17] Griffin, T.G., *Compiler Construction.* Lecture notes for the University of Cambridge, Lent 2016.

[18] Haftmann, F., *Code Generation from Isabelle/HOL Theories.* 2016.

[19] Howard, W.A., *The Formulæ-as-types Notion of Construction.* In Philippe De Groote, editor, The Curry-Howard Isomorphism. Academia, 1995.

[20] Johansson, I., *Der Minimalkalkül, ein reduzierter intutionistischer Formalismus.* Compositio mathematica, **4**(*1*):119–136, 1936.

[21] Krauss, A., *Defining Recursive Functions in Isabelle/HOL.* 2008.

[22] Lub, H., *rlwrap.* https://github.com/hanslub42/rlwrap, 2016.

[23] Luo, Z., *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, Inc., New York, NY, USA, 1994.

[24] Milner, R. *Logic for Computable Functions: Description of a Machine Implementation.* Technical report, Stanford, CA, USA, 1972.

[25] Minsky, Y., Madhavapeddy, A., and Hickey, J., *Real World OCaml: Functional Programming for the Masses.* O'Reilly, 2013.

[26] Nipkow, T., and Klein, G., *Concrete Semantics. A Proof Assistant Approach.* Springer, 2014.

[27] Nipkow, T., Paulson, L.C., and Wenzel, M., *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* 2016.

[28] Nipkow, T., *Winskel is (Almost) Right: Towards a Mechanized Semantics Textbook.* Formal Aspects of Computing, **10**(*2*):171–186, 1998.

[29] Ong, C.H.L., and Stewart, C.A., *A Curry-Howard Foundation for Functional Computation with Control.* POPL '97 (Proceedings), pages 215–227. New York, NY, USA, 1997. ACM.

[30] Parigot, M., *$\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction.* LPAR '92 (Proceedings), pages 190–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[31] Parigot, M., *Classical Proofs as Programs.* KGC '93 (Proceedings), pages 263–276. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[32] Parigot, M., *Strong Mormalization for Second Order Classical Natural Deduction.* LICS '93 (Proceedings), pages 39–46. 1993.

[33] Paulson, L.C., *Foundations of Computer Science.* Lecture notes for the University of Cambridge, Michaelmas 2014.

[34] Paulson, L.C., *Isabelle: The Next 700 Theorem Provers.* In Logic and Computer Science, **31,** pages 361–386, 1990.

[35] Paulson, L.C., and Blanchette, J.C., *Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers.* In The 8th International Workshop on the Implementation of Logics, 2010, Yogyakarta, Indonesia, October 9, 2011, pages 1–11, 2010.

[36] Pierce, B.C., *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[37] Pitts, A.M., *Nominal Logic, a First Order Theory of Names and Binding.* Information and Computation, **186**(*2*):165–193, 2003.

[38] Pitts, A.M., *Types.* Lecture notes for the University of Cambridge, Michaelmas 2016.

[39] Rehof, N.J., and Sørensen, M.H., *The λ_Δ-calculus.* TACS '94 (Proceedings), pages 516–542. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[40] Sewell, P., *Semantics of Programming Languages.* Lecture notes for the University of Cambridge, Michaelmas 2015.

[41] Sørensen, M.H., and Urzyczyin, P., *Lectures on the Curry-Howard Isomorphism.* Studies in Logic and the Foundations of Mathematics, **149**, Elsevier, 2006.

[42] Wenzel, M., *The Isabelle/Isar Reference Manual.* 2016.

[43] Winskel, G., *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993.

Appendix A

Isabelle Code

A.1 Type Preservation

```
theorem type-preservation:
  Γ, Δ ⊢T t : T ⟹ t →β s ⟹ Γ, Δ ⊢T s : T
  Γ, Δ ⊢C c : Command ⟹
     ∀ d. (c C→β d ⟶ Γ, Δ ⊢C d : Command)
proof(induct arbitrary: s
     rule: typing-dBT-typing- dBC.inducts)
  fix Γ :: nat ⇒ Types.type and x T Δ s
  assume Γ x = T
         LVar x →β s
  thus Γ , Δ ⊢T s : T
    by blast
next
  fix Γ Δ s
  assume Zero →β s
  thus Γ , Δ ⊢T s : Nat
    by blast
next
  fix Γ Δ t s
  assume Γ , Δ ⊢T t : Nat
         S t →β s
  and IH: ⋀s. t →β s ⟹ Γ , Δ ⊢T s : Nat
  thus Γ , Δ ⊢T s : Nat
    apply(safe)
    apply(frule struct-subst-type-command
      [where ?Δ1.0 = Δ⟨0:type.Nat⟩
       and ?Δ = Δ and ?α = 0
       and ?T1.0 = Nat and ?E = CSuc ◊
       and ?U = Nat])
    apply(fastforce)+
  done
next
  fix Γ Δ t T1 T2 s sa
  assume Γ , Δ ⊢T t : T1 → T2
         (⋀s. t →β s ⟹ Γ , Δ ⊢T s : T1 → T2)
         Γ , Δ ⊢T s : T1
         (⋀sa. s →β sa ⟹ Γ , Δ ⊢T sa : T1)
         t ° s →β sa
  thus Γ , Δ ⊢T sa : T2
    apply(safe)
    apply(clarsimp simp add: subst-type)
    apply(frule struct-subst-type-command
      [where ?Δ1.0 = Δ⟨0:T1 → T2⟩
      and ?Δ = Δ and ?α = 0
      and ?T1.0 = T1 → T2 and ?E = ◊ • s
      and ?U = T2])
    apply(fastforce)+
  done
next
  fix Γ T1 Δ t T2 s
  assume Γ⟨0:T1⟩ , Δ ⊢T t : T2
         (⋀s. t →β s ⟹ Γ⟨0:T1⟩ , Δ ⊢T s : T2)
         (λ T1 : t) →β s
  thus Γ , Δ ⊢T s : T1 → T2
    by (auto simp add: subst-type)
next
  fix Γ Δ r T s t sa
  assume Γ , Δ ⊢T r : T
         (⋀s. r →β s ⟹ Γ , Δ ⊢T s : T)
         Γ , Δ ⊢T s : Nat → T → T
         (⋀sa. s →β sa ⟹
             Γ , Δ ⊢T sa : Nat → T → T)
         Γ , Δ ⊢T t : Nat
         (⋀s. t →β s ⟹ Γ , Δ ⊢T s : Nat)
         Nrec T r s t →β sa
  thus Γ , Δ ⊢T sa : T
    apply(safe)
    apply(fastforce)+
    apply(frule struct-subst-type-command
      [where ?Δ1.0 = Δ⟨0:Nat⟩ and ?Δ = Δ
       and ?α = 0 and ?T1.0 = Nat
       and ?E = CNrec T r s ◊ and ?U = T])
   apply(fastforce)+
  done
next
  fix Γ Δ T c s
  assume Γ , Δ⟨0:T⟩ ⊢C c : Command
         ∀ d. c C→β d ⟶
              Γ , Δ⟨0:T⟩ ⊢C d : Command
```

```
        (μ T : c) →β s
  thus Γ , Δ ⊢T s : T
    apply(safe)
    apply(fastforce simp add: dropM-type(1))
    apply(clarsimp)
  done
next
  fix Γ Δ t T x
  assume Γ , Δ ⊢T t : T
        (⋀s. t →β s ⟹ Γ , Δ ⊢T s : T)
        Δ x = T
  thus ∀ d. <x> t C→β d ⟶
        Γ , Δ ⊢C d : Command
    apply(safe)
    apply(rule dropM-env(2)
      [where ?Δ1.0 = Δ⟨x:Δ x⟩])
    apply(drule struct-subst-type-command
      [where ?Δ1.0 = Δ⟨0:Δ x⟩ and ?Δ = Δ
       and ?α = 0 and ?T1.0 = Δ x
       and ?E = ◊ and ?U = Δ x])
     apply(fastforce)+
  done
next
  fix Γ Δ t
  assume Γ , Δ ⊢T t : ⊥
        (⋀s. t →β s ⟹ Γ , Δ ⊢T s : ⊥)
  thus ∀ d. <⊤> t C→β d ⟶
        Γ , Δ ⊢C d : Command
    apply(safe)
    apply(clarsimp
        simp add: struct-subst-type-top)+
  done
next
  fix Γ Δ s
  assume dBT.True →β s
  thus Γ , Δ ⊢T s : Bool
    by(clarsimp)
next
  fix Γ Δ s
  assume dBT.False →β s
  thus Γ , Δ ⊢T s : Bool
    by(clarsimp)
next
  fix Γ Δ t1 t2 T t3 s
  assume Γ , Δ ⊢T t1 : Bool
        (⋀s. t1 →β s ⟹ Γ , Δ ⊢T s : Bool)
        Γ , Δ ⊢T t2 : T
        (⋀s. t2 →β s ⟹ Γ , Δ ⊢T s : T)
        Γ , Δ ⊢T t3 : T
        (⋀s. t3 →β s ⟹ Γ , Δ ⊢T s : T)
        (If T t1  Then t2 Else t3) →β s
  thus Γ , Δ ⊢T s : T
    apply −
    apply(erule beta-cases)
    apply(fastforce)+
    apply(clarsimp)
    apply(frule struct-subst-type-command
      [where ?Δ = Δ and ?E = CIf T ◊ t2 t3
       and ?U = T and ?T1.0 =Bool
       and ?α = 0 and ?β = 0])
    apply(fastforce)+
  done
next
  fix Γ Δ t1 T1 t2 T2 s
  assume Γ , Δ ⊢T t1 : T1
        (⋀s. t1 →β s ⟹ Γ , Δ ⊢T s : T1)
        Γ , Δ ⊢T t2 : T2
        (⋀s. t2 →β s ⟹ Γ , Δ ⊢T s : T2)
        ((|t1,t2|):(T1×tT2)) →β s
  thus Γ , Δ ⊢T s : T1 ×t T2
    apply −
    apply(erule beta-cases)
    apply(fastforce)+
    apply(clarsimp)
    apply(frule struct-subst-type-command
      [where ?Δ = Δ
       and ?E = (|◊, t2|)l:(T1 ×t T2)
       and ?U = T1 ×t T2
       and ?T1.0 =T1 and ?α = 0
       and ?β = 0])
    apply(fastforce)+
    apply(clarsimp)
    apply(frule struct-subst-type-command
      [where ?Δ = Δ
       and ?E = (|t1, ◊|)r:(T1×t T2)
       and ?U = T1 ×t T2
       and ?T1.0 =T2 and ?α = 0
       and ?β =0])
    apply(fastforce)+
  done
next
  fix Γ Δ t T1 T2 s
  assume Γ , Δ ⊢T t : T1 ×t T2
        (⋀s. t →β s ⟹ Γ , Δ ⊢T s : T1 ×t T2)
        (π1 t) →β s
  thus Γ , Δ ⊢T s : T1
    apply(safe)
    apply(fastforce)
    apply(frule struct-subst-type-command
      [where ?Δ = Δ and ?E = Π1◊
       and ?U = T1 and ?T1.0 =T1 ×t T2
       and ?α = 0 and ?β = 0])
    apply(fastforce)+
  done
next
  fix Γ Δ t T1 T2 s
  assume Γ , Δ ⊢T t : T1 ×t T2
        (⋀s. t →β s ⟹ Γ , Δ ⊢T s : T1 ×t T2)
        (π2 t) →β s
  thus Γ , Δ ⊢T s : T2
    apply(safe)
    apply(fastforce)
    apply(frule struct-subst-type-command
      [where ?Δ = Δ and ?E = Π2◊
       and ?U = T2 and ?T1.0 =T1 ×t T2
       and ?α = 0 and ?β = 0])
    apply(fastforce)+
  done
next
  fix Γ Δ t T1 T2 s
  assume Γ , Δ ⊢_T t : T1
         (⋀s. t →_β s ⟹ Γ , Δ ⊢_T s : T1)
         Inl (T1 +_t T2) t →_β s
  thus Γ , Δ ⊢_T s : T1 +_t T2
    apply −
    apply(erule beta-cases)
    apply(fastforce)
    apply(clarsimp)
    apply(drule struct-subst-type-command
      [where ?Δ = Δ
       and ?E = CInl (T1 +_t T2) ◊
       and ?U = T1+_tT2 and ?T1.0 = T1
       and ?α = 0 and ?β = 0])
    apply(fastforce)+
 done
next
  fix Γ Δ t T1 T2 s
  assume Γ , Δ ⊢_T t : T2
         (⋀s. t →_β s ⟹ Γ , Δ ⊢_T s : T2)
         Inr (T1 +_t T2) t →_β s
  thus Γ , Δ ⊢_T s : T1 +_t T2
    apply −
    apply(erule beta-cases)
    apply(fastforce)
    apply(clarsimp)
    apply(drule struct-subst-type-command
      [where ?Δ = Δ
       and ?E = CInr (T1 +_t T2) ◊
       and ?U = T1+_tT2 and ?T1.0 = T2
       and ?α = 0 and ?β = 0])
    apply(fastforce)+
 done
next
  fix Γ Δ t0 T1 T2 t1 T t2 s
  assume Γ , Δ ⊢_T t0 : T1 +_t T2
         (⋀s. t0 →_β s ⟹ Γ , Δ ⊢_T s : T1 +_t T2)
         Γ⟨0:T1⟩ , Δ ⊢_T t1 : T
         (⋀s. t1 →_β s ⟹ Γ⟨0:T1⟩ , Δ ⊢_T s : T)
         Γ⟨0:T2⟩ , Δ ⊢_T t2 : T
         (⋀s. t2 →_β s ⟹ Γ⟨0:T2⟩ , Δ ⊢_T s : T)
  thus (Case T t0 Of Inl⇒ t1|Inr⇒ t2) →_β s
                                 ⟹ Γ , Δ ⊢_T s : T
    apply −
    apply(erule beta-cases)
    apply(fastforce simp add: subst-type)+
    apply(clarsimp)
    apply(drule struct-subst-type-command
      [where ?Δ = Δ and ?U = T
       and ?T1.0 = T1 +_t T2
       and ?E = CCase T ◊ Of CInl⇒ t1|
                             CInr⇒t2
       and ?β = 0])
    apply(fastforce)
    apply(rule refl)
    apply(fastforce)
  done
qed
```

A.2 Progress

```
lemma normal-forms:
  Γ, Δ ⊢_T t : T ⟹ flv-dBT t 0 = {} ⟹ (∀ s. ¬(t →_β s)) ⟹
    (is-val t) ∨ (∃ U β v. t = (μ U: (<β> v)) ∧ (is-val v))
      ∨ (∃ U v. t = (μ U: (<⊤> v)) ∧ (is-val v))
  Γ, Δ ⊢_C c : T ⟹ flv-dBC c 0 = {}  ⟹ (∀ β t. c = (<β> t) ⟶ (∀ d. ¬(t →_β d)) ⟶
    ((is-val t) ∨ (∃ U γ v. t = (μ U: (<γ> v)) ∧ (is-val v))
      ∨ (∃ U v. t = (μ U: (<⊤> v)) ∧ (is-val v))))
                                   ∧ (∀ t. c = (<⊤> t) ⟶ (∀ d. ¬(t →_β d)) ⟶
    ((is-val t) ∨ (∃ U γ v. t = (μ U: (<γ> v)) ∧ (is-val v))
      ∨ (∃ U v. t = (μ U: (<⊤> v)) ∧ (is-val v))))
proof(induct rule: typing-dBT-typing-dBC.inducts)
  fix Γ x T Δ
  assume flv-dBT (‘x) 0 = {}
         ∀s. (¬ (‘x) →_β s)
  thus is-val (‘x) ∨ (∃U β v. ‘x = (μ U: (<β> v)) ∧ is-val v)
    ∨ (∃U v. ‘x = (μ U:(<⊤> v)) ∧ is-val v)
  by(clarsimp)
next
  fix Γ Δ
  assume flv-dBT Zero 0 = {}
         ∀s. ¬ Zero →_β s
  thus is-val Zero ∨ (∃U β v. Zero = (μ U: (<β> v)) ∧ is-val v)
```

```
      ∨ (∃ U v. Zero = (μ U:(<⊤> v)) ∧ is-val v)
  by(clarsimp)
next
  fix Γ Δ t
  assume Γ , Δ ⊢_T t : Nat
         (flv-dBT t 0 = {} ⟹ ∀ s. (¬ t →_β s) ⟹
            (is-val t) ∨ (∃ U β v. t = (μ U: (<β> v)) ∧ is-val v)
            ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
         flv-dBT (S t) 0 = {}
         ∀ s. ¬ S t →_β s
  thus is-val (S t) ∨ (∃ U β v. S t = (μ U:(<β> v)) ∧ is-val v)
         ∨ (∃ U v. S t = (μ U:(<⊤> v)) ∧ is-val v)
  by(fastforce)
next
  fix Γ Δ t T1 T2 s
  assume Γ , Δ ⊢_T t : T1 → T2
         (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →_β s ⟹
          is-val t ∨ (∃ U β v. t = (μ U: (<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
         Γ , Δ ⊢_T s : T1
         (flv-dBT s 0 = {} ⟹ ∀ sa. ¬ s →_β sa ⟹
          is-val s ∨ (∃ U β v. s = (μ U: (<β> v)) ∧ is-val v) ∨ (∃ U v. s = (μ U:(<⊤> v)) ∧ is-val v))
         flv-dBT (t ° s) 0 = {}
         ∀ sa. ¬ t ° s →_β sa
  thus is-val (t ° s) ∨ (∃ U β v. t ° s = (μ U: (<β> v)) ∧ is-val v)
        ∨ (∃ U v. t ° s = (μ U:(<⊤> v)) ∧ is-val v)
    apply(clarsimp)
    apply(subgoal-tac ∀ s. ¬ t →_β s)
    apply(clarsimp)
    apply(rule disjE [where ?P = is-val t and ?Q = (∃ U β v. t = (μ U: (<β> v)) ∧ is-val v)])
    apply(fastforce)
    apply(drule typed-values [where ?Γ = Γ and ?Δ = Δ and ?T = T1→T2 and ?T1.0 = T1
                              and ?T2.0 = T2])
    apply(fastforce)+
  done
next
  fix Γ T1 Δ t T2
  assume Γ⟨0:T1⟩ , Δ ⊢_T t : T2
         (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →_β s ⟹
          is-val t ∨ (∃ U β v. t = (μ U: (<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
         flv-dBT (λ T1:t) 0 = {}
         ∀ s. ¬ (λ T1:t) →_β s
  thus is-val (λ T1:t) ∨ (∃ U β v. (λ T1:t) = (μ U: (<β> v)) ∧ is-val v)
        ∨ (∃ U v. λT1:t = (μ U:(<⊤> v)) ∧ is-val v)
    by(fastforce)
next
  fix Γ Δ r T s t
  assume Γ , Δ ⊢_T r : T
         (flv-dBT r 0 = {} ⟹ ∀ s. ¬ r →_β s ⟹
         is-val r ∨ (∃ U β v. r = (μ U: (<β> v)) ∧ is-val v) ∨ (∃ U v. r = (μ U:(<⊤> v)) ∧ is-val v))
         Γ , Δ ⊢_T s : Nat → T → T
         (flv-dBT s 0 = {} ⟹ ∀ sa. ¬ s →_β sa ⟹
         is-val s ∨ (∃ U β v. s = (μ U: (<β> v)) ∧ is-val v) ∨ (∃ U v. s = (μ U:(<⊤> v)) ∧ is-val v))
         Γ , Δ ⊢_T t : Nat
         (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →_β s ⟹
         is-val t ∨ (∃ U β v. t = (μ U: (<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
         flv-dBT (Nrec T r s t) 0 = {}
          ∀ sa. ¬ Nrec T r s t →_β sa
  thus  is-val (Nrec T r s t) ∨ (∃ U β v. Nrec T r s t = (μ U: (<β> v)) ∧ is-val v)
```

```
          ∨ (∃ U v. Nrec T r s t = (μ U:(<⊤> v)) ∧ is-val v)
     apply(clarsimp)
     apply(subgoal-tac ∀ s. ¬ t →_β s)
     apply(clarsimp)
     apply(rule disjE [where ?P = is-val t and ?Q = (∃ U β v. t = (μ U: (<β> v)) ∧ is-val v)])
     apply(fastforce)
     apply(drule typed-values [where ?Γ = Γ and ?Δ = Δ and ?T = Nat and ?T1.0 = T1
                               and ?T2.0 = T2])
     apply(fastforce)
     apply(fastforce)
     apply(clarsimp)
     apply(rule disjE [where ?P = t = Zero and ?Q = (∃ v1. t = S v1 ∧ is-natval v1)])
     apply(blast)+
   done
next
  fix Γ Δ T c
  assume Γ , Δ⟨0:T⟩ ⊢_C c : Command
        (flv-dBC c 0 = {}  ⟹ (∀ β t. c = (<β> t) ⟶ (∀ d. ¬ t →_β d)
         ⟶ (is-val t ∨ (∃ U γ v. t = (μ U: (<γ> v)) ∧ is-val v)
            ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v)))
            ∧ (∀ t. c = <⊤> t ⟶ (∀ d. ¬ t →_β d)
         ⟶ is-val t ∨ (∃ U γ v. t = (μ U:(<γ> v)) ∧ is-val v)
            ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v)))
        flv-dBT (μ T:c) 0 = {}
        ∀ s. ¬ (μ T:c) →_β s
  thus is-val (μ T:c) ∨ (∃ U β v. (μ T:c) = (μ U: (<β> v)) ∧ is-val v)
        ∨ (∃ U v. μ T:c = (μ U:(<⊤> v)) ∧ is-val v)
     apply(clarsimp)
     apply(cases c)
     apply(fastforce)+
   done
next
  fix Γ Δ t T x
  assume Γ , Δ ⊢_T t : T
        (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →_β s ⟹
         is-val t ∨ (∃ U β v. t = (μ U: (<β> v)) ∧ is-val v) ∨  (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
        Δ x = T
        flv-dBC (<x> t) 0 = {}
  thus (∀ β ta. <x> t = <β> ta  ⟶ (∀ d. ¬ ta →_β d)
       ⟶ is-val ta ∨ (∃ U γ v. ta = (μ U: (<γ> v)) ∧ is-val v)
            ∨ (∃ U v. ta = (μ U:(<⊤> v)) ∧ is-val v)) ∧
       (∀ ta. <x> t = <⊤> ta ⟶ (∀ d. ¬ ta →_β d)
       ⟶ is-val ta ∨ (∃ U γ v. ta = (μ U:(<γ> v)) ∧ is-val v)
            ∨ (∃ U v. ta = (μ U:(<⊤> v)) ∧ is-val v))
     by(clarsimp)
next
  fix Γ Δ t
  assume Γ , Δ ⊢_T t : ⊥
        (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →_β s ⟹
         is-val t ∨ (∃ U β v. t = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
        flv-dBC (<⊤> t) 0 = {}
  thus (∀ β ta. <⊤> t = <β> ta ⟶ (∀ d. ¬ ta →_β d) ⟶
        is-val ta ∨ (∃ U γ v. ta = (μ U:(<γ> v)) ∧ is-val v)
            ∨ (∃ U v. ta = (μ U:(<⊤> v)) ∧ is-val v)) ∧
       (∀ ta. <⊤> t = <⊤> ta ⟶ (∀ d. ¬ ta →_β d) ⟶
       is-val ta ∨ (∃ U γ v. ta = (μ U:(<γ> v)) ∧ is-val v) ∨ (∃ U v. ta = (μ U:(<⊤> v)) ∧ is-val v))
     by(clarsimp)
next
```

```
  fix Γ Δ
  assume flv-dBT True 0 = {}
         ∀ s. ¬ True →β s
  thus is-val True ∨ (∃ U β v. True = (μ U:(<β> v)) ∧ is-val v)
            ∨ (∃ U v. True = (μ U:(<⊤> v)) ∧ is-val v)
    by(clarsimp)
next
  fix Γ Δ
  assume flv-dBT False 0 = {}
         ∀ s. ¬ False →β s
  thus is-val False ∨ (∃ U β v. False = (μ U:(<β> v)) ∧ is-val v)
         ∨ (∃ U v. False = (μ U:(<⊤> v)) ∧ is-val v)
    by(clarsimp)
next
  fix Γ Δ t1 t2 T t3
  assume Γ , Δ ⊢T t1 : Bool
         (flv-dBT t1 0 = {} ⟹ ∀ s. ¬ t1 →β s ⟹ is-val t1
            ∨ (∃ U β v. t1 = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t1 = (μ U:(<⊤> v)) ∧ is-val v))
         Γ , Δ ⊢T t2 : T
         (flv-dBT t2 0 = {} ⟹ ∀ s. ¬ t2 →β s ⟹ is-val t2
            ∨ (∃ U β v. t2 = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t2 = (μ U:(<⊤> v)) ∧ is-val v))
         Γ , Δ ⊢T t3 : T
         (flv-dBT t3 0 = {} ⟹ ∀ s. ¬ t3 →β s ⟹ is-val t3
            ∨ (∃ U β v. t3 = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t3 = (μ U:(<⊤> v)) ∧ is-val v))
         flv-dBT (If T t1 Then t2 Else t3) 0 = {}
         ∀ s. ¬ (If T t1 Then t2 Else t3) →β s
  thus is-val (If T t1 Then t2 Else t3)
         ∨ (∃ U β v. (If T t1 Then t2 Else t3) = (μ U:(<β> v)) ∧ is-val v)
         ∨ (∃ U v. (If T t1 Then t2 Else t3) = (μ U: (<⊤> v)) ∧ is-val v)
    apply(clarsimp)
    apply(subgoal-tac ∀ s. ¬t1→β s)
    apply(clarsimp)
    apply(rule disjE)
    apply(assumption)

    apply(cases t1 rule: is-val.cases)
    apply(clarsimp)
    apply(blast)+
  done
next
  fix Γ Δ t1 T1 t2 T2
  assume Γ , Δ ⊢T t1 : T1
         (flv-dBT t1 0 = {} ⟹ ∀ s. ¬ t1 →β s ⟹
          is-val t1 ∨ (∃ U β v. t1 = (μ U:(<β> v)) ∧ is-val v)
            ∨ (∃ U v. t1 = (μ U:(<⊤> v)) ∧ is-val v))
         Γ , Δ ⊢T t2 : T2
         (flv-dBT t2 0 = {} ⟹ ∀ s. ¬ t2 →β s ⟹
          is-val t2 ∨ (∃ U β v. t2 = (μ U:(<β> v)) ∧ is-val v)
            ∨ (∃ U v. t2 = (μ U:(<⊤> v)) ∧ is-val v))
         flv-dBT ((|t1,t2|):(T1 ×t T2)) 0 = {}
         ∀ s. ¬ ((|t1,t2|):(T1 ×t T2)) →β s
  thus is-val ((|t1,t2|):(T1 ×t T2)) ∨ (∃ U β v. ((|t1,t2|):(T1 ×t T2)) = (μ U:(<β> v)) ∧ is-val v)
         ∨ (∃ U v. ((|t1,t2|):(T1 ×t T2)) = (μ U:(<⊤> v)) ∧ is-val v)
    by(fastforce)
next
  fix Γ Δ t T1 T2
  assume Γ , Δ ⊢T t : T1 ×t T2
         (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →β s ⟹
          is-val t ∨ (∃ U β v. t = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
        flv-dBT (π₁ t) 0 = {}
        ∀ s. ¬ (π₁ t) →β s
  thus is-val (π₁ t) ∨ (∃ U β v. (π₁ t) = (μ U:(<β> v)) ∧ is-val v)
           ∨ (∃ U v. (π₁ t) = (μ U:(<⊤> v)) ∧ is-val v)
    apply(clarsimp)
    apply(subgoal-tac ∀ s. ¬ t →β s)
    apply(clarsimp)
    apply(rule disjE)
    apply(assumption)
    apply(cases t rule: is-val.cases)
    apply(fastforce)+
  done
next
  fix Γ Δ t T1 T2
  assume Γ , Δ ⊢T t : T1 ×t T2
        (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →β s ⟹
          is-val t ∨ (∃ U β v. t = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
        flv-dBT (π₂ t) 0 = {}
        ∀ s. ¬ (π₂ t) →β s
  thus is-val (π₂ t) ∨ (∃ U β v. (π₂ t) = (μ U:(<β> v)) ∧ is-val v)
        ∨ (∃ U v. (π₂ t) = (μ U:(<⊤> v)) ∧ is-val v)
    apply(clarsimp)
    apply(subgoal-tac ∀ s. ¬ t →β s)
    apply(clarsimp)
    apply(rule disjE)
    apply(assumption)
    apply(cases t rule: is-val.cases)
    apply(fastforce)+
  done
next
  fix Γ Δ t T1 T2
  assume Γ , Δ ⊢T t : T1
        (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →β s ⟹
           is-val t ∨ (∃ U β v. t = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
        flv-dBT (Inl (T1 +t T2) t) 0 = {}
        ∀ s. ¬ Inl (T1 +t T2) t →β s
  thus is-val (Inl (T1 +t T2) t) ∨
       (∃ U β v. Inl (T1 +t T2) t = (μ U:(<β> v)) ∧ is-val v)
           ∨ (∃ U v. Inl (T1 +t T2) t = (μ U:(<⊤> v)) ∧ is-val v)
     by(fastforce)
next
  fix Γ Δ t T1 T2
  assume Γ , Δ ⊢T t : T2
        (flv-dBT t 0 = {} ⟹ ∀ s. ¬ t →β s ⟹
           is-val t ∨ (∃ U β v. t = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t = (μ U:(<⊤> v)) ∧ is-val v))
        flv-dBT (Inr (T1 +t T2) t) 0 = {}
        ∀ s. ¬ Inr (T1 +t T2) t →β s
  thus is-val (Inr (T1 +t T2) t) ∨
       (∃ U β v. Inr (T1 +t T2) t = (μ U:(<β> v)) ∧ is-val v)
           ∨ (∃ U v. Inr (T1 +t T2) t = (μ U:(<⊤> v)) ∧ is-val v)
     by(fastforce)
next
  fix Γ Δ t0 T1 T2 t1 T t2
  assume Γ , Δ ⊢T t0 : T1 +t T2
       (flv-dBT t0 0 = {} ⟹ ∀ s. ¬ t0 →β s ⟹ is-val t0 ∨ (∃ U β v. t0 = (μ U:(<β> v)) ∧ is-val v)
           ∨ (∃ U v. t0 = (μ U:(<⊤> v)) ∧ is-val v))
        Γ⟨0:T1⟩ , Δ ⊢T t1 : T
        (flv-dBT t1 0 = {} ⟹ ∀ s. ¬ t1 →β s ⟹ is-val t1
            ∨ (∃ U β v. t1 = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t1 = (μ U:(<⊤> v)) ∧ is-val v))
        Γ⟨0:T2⟩ , Δ ⊢T t2 : T
        (flv-dBT t2 0 = {} ⟹ ∀ s. ¬ t2 →β s ⟹ is-val t2
            ∨ (∃ U β v. t2 = (μ U:(<β> v)) ∧ is-val v) ∨ (∃ U v. t2 = (μ U:(<⊤> v)) ∧ is-val v))
        flv-dBT (Case T  t0 Of Inl⇒ t1|Inr⇒ t2) 0 = {}
        ∀ s. ¬ (Case T  t0 Of Inl⇒ t1|Inr⇒ t2) →β s
  thus is-val (Case T  t0 Of Inl⇒ t1|Inr⇒ t2) ∨
        (∃ U β v. (Case T  t0 Of Inl⇒ t1|Inr⇒ t2) = (μ U:(<β> v)) ∧ is-val v)
            ∨ (∃ U v. (Case T  t0 Of Inl⇒ t1|Inr⇒ t2) = (μ U:(<⊤> v)) ∧ is-val v)
    apply(clarsimp)
    apply(subgoal-tac ∀ s. ¬ t0 →β s)
    apply(clarsimp)
    apply(rule disjE)
    apply(assumption)
    apply(cases rule: is-val.cases)
    apply(blast)+
   done
qed

theorem progress-closed:
  Γ, Δ ⊢T t : T ⟹ flv-dBT t 0 = {} ⟹
    ((is-val t) ∨ (∃  U β v. t = (μ U: (<β> v)) ∧ (is-val v)) ∨
    (∃  U v. t = (μ U: (<⊤> v)) ∧ (is-val v))) ∨
    (∃  s. t →β s)
  apply(rule disjCI)
  apply(rule normal-forms)
  apply(fastforce)+
done
```

A.3 Isabelle Definition of the Reduction Function Used in the Interpreter

```
function red-term :: dBT ⇒ dBT option
and red-command :: dBC ⇒ dBC option
where
    red-app: red-term (s°t)
      = (case s of
          (λ T:r) ⇒ Some (r[t/0]^T)
         |(μ (T1→T2) : c) ⇒
                Some (μ T2 : (c[(Some 0) = (Some 0) (◇ • (liftM-dBT t 0))]^C))
         | - => (case (red-term s) of
                 Some u ⇒ Some (u°t)
                |None ⇒ (case (red-term t) of
                          Some u ⇒ Some (s°u)
                         |None ⇒ None)))

  | red-lambda: red-term (λ T : s)
      = (case (red-term s) of
         Some u ⇒ Some (λ T : u)
        |None ⇒ None)

  | red-suc: red-term (S t)
      = (case t of
         (μ T : c) ⇒ Some (μ T : (c[(Some 0) = (Some 0) (CSuc ◇)]^C))
        | - ⇒ (case (red-term t) of
                Some u ⇒ Some (S u)
```

```
            |None ⇒ None))

| red-mu: red-term (μ T : c)
    = (case c of
        (<0> t) ⇒ (if (0 ∉ (fmv-dBT t 0)) then Some (dropM-dBT t 0)
                    else (case (red-command (<0> t)) of
                            Some d ⇒ Some (μ T : d)
                           |None ⇒ None))
       | - ⇒ (case (red-command c) of
                Some d ⇒ Some (μ T : d)
               |None ⇒ None))

| red-mVar: red-command (<i> t)
    = (case t of
        (μ T : c) ⇒ Some (dropM-dBC (c[(Some 0) = (Some i) ◇]^C) i)
       | - ⇒ (case (red-term t) of
                Some u ⇒ Some (<i> u)
               |None ⇒ None))

| red-nrec: red-term (Nrec T r s t)
    = (case t of
        Zero ⇒ Some r
       |(S n) ⇒ (if is-natval n then Some (s°n°(Nrec T r s n))
                  else (case red-term (S n) of
                          Some u ⇒ Some (Nrec T r s u)
                         |None ⇒(case (red-term r) of
                                   Some u ⇒ Some (Nrec T u s (S n))
                                  |None ⇒ (case (red-term s) of
                                             Some u ⇒ Some (Nrec T r u (S n))
                                            |None ⇒ None))))
       |(μ T1 : c) ⇒ Some (μ T : (c[(Some 0) =
                  (Some 0) (CNrec T (liftM-dBT r 0) (liftM-dBT s 0) ◇)]^C))
       | - ⇒ (case (red-term r) of
                Some u ⇒ Some (Nrec T u s t)
               |None ⇒ (case (red-term s) of
                          Some u ⇒ Some (Nrec T r u t)
                         |None ⇒ (case (red-term t) of
                                    Some u ⇒ Some (Nrec T r s u)
                                   |None ⇒ None))))

| red-top: red-command (<⊤> t)
    = (case t of
        (μ T : c) ⇒ Some (dropM-dBC (c[(Some 0) = None ◇]^C) 0)
       | - ⇒ (case (red-term t) of
                Some u ⇒ Some (<⊤> u)
               |None ⇒ None))

| red-if: red-term (If T t1 Then t2 Else t3)
    = (case t1 of
        True ⇒ Some t2
       |False ⇒ Some t3
       |(μ U: c) ⇒ Some μ T:(c[(Some 0) =
                   (Some 0) (CIf T ◇ (liftM-dBT t2 0) (liftM-dBT t3 0))]^C)
       | - ⇒ (case red-term t1 of
                Some s1 ⇒ Some (If T s1 Then t2 Else t3)
               |None ⇒ None))

| red-proj1: red-term (π_1 t)
```

```
    = (case t of
        (|t1, t2|):T ⇒ Some t1
       |(μ T1×_t T2: c) ⇒ Some μ T1:(c[(Some 0) = (Some 0) (Π_1 ◇)]^C)
       | - ⇒ (case red-term t of
                Some s ⇒ Some (π_1 s)
               |None ⇒ None))

  | red-proj2: red-term (π_2 t)
    = (case t of
        (|t1, t2|):T ⇒ Some t2
       |(μ T1×_t T2: c) ⇒ Some μ T2:(c[(Some 0) = (Some 0) (Π_2 ◇)]^C)
       | - ⇒ (case red-term t of
                Some s ⇒ Some (π_2 s)
               |None ⇒ None))

  | red-pair: (red-term (|t1, t2|):T)
    = (case t1 of
        μ T1: c ⇒ Some μ T:(c[(Some 0) =
                                  (Some 0) (|◇, (liftM-dBT t2 0)|)_l:T]^C)
       | - ⇒ (case t2 of
                μ T2: c ⇒ Some μ T:(c[(Some 0) =
                                  (Some 0) (|(liftM-dBT t1 0), ◇|)_r:T]^C)
               | - ⇒ (case red-term t1 of
                        Some s1 ⇒ Some (|s1, t2|):T
                       |None ⇒ (case red-term t2 of
                                  Some s2 ⇒ Some (|t1, s2|):T
                                 |None ⇒ None))))

  | red-case: red-term (Case U t Of Inl⇒ t1|Inr⇒ t2)
    = (case t of
        (Inl T s) ⇒ Some (t1[s/0]^T)
       |(Inr T s) ⇒ Some (t2[s/0]^T)
       |(μ T: c) ⇒ Some μ U:(c[(Some 0) = (Some 0)
                          (CCase U ◇ Of CInl⇒ (liftM-dBT t1 0)|CInr⇒ (liftM-dBT t2 0))]^C)
       | - ⇒ (case red-term t of
                Some u ⇒ Some (Case U u Of Inl⇒ t1|Inr⇒ t2)
               |None ⇒ None))

  | red-inl: red-term (Inl T s)
    = (case s of
        μ U: c ⇒ Some μ T:(c[(Some 0) = (Some 0) (CInl T ◇)]^C)
       | - ⇒ (case red-term s of
                Some u ⇒ Some (Inl T u)
               |None ⇒ None))

  | red-inr: red-term (Inr T s)
    = (case s of
        μ U: c ⇒ Some μ T:(c[(Some 0) = (Some 0) (CInr T ◇)]^C)
       | - ⇒ (case red-term s of
                Some u ⇒ Some (Inr T u)
               |None ⇒ None))

  | red-term-lvar: red-term (‘x) = None
  | red-term-zero: red-term Zero = None
  | red-term-true: red-term True = None
  | red-term-false: red-term False = None
by pat-completeness auto
termination by lexicographic-order
```

Appendix B

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Formalisation of the $\lambda\mu^{\mathbf{T}}$-calculus in Isabelle/HOL

C. Matache, Fitzwilliam College
Originator: Dr V. B. F. Gomes
20 October 2016

Project Supervisor: Dr V. B. F. Gomes
Director of Studies: Dr R. K. Harle
Project Overseers: Dr D. J. Greaves & Prof J. Daugman

Introduction

The Curry-Howard correspondence [1] is a result from programming language theory which relates types from the typed λ-calculus to propositions in intuitionistic logic. The $\lambda\mu$-calculus is a calculus whose types correspond to propositions in classical logic in the same way. This calculus was introduced by M. Parigot in [2], and builds on the work of T. Griffin, who was the first to notice, in [3], the correspondence between constructs specific to classical logic and control operators in programming languages. Apart from the terms in the λ-calculus, the $\lambda\mu$-calculus has μ-variables that can name any subterm of a λ-term. The $\lambda\mu^{\mathbf{T}}$-calculus, which forms the basis of this project, was introduced in [4]. It is an extension of the $\lambda\mu$-calculus containing in addition a datatype for natural numbers based on Gödel's **T**. This is an attempt to produce a calculus that has both control operators and datatypes, and therefore bears a stronger resemblance to a programming language.

My project aims to derive a language from the $\lambda\mu^{\mathbf{T}}$-calculus, then implement an interpreter for this language. The next step is to formally prove metatheoretical properties of the language, such as type preservation and type progress, that are described in [4], using the proof assistant Isabelle. Two important parts of the interpreter, the type-checker and the evaluator, are to be implemented first in Isabelle, then exported to OCaml, where they can be executed. The reason for formalising them in Isabelle is that properties of the language can then be proved mechanically. The lexer, parser and pretty-printer for the interpreter will be implemented in OCaml, which is why the Isabelle implementation will be exported to OCaml code.

Starting point

To the best of my knowledge, there is no previous implementation of a language derived directly from the $\lambda\mu^{\mathbf{T}}$-calculus, neither a formalisation of its properties that are described in [4].

In preparation for this project, I have read the paper [4] to familiarise myself with the $\lambda\mu^{\mathbf{T}}$-calculus. Also, I have consulted papers [2, 3] to gain a better understanding of the $\lambda\mu$-calculus and what is meant by its correspondence to classical logic.

Prior to arranging this project, I had no experience working with proof assistants. When it became apparent that I was going to use Isabelle, I started working through the examples and exercises in the tutorial *Programming and Proving in Isabelle/HOL*, which constitutes the first part of the book [5]. My current knowledge of OCaml is limited to the exposure I had to it through the Part IB Compiler Construction course.

Resources required

The software that this project requires, such as Isabelle and the OCaml compiler, is open-source. I plan to use my own computer for working on the project. Its specifications are: Intel i5 1.6 GHz CPU, 8 GB RAM, 128 GB SSD disk, running Windows 10. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. My contingency plan includes storing all my code and other documents in a Git repository hosted on Bitbucket, as well as on Google Drive (with automatic backups), and making weekly checkpoints of the work on an external SSD drive that I own. Should my computer fail, I will be able to easily transfer my work to one of the MCS machines. I do not require any other special resources.

Work to be done

The aim of the project is to produce a verified interpreter for a language based on the $\lambda\mu^{\mathbf{T}}$-calculus, and prove properties of this language. Therefore, the project will have the following components:

1. Learning Isabelle well enough to write the automated proofs necessary for this project. I am going to do this by continuing to read [5], and doing any other tutorials that might be useful. At the same time, I need to become more comfortable using OCaml.

2. Defining a programming language derived from the $\lambda\mu^{\mathbf{T}}$-calculus. My starting point for this is the paper [4], where the calculus is defined. I will attempt to adapt the definition of the reduction rules given here to ones that do not use contexts, as this would make the implementation of the evaluation function more straightforward.

3. Formalising the $\lambda\mu^{\mathbf{T}}$-calculus in Isabelle. For this, I will use the De Bruijn index notation to represent bound variables.

4. Writing an interpreter for my language. This includes: exporting the type-checker and evaluator from Isabelle to OCaml using Isabelle's code generation feature [6]; writing a lexer and parser in OCaml, using OCamllex and Menhir respectively; writing a pretty printer; putting all these components together.

5. Proving type preservation for my programming language in Isabelle.

6. Proving type progress.

Success criteria

Whether the following goals have been achieved can be used as an indicator for the success of the project:

1. Become familiar enough with using OCaml and writing proofs in Isabelle so as to be able to complete the rest of the project.

2. Define a language based on the $\lambda\mu^{\mathbf{T}}$-calculus.

3. Formalise the $\lambda\mu^{\mathbf{T}}$-calculus in Isabelle.

4. Generate OCaml code for the type-checker and evaluator based on this formalisation.

5. Write a lexer, parser, and pretty-printer in OCaml. Use these components, and the code generated by Isabelle, to create an interpreter for my language.

6. Prove type preservation for my language in Isabelle.

7. Prove type progress for my language in Isabelle.

Possible extensions

Should the time allow, possible extensions of the project include:

1. Proving more properties of the language in Isabelle, such as confluence.

2. Proving some results that are true in classical logic but not in intuitionistic logic, such as Euclid's proof of the infinitude of primes. This can be done by writing a program in my language that type-checks, and whose result type corresponds to the statement of the theorem. The ease with which such proofs can be written can serve as an evaluation metric.

3. Implementing a type inference algorithm for my language.
4. Extending the calculus with more datatypes so that the properties proved previously still hold. This should be accompanied by proofs in Isabelle that these properties are preserved. Then, the interpreter can be extended to include the new datatypes. The motivation for this extension is to make the calculus more similar to a real programming language.

Timetable

The planned starting date is 21/10/2016.

Michaelmas term

1. **Weeks 3–4:** Learn to use Isabelle well enough to do the automated proofs required by the project. Familiarise myself more with OCaml. Read any other papers that might be useful apart from the ones I have already read.

   Milestone: Be able to use Isabelle and OCaml at a satisfactory level, and have the required theoretical background for the project.
2. **Weeks 5–6:** Design a language based on the $\lambda\mu^{\mathbf{T}}$-calculus and formalise the calculus in Isabelle. Then export the code to OCaml.

   Milestone: Have the type-checker and evaluator working in OCaml.
3. **Weeks 7–8:** Implement the other components of the interpreter in OCaml: lexer, parser, pretty-printer. Glue all the components together.

   Milestone: Have a working interpreter.
4. **Christmas vacation:** Solve any issues with the interpreter. Do more research into the methods for proving type preservation and start this proof for my language in Isabelle.

   Milestone: The interpreter is finished. The proof of type preservation is underway.

Lent term

5. **Weeks 1–2:** Write the progress report and prepare the presentation. Finish the type preservation proof. Start the proof of type progress.

   Milestone: The type preservation proof is finished.

6. **Weeks 3–5:** Do the type progress proof in Isabelle.

   Milestone: The type progress proof is finished.

7. **Weeks 6–8:** Work on the evaluation of the project. Start any extensions if there is time.

   Milestone: Have the evaluation data.

8. **Easter vacation:** Write the main chapters of the dissertation. At this point, I hope to reuse material that I have written in the course of the project. Continue implementing some of the extensions if there is time.

   Milestone: The dissertation is entirely written.

Easter term

9. **Weeks 1–2:** Improve the dissertation where necessary.

   Milestone: The dissertation is in its final form.

10. **Week 3:** Proof read the dissertation and submit it early.

    Milestone: The dissertation and code are submitted.

References

[1] Howard, W. A. (September 1980) [original paper manuscript from 1969]. The formulae-as-types notion of construction. In Seldin, Jonathan P.; Hindley, J. Roger, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Boston, MA: Academic Press, pp. 479490, ISBN 978-0-12-349050-6.

[2] Parigot, M. (1992). $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning* (pp. 190-201). Springer Berlin/Heidelberg.

[3] Griffin, T. G. (1989, December). A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 47-58). ACM.

[4] Geuvers, H., Krebbers, R.J., McKinna, J. (2013). The $\lambda\mu^{\mathbf{T}}$-calculus. In *Annals of Pure and Applied Logic*, *164*(6), 676-701.

[5] Nipkow, T., Klein, G. (2014). Concrete Semantics. A Proof Assistant Approach.

[6] Haftmann, F., Bulwahn, L. (2013). Code generation from Isabelle/HOL theories.