Chapter 1

Introduction

1.1 Motivation

1.1.1 The Dual Calculus

The Curry–Howard correspondence is one of the most striking results in Computer Science, forging a link between two seemingly disconnected fields of study: programming language theory and mathematical logic. Beginning from a series of separate observations [13, 15, 23], rather than one single discovery, the relationship between intuitionistic logic systems and models of computation is well understood. However, it came as a surprise to most when, in 1989, this relationship was extended into classical logic, with Tim Griffin [21] noting its correspondence to programming with continuations.

Also in 1989, Filinski [19] became the first to conjecture another interesting result: that, in the presence of continuations, the Call-by-Value (CbV) evaluation strategy was the de Morgan dual of Call-by-Name (CbN). A stream of papers followed, each aiming to demonstrate this duality in a different way, none of which were wholly compelling.

The Dual Calculus of Philip Wadler [46] [1] lies at the confluence of these two streams of research. It is a computational calculus that corresponds exactly to Gentzen's classical sequent calculus [20], that also cleanly demonstrates the duality between Call-by-Value and Call-by-Name.

1.1.2 Language Formalisation

Embedding the syntax and semantics of a programming language into a proof assistant is not a new idea, and its benefits are well-documented. These benefits are neatly demonstrated by Nipkow [30] in "Winskel is (almost) right", in which he produces a formalisation of the first 100 pages of Winskel's widely-cited textbook: "The Formal Semantics of Programming Languages" [50]. As the name would suggest, Nipkow used this formalisation to identify an error in one of Winskel's proofs. Wadler presents many theorems regarding the Dual Calculus without demonstrating a proof; developing a formalisation of this paper can give us much greater confidence in its correctness, and can identify any necessary corrections. In fact, in the process of producing this formalisation, I located a minor error in an example Wadler gives in the original paper.

Over years of research, a series of different techniques for formalising the syntax of formal languages in proof assistants. These include: higher-order abstract syntax [39], locally nameless representation [9, 29, 49, 8], and intrinsically-typed representation [18, 1, 7].

[1] I will refer to this paper frequently in this dissertation, as such from here on I will call it 'the original paper'

While it would be great to guarantee the correctness of every mathematical proof by producing such a formalisation, the process is “extremely laborious” [38] and time-consuming. What one might consider to be trivial in an informal setting can become very complex when considered formally.

1.2 Project Description

The aim of the project was to formalise the DC as set out in the original paper. The formalisation is split into three parts: syntax, operational semantics, and denotational semantics. Definitions and proofs regarding these concepts are found in Sections 3.2, 3.3, and 3.4 respectively. Section 3.5 discusses, and demonstrates proofs of, the duality of each of the three previous parts.

Note that defining an operational semantics for the DC, including formal definition of substitution it required, was completed as an extension, as was the proof of the soundness of the DC’s denotational semantics found in Sections 3.4.3 and 3.4.4.

In a sense, the existence of the formalisation is itself an evaluation of the project, though I used other methods to judge whether I was successful. This included using the formalised DC to produce ‘unit tests’ to validate the formalisation’s behaviour (Section 4.2), as well as using the DC formalisation to simulate a variety of other Calculi (Section 4.3). I then used one of these simulated calculi to show that it is possible to do simple arithmetic in the DC formalisation.

I used the proof assistant *Agda* [31] for this project. The first reason for this is that it is a widely used proof assistants, so a good level of documentation and support exists online, including Philip Wadler’s Programming Language Foundations in Agda (PLFA) [48]. Proofs in Agda are spelled out explicitly, rather than being hidden behind opaque tactics like in Coq [5] or Isabelle [36], and it’s syntax is cleaner and more flexible: allowing mixfix operators and Unicode characters. This means that proofs in Agda are significantly more readable than those in an alternative proof assistant. A final, more practical, consideration was that it’s the main language my supervisor works in, so I would be able to draw on his knowledge for assistance.

1.3 Related Work

Tzevelekos [45] describes the DC as “the outcome of two distinct lines of research ... (A) Efforts to extend the Curry-Howard isomorphism ... to classical logic. (B) Efforts to establish the tacit conjecture that CBV reduction ... is dual to CBN reduction”.

The first line of research has generated considerable interest since Griffin [21] first introduced the idea that there was a correspondence between languages with control operators and classical logic. The most well-known demonstration of this is Parigot’s $\lambda\mu$-calculus [35], which extends Church’s Simply Typed λ-calculus (STLC) [10] with a μ binder that abstracts on the program’s current continuation. The $\lambda\mu$-calculus has been shown to correspond to Gentzen’s Classical Natural Deduction, and its Call-by-Name and Call-by-Value semantics have both been investigated [34, 33]. Crolard used the $\lambda\mu$-calculus to derive a λ-calculus with an explicit catch/throw mechanism; he calls this the λ_{ct}-calculus [11]. Separately, Barbanera produced a λ-calculus with symmetric reduction rules that corresponds to classical logic [4].

There have been various investigations into the duality of Call-by-Value and Call-by-Name since Filinski first introduced the idea [19]. Sellinger, in two separate papers [42, 43], modeled the CbN and CbV semantics of the $\lambda\mu$-calculus in a control category and a dual co-control category, though the duality between these categories is not an involution. This work was improved upon by Curien and Herbelin [12] who, exploiting its inherent duality, derived a computational calculus from the classical sequent calculus. In doing this, however, they introduce a difference operator as the dual of implication; the computational interpretation of which is far from intuitive. Barbenera’s symmetric λ-calculus [4] also has a clear notion of duality, though he does not consider evaluation strategies. Wadler’s DC demonstrates the duality without any of the limitations of the above.

Chapter 2

Preparation

This chapter provides the necessary background knowledge on various aspects of the project, primarily Agda and the Dual Calculus. I also give the starting point of the project and list the tools used for the implementation.

2.1 Curry–Howard Correspondence

The Curry–Howard correspondence relates programs and mathematical proofs. Haskell Curry made the first observation of this relationship in 1934 when he noted that the types of combinators, a kind of programming construct, can be interpreted as axiom schemes for a variant of intuitionistic logic [13]. He later extended this [15] to a relationship between Hilbert-style deduction systems and combinatory logic [41, 14]. William Howard then spelt out this correspondence explicitly when he demonstrated a direct relationship between Intuitionistic Natural Deduction [20] and the Simply-Typed λ-calculus (STLC) [23].

Despite the prevailing thought of the time being that classical logic had no computational interpretation, the correspondence was later extended to classical logic. This was when Tim Griffin [21] demonstrated that the type call/cc – a programming language construct giving explicit control over the program continuation – corresponded to Peirce's Law, a law that does not hold in intuitionistic logic.

Table 2.1 lists some instances of the Curry–Howard correspondence, for example, propositions correspond to types and proofs to programs. This means that a proof of a proposition A directly relates to a program of type A. Double negation translation is a method for embedding a classical logic system into an intuitionistic one; it's computational equivalent is continuation-passing style transformation.

Table 2.1: A non-exhaustive list of examples of the Curry–Howard correspondence

Formal Logic	**Programming Language Theory**
Proposition	Type
Proof	Program
Proof Simplification	Program Execution
Conjunction	Product Type
Disjunction	Sum Type
Implication	Function Type
Universal Quantification	Dependent Product Type
Peirce's Law	`call/cc`
Double Negation translation	Continuation-Passing Style Transformation

2.2 The Dual Calculus

The Dual Calculus can be interpreted as a language of proofs in Gentzen's classical sequent calculus, or as a programming language where the control stack is explicit. It is designed to exhibit both the de Morgan duality between Call-by-Value and Call-by-Name, and the computational duality between producers and consumers of values. The following section outlines the calculus.

2.2.1 Syntax

The DC's type system consists of a base type X as well as conjunction, disjunction, and negation.

Definition 2.2.1 (Grammar of Types)**.**

Type $\quad A, B, C ::= X \mid A \,\&\, B \mid A \vee B \mid \neg A$

These types are introduced by terms: for example, the conjunctive type is constructed by $\langle M, N\rangle$ and disjunctive type by an injection $\langle M\rangle\mathrm{inl}$ or $\langle N\rangle\mathrm{inr}$. Coterms eliminate types by consuming values: $\mathrm{fst}[K]$ consumes a conjunction, and $[K, L]$ consumes a disjunction by case analysis. Statements are constructed from a term and a coterm with the same type, while terms and coterms can be constructed from statements by covariable and variable abstraction. The computational and logical interpretation of these constructs will be explained later.

Definition 2.2.2 (Syntax)**.**

Term $\quad M, N ::= x \mid \langle M, N\rangle \mid \langle M\rangle\mathrm{inl} \mid \langle N\rangle\mathrm{inr} \mid [K]\mathrm{not} \mid (S).\alpha$

Coterm $\quad K, L ::= \alpha \mid [K, L] \mid \mathrm{fst}[K] \mid \mathrm{snd}[L] \mid \mathrm{not}\langle M\rangle \mid x.(S)$

Statement $\quad S ::= M \bullet K$

2.2.2 Typing Relation

The DC has three different typing judgements: right sequents, left sequents and centre sequents. They judge terms, coterms and statements respectively. Since all three syntactic forms can contain free (co)variables, each typing judgement includes two typing environments: the antecedent (for variables), and the succedent (for covariables).

Definition 2.2.3 (Grammar of Antecedents and Succedents)**.**

Antecedent $\quad \Gamma = x_1 : A_1, ..., x_m : A_m$

Succedent $\quad \Theta = \alpha_1 : B_1, ..., \alpha_m : B_m$

Definition 2.2.4 (Sequents)**.**

Right Sequent $\quad \Gamma \rightarrow \Theta \mid M : A$

Left Sequent $\quad K : A \mid \Gamma \rightarrow \Theta$

Centre Sequent $\quad \Gamma \mid S \mapsto \Theta$

.

Definition 2.2.5 (Typing)**.** The typing judgements are mutually inductively defined by the rules and axioms in Figure 2.2.1

Figure 2.2.1: The typing rules for the Dual Calculus

$$\frac{}{\Gamma, x : A \boldsymbol{\rightarrow} \Theta \mid x : A}\ (\textsc{IdR}) \qquad \frac{}{\alpha : A \mid \Gamma \rightarrow \Theta, \alpha : A}\ (\textsc{IdL})$$

$$\frac{\Gamma \boldsymbol{\rightarrow} \Theta \mid M : A \qquad \Gamma \boldsymbol{\rightarrow} \Theta \mid N : B}{\Gamma \boldsymbol{\rightarrow} \Theta \mid \langle M, N\rangle : A \,\&\, B}\ (\&\mathrm{R}) \qquad \frac{K : A \mid \Gamma \rightarrow \Theta \qquad L : B \mid \Gamma \rightarrow \Theta}{[K, L] : A \vee B \mid \Gamma \rightarrow \Theta}\ (\vee\mathrm{L})$$

$$\frac{\Gamma \boldsymbol{\rightarrow} \Theta \mid M : A}{\Gamma \boldsymbol{\rightarrow} \Theta \mid \langle M\rangle\mathrm{inl} : A \vee B}\ (\vee\mathrm{R1}) \qquad \frac{K : A \mid \Gamma \rightarrow \Theta}{\mathrm{fst}[K] : A \,\&\, B \mid \Gamma \rightarrow \Theta}\ (\&\mathrm{L1})$$

$$\frac{\Gamma \boldsymbol{\rightarrow} \Theta \mid N : B}{\Gamma \boldsymbol{\rightarrow} \Theta \mid \langle N\rangle\mathrm{inr} : A \vee B}\ (\vee\mathrm{R2}) \qquad \frac{L : B \mid \Gamma \rightarrow \Theta}{\mathrm{snd}[L] : A \,\&\, B \mid \Gamma \rightarrow \Theta}\ (\&\mathrm{L2})$$

$$\frac{K : A \mid \Gamma \rightarrow \Theta}{\Gamma \boldsymbol{\rightarrow} \Theta \mid [K]\mathrm{not} : \neg A}\ (\neg\mathrm{R}) \qquad \frac{\Gamma \boldsymbol{\rightarrow} \Theta \mid M : A}{\mathrm{not}\langle M\rangle : \neg A \mid \Gamma \rightarrow \Theta}\ (\neg\mathrm{L})$$

$$\frac{\Gamma \mid S \mapsto \Theta, \alpha : A}{\Gamma \boldsymbol{\rightarrow} \Theta \mid (S).\alpha : A}\ (\mathrm{IR}) \qquad \frac{\Gamma, x : A \mid S \mapsto \Theta}{x.(S) : A \mid \Gamma \rightarrow \Theta}\ (\mathrm{IL})$$

$$\frac{\Gamma \boldsymbol{\rightarrow} \Theta \mid M : A \qquad K : A \mid \Gamma \rightarrow \Theta}{\Gamma \mid M \bullet K \mapsto \Theta}\ (\textsc{Cut})$$

There are no elimination forms in the rules, instead each term introduction rule has a symmetric coterm introduction rule. Negation converts between right sequents and left sequents, while covariable abstraction converts a statement with a free covariable to a term. These rules hint at the notion of duality in the DC which is discussed in depth in Section 3.5.

Initially, some of the typing rules may seem unintuitive: for example, $\&L1$ states that if a coterm K has type A then $\mathrm{fst}[K]$ has type $A \,\&\, B$. This is the opposite of what one might expect from first projection. The reasons for this, however, become clear when one considers the logical and computational interpretation of the DC.

2.2.3 Curry–Howard Correspondence for the Dual Calculus

If we were to erase all the terms from the typing rules of a computational calculus, we would be left with a purely logical inference system. Applying this to the DC produces exactly Gentzen's Classical Sequent Calculus, as Figure 2.2.3 demonstrates.

Types The types of the DC correspond to propositions in classical logic, and a term, coterm, or statement of a given type represents a proof of that corresponding proposition.

Terms, Coterms, and Statements A term proves a proposition in the succedent. A coterm proves a proposition in the antecedent, which is equivalent to refuting it in the succedent [37]. A statement represents a

Figure 2.2.2: Side by side comparison of some sequent calculus and Dual Calculus inference rules

$$\textsc{Dual-IdR}\quad \frac{}{\Gamma, x : A \to \Theta \mid x : A} \qquad \textsc{Dual-IdL}\quad \frac{}{\alpha : A \mid \Gamma \to \Theta, \alpha : A} \qquad \textsc{Sequent-Id}\quad \frac{}{\Gamma, A \to \Theta, A}$$

$$\textsc{Dual-\&R}\quad \frac{\Gamma \to \Theta \mid M : A \qquad \Gamma \to \Theta \mid N : B}{\Gamma \to \Theta \mid \langle M, N \rangle : A \,\&\, B} \qquad \textsc{Sequent-\&R}\quad \frac{\Gamma \to \Theta, A \qquad \Gamma \to \Theta, B}{\Gamma \to \Theta, A \,\&\, B}$$

$$\textsc{Dual-\&L1}\quad \frac{K : A \mid \Gamma \to \Theta}{\mathrm{fst}[K] : A \,\&\, B \mid \Gamma \to \Theta} \qquad \textsc{Sequent-\&L1}\quad \frac{A, \Gamma \to \Theta}{A \,\&\, B, \Gamma \to \Theta}$$

$$\textsc{Dual-Cut}\quad \frac{\Gamma \to \Theta \mid M : A \qquad K : A \mid \Gamma \to \Theta}{\Gamma \mid M \bullet K \mapsto \Theta} \qquad \textsc{Sequent-Cut}\quad \frac{\Gamma \to \Theta, A \qquad A, \Gamma \to \Theta}{\Gamma \to \Theta}$$

contradiction between a term and a coterm. Covariable abstraction derives a term of type A from a statement with a free covariable of the same type. The logical interpretation of this is proof by contradiction: deriving a contradiction from the assumption of $\neg A$ proves A. This is a key feature of classical logic, as its validity relies on the Law of Double Negation Elimination ($\neg\neg A \supset A$). The logical interpretations of terms, coterms, and statements defined in more detail in Table 2.2.

Sequents Variables and covariables label antecedents and succedents, which themselves are respectively interpreted as a conjunction or disjunction of propositions. Sequents are interpreted as: the conjunction of the propositions in the antecedent implies the disjunction of the propositions in the succedent. For example the right sequent

$$x_1 : A_1, ..., x_m : A_m \to \alpha_1 : B_1, ..., \alpha_m : B_m \mid M : A$$

is interpreted to state that if all of the propositions A_i hold then some B_i *or* A holds.

The relationship to classical logic we have outlined gives us the following proposition.

Proposition 2.2.6. *A proposition A is provable in classical logic iff there exists a closed Dual Calculus term M such that $\varnothing \to \varnothing \mid M : A$.*

Wadler also defines a CPS transformation that embeds the syntax of the DC into the STLC, a calculus that corresponds to an intuitionistic, rather than classical, logic. This gives us the surprising result that intuitionistic logic is no less expressive than classical logic. This CPS transformation is explored in Section 3.4.

2.2.4 Computational Content

Types Conjunctive and disjunctive types are standard, and the logical equivalence between $\neg A$ and $A \supset \bot$ reveals that $\neg A$ is the type of a function taking values of type A and returning nothing. Such a function is known as a *continuation.*

Terms, Coterms, and Statements Terms produce values, while coterms consume them. Statements pipe together a term and a coterm. Computation in the DC is defined by its reduction relations, both of which transform statements; modifying its component term and coterm in parallel. These reduction relations are detailed in Section 3.3. The computational meaning of terms, coterms and statements are detailed in Table 2.2.

Sequents The computational meaning of a sequent is that, with a value supplied to every variable in the antecedent, the computation will result in a value being passed to some covariable in the succedent. Considering this, the interpretation of the right sequent

$$x_1 : A_1, ..., x_m : A_m \rightarrow \alpha_1 : B_1, ..., \alpha_m : B_m \mid M : A$$

is: if one supplies a value of type A_i to each x_i, then evaluating the term M will either return a value of type A or pass a value of type B_i to some α_i. Similarly, the interpretation of the left sequent

$$K : A \mid x_1 : A_1, ..., x_m : A_m \rightarrow \alpha_1 : B_1, ..., \alpha_m : B_m$$

is: if one supplies a value of type A_i to each x_i and a value of type A to the coterm K, then evaluation will pass a value of type B_i to some α_i. The interpretation of the centre sequent

$$x_1 : A_1, ..., x_m : A_m \mid S \mapsto \alpha_1 : B_1, ..., \alpha_m : B_m$$

is: if each variable x_i is given a value of type A_i, then evaluation will return a value of type B_i to some α_i.

2.2.5 Implication

The omission of any discussion of functions and implication up to this point was deliberate. This was because the dual of implication is not a natural concept, so it does not fit nicely into our framework. Despite this, Wadler does include implication types, abstraction, and application in the syntax of the DC, forcing theorems and definitions to be restricted to only those DC programs that do not include implication. This is resolved by deriving implication from the other connectives, though the definition is dependent on which reduction relation is employed. The reason for this is revealed by the definition of values and covalues.

Definition 2.2.7 (Values and Covalues)**.**

Value	$V, W ::= x \mid \langle V, W\rangle \mid \langle V\rangle\text{inl} \mid \langle W\rangle\text{inr} \mid [K]\text{not} \mid \lambda x.N$
Covalue	$P, Q ::= \alpha \mid [P, Q] \mid \text{fst}[P] \mid \text{snd}[Q] \mid \text{not}\langle M\rangle \mid M \mathbin{@} Q$

A Call-by-Value reduction relation requires a function abstraction to be a value, while Call-by-Name evaluation requires that function application is a covalue. This is why we require two separate definitions of implication.

Proposition 2.2.8. *With Call-by-Value reduction, implication is defined as*

$$\begin{aligned} A \supset B &\equiv \neg A \,\&\, \neg B \\ \lambda x.N &\equiv [z.(z \bullet \text{fst}[x.(z \bullet \text{snd}[\text{not}\langle N\rangle])])]\text{not} \\ M \mathbin{@} L &\equiv \text{not}\langle\langle M, [L]\text{not}\rangle\rangle \end{aligned}$$

Proposition 2.2.9. *With Call-by-Name reduction, implication is defined as*

Table 2.2: The computational and logical interpretations of the terms, coterms, and statements of the Dual Calculus

	Computational Interpretation	Logical Interpretation
Terms		
x	Produces the value given to the variable x.	Proves the proposition A by propagating the hypothesis x of proposition A.
$\langle M, N\rangle$	Produces a value of type $A \,\&\, B$, a pair of the values of type A and B that are yielded by M and N.	Proves the proposition $A \,\&\, B$, with a pair of proofs of propositions A and B: M and N.
$\langle M\rangle\text{inl}/\langle N\rangle\text{inr}$	$\langle M\rangle$inl produces a value of type $A \vee B$ given by the left injection of M, a value of type A. Similar for the term $\langle N\rangle$inr.	$\langle M\rangle$inl proves the proposition $A \vee B$ by left injection of M, a proof of proposition A. Similar for the term $\langle N\rangle$inr.
$[K]\text{not}$	Produces a continuation of type $\neg A$ (equivalent to $A \supset \bot$), this takes a value of type A that is consumed by coterm K.	Proves the proposition $\neg A$ by using the coterm K, a refutation of A.
$(S).\alpha$	Executes the statement S, yielding the value of type A passed to the covariable α.	Proves the proposition A by assuming the existence of α, a refutation of the A, and deriving the contradiction S.
Coterms		
α	Consumes a value, giving it out to the covariable α.	Refutes the proposition A by propagating the hypothesis α of the refutation of A.
$[K, L]$	Consumes a value of type $A \vee B$ and passes it to the relevant component coterm (K or L), depending on whether the said value is a left injection of a value of type A or a right injection of a value of type B.	Refutes the proposition $A \vee B$ with a pair of refutations, K and L, of the propositions A and B respectively.
$\text{fst}[K]/\text{snd}[L]$	fst$[K]$ consumes a value of type $A \,\&\, B$ and projects out the value of type A which is then consumed by the coterm K. Similar for snd$[L]$.	fst$[K]$ refutes the proposition $A \,\&\, B$, by projecting the coterm K, a refutation of A. Similar for snd$[L]$.
$\text{not}\langle M\rangle$	Consumes a continuation of type $\neg A$ by passing it the value of type A produced by the term M.	Refutes the proposition $\neg A$ by using the proof of proposition A given by the term M.
$x.(S)$	Consumes a value of type A, binds this to the variable x and then executes the statement S.	Refutes the proposition A, by assuming the existence of x, a proof of A, and deriving the contradiction S.
Statements		
$M \bullet K$	Pipes together a term and a coterm, feeding the value of type A produced by term M to be consumed by coterm K.	Represents a contradiction between the term M, a proof of A, and coterm K, a refutation of A.

$$\begin{aligned} A \supset B &\equiv \neg A \vee B \\ \lambda x.N &\equiv (\langle[x.(\langle N\rangle\mathrm{inr} \bullet \gamma)]\mathrm{not}\rangle\mathrm{inl} \bullet \gamma).\gamma \\ M @ L &\equiv [\mathrm{not}\langle M\rangle, L] \end{aligned}$$

The typing inference rules for function application and abstraction are given below. They can be derived from the rules given in Figure 2.2.1.

Definition 2.2.10 (Typing for implication)**.**

$$\frac{x : A, \Gamma \to \Theta \mid N : B}{\Gamma \to \Theta \mid \lambda x.N : A \supset B}\ (\supset\mathrm{R}) \qquad \frac{\Gamma \to \Theta \mid M : A \qquad L : B \mid \Gamma \to \Theta}{M @ L : A \supset B \mid \Gamma \to \Theta}\ (\supset\mathrm{L})$$

2.3 Agda

This section introduces the dependently typed functional programming language Agda. Through the Curry–Howard correspondence and intuitionistic type theory Agda can also be used as a proof assistant [32].

2.3.1 Agda as a Functional Programming Language

Despite not being its main use case, Agda can be used as a normal functional programming language; just like Haskell or ML. An important part of programming in Agda is pattern matching over algebraic data types, we can declare these data types as follows: data Name : Set where, where Set is the ‘type of types’ [1] The classic example of a data type in Agda is the natural numbers, which are defined below. The data type has two constructors: zero and suc.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Functions are defined in Agda by declaring the type of the function (dependent types make inference impossible) followed by defining its behaviour. To define a function over a data type we must pattern match over all possible cases, and define the behaviour for each. Agda syntax is flexible: allowing mixfix operators by using ‘_’ where an argument should be expected, as well as the use of Unicode characters. Addition over the natural numbers is given as an example below.

```
_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)
```

This defines an operator that takes two natural numbers, given each side of the + symbol, and returns a natural number.

It is also possible to parameterise data types by other types, for example, the list of elements of an arbitrary type A is defined below.

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

[1]This may raise the question: What is the type of Set? Well, it’s Set1. Naturally, Set1 has type Set2, Set2 has type Set3, and so on and so on.

2.3.2 Dependent Types

Given a type $A : U$, where U is a universe of types, we can define a family of types $B : A \to U$ that produces a type $B(x) : U$ for each term $x : A$. The family of types given by B is known as a *dependent type*, and they are what gives Agda much of its power. From a dependent type $B : A \to U$, we can define the type of *dependent functions* $\prod_{x:A} B(x)$. Terms of this type take a term $x : A$ and return a term of type $B(x)$. Note that if $B : A \to U$ is a constant function then $\prod_{x:A} B(x)$ is equivalent to $A \to B$, as B does not depend on x.

In Agda these dependent functions are represented as (x : A) → B x. A simple example of this is the polymorphic map function given below. Note that curly braces are used to declare a function argument to be *implicit*, meaning that the type checker will try to work out the value of the argument independently.

```
map : {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x :: xs) = f x :: map f xs
```

Dependent types are very powerful and can be used for many impressive things, for example defining the types of lists of a specific length. This definition of this is given below.

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Vec *A* is a family of types indexed by the natural numbers: ℕ → Set. This means that for each $n \in \mathbb{N}$, Vec *A* *n* is a separate type. Vec is parameterised on type A and indexed by the natural numbers. _::_ is an example of a dependent function, with the type of the Vec it both takes and returns dependent on the implicit argument *n*.

Dependent data types such as this have many uses. For example, we can use the Vec datatype to define the head operation on only non-empty lists.

```
head : {A : Set}{n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

Despite only defining one case, the type checker recognises this as an exhaustive pattern match, as [] cannot produce a list of type Vec *A* (suc *n*). This means that taking the head of an empty list, an invalid operation that should be avoided, will not type-check.

2.3.3 Agda as a Proof Assistant

The Curry–Howard correspondence tells us that producing a term of A is equivalent to proving the proposition A, this is how Agda is used as a proof assistant.

Many of the proofs in this project were proofs of equality between terms, however, Agda defines multiple types of equality. *Definitional equality* is the equality described in function definitions. For example, if we consider the definition of addition given above, we can see that (suc *m*) + *n* is definitionally equal to suc (*m* + *n*). Definitional equality implies the other type of equality we're interested in: *propositional equality*. This is defined as:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

For any type A and term *x*, refl proves that $x \equiv x$, therefore every term is equal to itself and this is the only way to prove terms are equal. From this definition, we can derive the other properties necessary for equality:

substitution, transitivity, and symmetry. When we prove that two terms are equivalent, we will be proving propositional equality.

Universal quantification is the logical interpretation of dependent functions. The ∀ syntax in Agda is a dependent function type for which the type of the argument can be inferred. In general, if we have a variable x of type A and a dependent type B x, then we can produce an Agda term of type ∀ x → B x if, for every term M : A we can produce a term of type B M. Therefore, a proof of the proposition ∀ x → B x takes the form of λ x → N x, where N x is a term of type B x. A simple example of this, an inductive proof of the associativity of addition, is given below:

```
+-assoc : ∀ m n p → (m + n) + p ≡ m + (n + p)
+-assoc zero n p = refl
+-assoc (suc m) n p = cong suc (+-assoc m n p)
```

2.3.4 Examples of Agda Proofs

Here follows a proof of the commutativity of addition, with the aim to familiarise the reader with what proofs in Agda look like.

This proof requires us to prove a couple of lemmas: the first is that zero is the identity when on the right-hand side of an addition, formally stated as $\forall n \in \mathbb{N}.\, n + 0 \equiv n$. First, we must provide a type signature for the lemma, which looks a lot like the mathematical formulation.

```
+-identityʳ : ∀ (m : ℕ) → m + zero ≡ m
```

We prove this lemma by induction on m. For the base case we instantiate m to zero; this means we must provide a proof that zero + zero ≡ zero. The definition of _+_ asserts this as a definitional equality, therefore refl is sufficient proof.

```
+-identityʳ zero = refl
```

For the inductive step, m is instantiated as (suc m), therefore we must prove (suc m) + zero ≡ suc m. _+_ defines (suc m) + zero to be equal to suc(m + zero) so we must show that this is equivalent to suc m. To prove this we make use of a function from Agda's standard library: cong, which states that the results of applying a function to two terms will be equivalent if the two terms are equivalent.

```
cong : ∀ {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl
```

We can use suc as the function argument for cong and appeal to our inductive hypothesis to produce the proof that the two terms are equal.

```
+-identityʳ (suc m) = cong suc (+-identityʳ m)
```

The second lemma we must prove is that suc can be pushed from the second argument of an addition to the outside. Once again, the type signature looks just like the mathematical formulation.

```
+-suc : ∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
```

This is also proved by induction and congruence, in a similar manner to the previous lemma. Hence, we omit its proof.

Now we can prove the commutativity of addition: $\forall m, n \in \mathbb{N}.\, m + n \equiv n + m$. The type signature is as expected.

```
+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
```

We prove this by induction on n, so our base case requires a proof that m + zero equals zero + m. Since zero + m is definitionally equal to m we can prove this by our +-identity^r lemma.

```
+-comm m zero = +-identityʳ m
```

The inductive case is slightly more complex, so for better clarity we will make use of the Agda standard library's equational reasoning.

```
+-comm m (suc n) = begin
  m + (suc n)  ≡⟨ +-suc m n ⟩
  suc (m + n)  ≡⟨ cong suc (+-comm m n) ⟩
  (suc(n + m)) □
```

Equational reasoning allows us to prove an equivalence by writing a series of equivalent equations separated by proofs of their equivalence. Here we must prove that m + (suc n) is equal to (suc n) + m, we prove this in two separate steps. The first step is made by using our +-suc lemma to show that m + (suc n) equals suc (m + n). The second step uses cong and our inductive hypothesis to demonstrate that suc (m + n) is equivalent to suc (n + m), which itself is definitionally equal to (suc n) + m, completing the proof.

2.4 Software Engineering

2.4.1 Starting Point

I am not aware of other attempts to formalise the Dual Calculus is any proof assistant.

Before I started working on this project I had no experience with Agda or any other proof assistant; the extent of my knowledge was the brief discussion they receive in the Part IB Logic and Proof course [37]. As soon as I knew my project would involve using Agda I started studying PLFA [48] to teach myself how to progam in it. I also studied sections of the textbook *Practical Foundations for Programming Languages* [22] to familiarise myself with the Curry–Howard correspondence. The intrinsically-typed Agda formalisation of the λ-calculus in PLFA proved useful to me throughout my project.

2.4.2 Tools and Design

I used *banacorn's agda-mode* extension for Visual Studio Code, which allows for the execution of Agda commands and simple Unicode input in VSCode, making it an effective IDE for writing in Agda. The Agda distribution I used has a LaTeXback-end allowing easy typesetting of Agda code, which I used to prepare this dissertation. The source code of my project is stored in a git repository hosted on GitHub and I downloaded a backup of the repository to a memory stick twice a week.

The DC formalisation is not a standard software project as it does not require testing in the normal sense of the word: the proofs I have produced guarantee that the formalisation is correct, and the Agda type checker guarantees that these proofs are correct. Despite this, to give me confidence in its correctness, I used my DC formalisation to prove some theorems of classical logic and recreate some examples from the original paper. As well as this, I simulated various computational calculi in the DC formalisation, and demonstrated that it is capable of simple arithmetic.

Chapter 3

Implementation

This chapter begins with an overview of my repository before presenting the work I have completed. The second section of this chapter defines the syntax of the DC, and the next two define its semantics: operational and denotational. The final section of this chapter discusses the duality of each of the previous sections.

3.1 Repository Overview

Most of the final source code for my project is split into three main modules: Syntax, Operational Semantics, and Denotational Semantics. Each folder includes Agda source code files that implement the key definitions and proofs, separated thematically into different files. Every folder contains a file that demonstrates the duality of the relevant concept, as well as a file containing any code that I wrote to evaluate that part of the project. The source code defining the simulation of other calculi in the DC, that I use to evaluate the project, does not fit into any of the three described categories. This is presented graphically in Figure 3.1.1.

All the code for this project was written from scratch, though in some cases is based on standard methods for certain common problems that I extended to the DC [28, 2].

3.2 Syntax

This section presents the important definitions of my DC formalisation, and discusses and justifies any design choices I made.

3.2.1 Intrinsically-typed representation of syntax

There are two main ways to encode a formal language in a proof assistant, they are known as *extrinsic* and *intrinsic* encoding [40]. Terms in an extrinsic definition have a meaning independent of how they are typed, their typings simply represent properties of the language. In contrast, terms in an intrinsic definition do not exist independently of their typing; it is to the typing judgements, rather than the terms themselves, that meaning is assigned. Using *inductive families* [17] to represent syntax, Altenkirch and Reus [2], Bellegarde and Hook [6], and Bird and Paterson [8] all showed how intrinsic typing can be used to enforce both *type-safety* and *scope-safety* of terms at the type level.

Scope-safe terms require that every variable is bound by some binder, or is explicitly tracked in some typing environment, known as a *context*. Using an inductive family, indexed by a set of variables, to represent the terms of a language keeps track of the scoping information at the type level, meaning that type-checking will guarantee

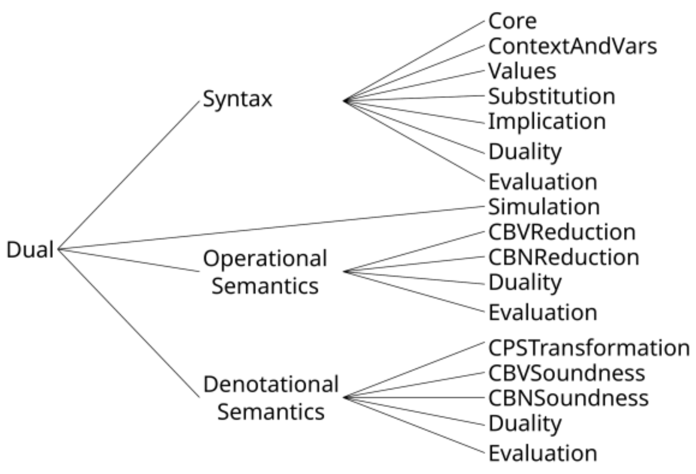

Figure 3.1.1: A tree outlining the structure of the source code repository

that a term is well-scoped. This can be extended to enforce type-safety in typed languages by indexing the family by a type as well. This guarantees that a term that type-checks in Agda is both well-typed and well-scoped in the defined language.

An extrinsically typed representation of a language requires representing variables as either strings or numeric de Bruijn indices [16], both of which have their issues. Using strings makes maintaining α-equivalence difficult. And the requirement to correctly modify all indices each time the scope changes when using de Bruijn indices can cause difficult to debug errors, as Cristina Matache discovered in her Part II Project [27]. In an intrinsically-typed representation, one can define an inductive family to represent well-typed and well-scoped de Bruijn indices. As such, when defining operations that changes scope, Agda's type system will force the user to modify the indices correctly, and assist them in doing so.

For the reasons above, I decided to use an intrinsically-typed representation of the syntax of the DC.

3.2.2 Core Syntax

The types of the DC can be represented in Agda with an inductive data type.

```
data Type : Set where
  X : Type
  _`×_ : Type → Type → Type
  _`+_ : Type → Type → Type
  `¬_ : Type → Type
```

As per intrinsic-typing, terms, coterms, and statements must be indexed by two sets representing the scope of the program. We implement these sets, known as contexts, as a list of Types.

```
data Context : Set where
  ∅ : Context
  _,_ : Context → Type → Context
```

We then define free (co)variables as an inductive family indexed by a Context and Type. There are two constructors for this datatype: `Z proves that the variable is in a Context by showing it is the most recently bound variable, `S uses evidence that the variable is in a Context Γ to prove that it is in Γ extended with another variable. For example, the third most recently bound variable can be constructed with `S (`S (`Z)). The constructors are analogous to the zero and suc constructors for ℕ; indeed variables can be seen as type- and scope-safe de Bruijn indices.

```
data _∋_ : Context → Type → Set where
  `Z : ∀ {Γ A}         → Γ , A ∋ A
  `S : ∀ {Γ A B} → Γ ∋ A → Γ , B ∋ A
```

Note here that we make no distinction between variables and covariables. Whether an instance of the _∋_ datatype (I will call such an instance a var from now on) is a variable or a covariable is decided entirely by whether the Context the var is indexed by is used as an antecedent or a succedent.

We now define DC terms, coterms, and statements. With our intrinsically-typed representation, these are all indexed by two Contexts (the antecedent and succedent), while terms and coterms are also indexed by a Type. This means we don't really define terms, coterms, and statements – we define left sequents, right sequents, and centre sequents, which cannot exist independently of their Type and Contexts. Note that the constructors of these sequents closely resemble the typing relation of the DC.

```
data _⟶_|_ : Context → Context → Type → Set

data _|_⟶_ : Type → Context → Context → Set

data _↦_ : Context → Context → Set
data _⟶_|_ where

  `_ : ∀ {Γ Θ A}
    → Γ ∋ A
    → Γ ⟶ Θ | A

  `⟨_,_⟩ : ∀ {Γ Θ A B}
    → Γ ⟶ Θ | A
    → Γ ⟶ Θ | B
    → Γ ⟶ Θ | A `× B

  inl⟨_⟩ : ∀ {Γ Θ A B}
    → Γ ⟶ Θ | A
    → Γ ⟶ Θ | A `+ B

  inr⟨_⟩ : ∀ {Γ Θ A B}
    → Γ ⟶ Θ | B
    → Γ ⟶ Θ | A `+ B

  not[_] : ∀ {Γ Θ A}
    → A | Γ ⟶ Θ
    → Γ ⟶ Θ | `¬ A

  μθ : ∀ {Γ Θ A}
    → Γ ⟼ Θ , A
    → Γ ⟶ Θ | A
```

```
data _|_⟶_ where

  `_ : ∀ {Γ Θ A}
    → Θ ∋ A
    → A | Γ ⟶ Θ

  `[_,_] : ∀ {Γ Θ A B}
    → A | Γ ⟶ Θ
    → B | Γ ⟶ Θ
    → A `+ B | Γ ⟶ Θ

  fst[_] : ∀ {Γ Θ A B}
    → A | Γ ⟶ Θ
    → A `× B | Γ ⟶ Θ

  snd[_] : ∀ {Γ Θ A B}
    → B | Γ ⟶ Θ
    → A `× B | Γ ⟶ Θ

  not⟨_⟩ : ∀ {Γ Θ A}
    → Γ ⟶ Θ | A
    → `¬ A | Γ ⟶ Θ

  μγ : ∀ {Γ Θ A}
    → Γ , A ⟼ Θ
    → A | Γ ⟶ Θ
```

```
data _⟼_ where
  _●_ : ∀ {Γ Θ A} → Γ ⟶ Θ | A → A | Γ ⟶ Θ → Γ ⟼ Θ
```

3.2.3 Values and Covalues

We introduce both a Value and Covalue inductive family, indexed on right sequents and left sequents respectively. The Agda term Value M proves that the DC term represented by M is a value. The constructors match the definitions of values and covalues from the original paper. A couple of examples are given below.

```
data Value where
  V-var : ∀ {Γ Θ A} {x : Γ ∋ A}
    → Value {Θ = Θ} (` x)

  V-prod : ∀ {Γ Θ A B} {M : Γ ⟶ Θ | A} {N : Γ ⟶ Θ | B}
    → Value M → Value N
    → Value `⟨ M , N ⟩
```

We also introduce two new inductive families: TermValue and CotermValue. A TermValue is a dependent product made up of a right sequent and a proof that it is a Value. A CotermValue is the equivalent for left sequents.

```
TermValue : Context → Context → Type → Set
TermValue Γ Θ A = Σ (Γ ⟶ Θ ∣ A) Value
```

```
CotermValue : Context → Context → Type → Set
CotermValue Γ Θ A = Σ (A ∣ Γ ⟶ Θ) Covalue
```

3.2.4 Renaming and Substitution

As we have seen, the Dual Calculus includes variable binding, therefore evaluation of DC programs requires a substitution operation. Such an operation is simple to define on paper, provided one takes care to avoid variable capture. However, substitution is notoriously difficult to formalise in a proof assistant, even for a language as simple as the STLC; it is the subject of significant research [1, 28]. It is common for a large part of a language formalisation task to consist of dealing with the nuances of substitution, and my project was no exception. In fact, adapting existing approaches to handle a mutually inductively defined syntax with two kinds of variables made this an even bigger task. Despite the considerable effort put into overcoming these issues, listing the technical details of the rather involved implementation would detract from the main themes of my dissertaion. As such, I give a high-level overview of the definition of substitution and defer the details to an Appendix.

The CbV reduction relation requires only that we define the substitution of TermValue or coterm into a statement [1]. The type signature of these operations are given below; they are derived from Kimura's doctoral thesis [25].

```
_ᵛ⟨_/⟩ˢ : ∀ {Γ Θ A}
  → Γ , A ↦ Θ
  → TermValue Γ Θ A
  ---------
  → Γ ↦ Θ


_[_/]ˢ : ∀ {Γ Θ A}
  → Γ ↦ Θ , A
  → A ∣ Γ ⟶ Θ
  -----
  → Γ ↦ Θ
```

However, it is impossible to define these operations from scratch, as we cannot recursively apply such an operation to the body of a (co)variable abstraction due to the presence of a new var. As such, we define it in terms of a general, mutually inductive, simultaneous substitution operation.

This operation makes use of *context maps*, representing the operation on variables that the substitution lifts to an operation on sequents. A context map Γ –[T]→ Δ, represents an operation that, for an arbitrary Type A, transforms elements of $\Gamma \ni A$ to $T\ \Delta\ A$. In the case of the DC, sequents are indexed by two Contexts, both of which are transformed independently, with two separate context maps.

Note that Fix$_n$ $T\ \Gamma$ fixes the nth Context of T, some family indexed by two Contexts and a Type, to Γ.

```
sub-T : ∀ {T A C Γ Θ Γ′ Θ′} → TermKit T → CotermKit C
  → Γ –[ (Fix₂ T Θ′) ]→ Γ′ → Θ –[ (Fix₁ C Γ′) ]→ Θ′
  → Γ ⟶ Θ ∣ A → Γ′ ⟶ Θ′ ∣ A
sub-C : ∀ {T A C Γ Θ Γ′ Θ′} → TermKit T → CotermKit C
  → Γ –[ (Fix₂ T Θ′) ]→ Γ′ → Θ –[ (Fix₁ C Γ′) ]→ Θ′
```

[1]Dually, CbN reduction requires substituting a term or CotermValue into a statement

```
  → A ∣ Γ ⟶ Θ → A ∣ Γ′ ⟶ Θ′
sub-S : ∀ {T C Γ Θ Γ′ Θ′} → TermKit T → CotermKit C
  → Γ –[ (Fix₂ T Θ′) ]→ Γ′ → Θ –[ (Fix₁ C Γ′) ]→ Θ′
  → Γ ↦ Θ → Γ′ ↦ Θ′
```

This operation requires both a TermKit and CotermKit: *kits* provide a series of general functions required for the implementation of substitution and are based on the work of McBride [28]. Keller [24] provides an example of how they can be implemented using Agda records.

Kit operations allow us to define various combinators on context maps: sub-weaken weakens the Context that a context map maps into; sub-lift weakens both the Context it maps from, and the Context it maps into; fmap modifies the indexed family that the context map produces instances of.

One of the operations these kits provide is the weakening of a Context (extending it with another var); to implement this we must define *renaming*. Renaming, like substitution, is a sequent traversal, they both take an operation on vars and lift it to an operation on sequents. They differ only in that while substitution maps vars to sequents, renaming maps vars to vars. As such, the implementation of renaming is essentially a simpler version of substitution.

```
ren-T : ∀ {Γ Γ′ Θ Θ′ A} → Γ ↝ Γ′ → Θ ↝ Θ′ → Γ ⟶ Θ ∣ A → Γ′ ⟶ Θ′ ∣ A
ren-C : ∀ {Γ Γ′ Θ Θ′ A} → Γ ↝ Γ′ → Θ ↝ Θ′ → A ∣ Γ ⟶ Θ → A ∣ Γ′ ⟶ Θ′
ren-S : ∀ {Γ Γ′ Θ Θ′} → Γ ↝ Γ′ → Θ ↝ Θ′ → Γ ↦ Θ → Γ′ ↦ Θ′
```

A renaming map, _ ↝ _, is a special instance of a context map for which the T is instantiated to _∋_. We also define combinators on renaming maps: ren-weaken and ren-lift.

3.2.5 Implication

Implication is defined in terms of the other connectives, with different definitions for Call-by-Value and Call-by-Name.

```
_⇒ᵛ_ : Type → Type → Type
A ⇒ᵛ B = `¬ (A `× `¬ B)

_⇒ⁿ_ : Type → Type → Type
A ⇒ⁿ B = `¬ A `+ B
```

We earlier asserted that we can derive the inference rules for implication from the primitive inference rules. Intrinsic typing makes this proof easy. Note that wk and int are the derived structural rules of weakening and interchange.

```
λᵛ_ : ∀ {Γ Θ A B} → Γ , A ⟶ Θ ∣ B → Γ ⟶ Θ ∣ A ⇒ᵛ B
λⁿ_ : ∀ {Γ Θ A B} → Γ , A ⟶ Θ ∣ B → Γ ⟶ Θ ∣ A ⇒ⁿ B
_·ᵛ_ : ∀ {Γ Θ A B} → Γ ⟶ Θ ∣ A → B ∣ Γ ⟶ Θ → A ⇒ᵛ B ∣ Γ ⟶ Θ
_·ⁿ_ : ∀ {Γ Θ A B} → Γ ⟶ Θ ∣ A → B ∣ Γ ⟶ Θ → A ⇒ⁿ B ∣ Γ ⟶ Θ

λᵛ N = not[ μγ(γ 0 ● fst[ μγ (γ 1 ● snd[ not⟨ intΓᵗ (wkΓᵗ N) ⟩ ]) ]) ]
λⁿ N = μθ (inl⟨ not[ μγ(inr⟨ wkΘᵗ N ⟩ ● θ 0) ] ⟩ ● θ 0)
M ·ᵛ N = not⟨ `⟨ M , not[ N ] ⟩ ⟩
M ·ⁿ N = `[ not⟨ M ⟩ , N ]
```

3.3 Operational Semantics

This section describes the reduction relations of the DC.

3.3.1 Reduction Relations

The DC has two reduction relations, one Call-by-Value, the other Call-by-Name[2], as such, we define two inductive families to represent them: one for the Call-by-Value reduction relation, and one for Call-by-Name. Each reduction rule takes a statement to a statement, so these inductive families are indexed by two centre sequents: the sequent both before and after the rule is applied. The application of a reduction rule modifies the statement's component term and coterm in parallel. I will only discuss the CbV relation as the CbN relation is dual.

```
data _ˢ⟶ᵛ_ : ∀ {Γ Θ} → (Γ ⟼ Θ) → (Γ ⟼ Θ) → Set where
```

For calculi like the STLC, the intrinsically-typed reduction relation is defined between terms with the same type and context, therefore it doubles up as a proof of the Type Preservation theorem. Since centre sequents do not have a Type however, this reduction relation only proves a, still valuable, "Context Preservation" theorem.

The constructors of this inductive family correspond directly to the reduction rules laid out in the original paper. Note that many of the rules require proof that a given term is a Value.

```
βx₁ : ∀ {Γ Θ A B} {V : Γ ⟶ Θ | A} {W : Γ ⟶ Θ | B} {K : A | Γ ⟶ Θ} (v : Value V) (w : Value W)
  → `⟨ V , W ⟩ • fst[ K ] ˢ⟶ᵛ V • K

βx₂ : ∀ {Γ Θ A B} {V : Γ ⟶ Θ | A} {W : Γ ⟶ Θ | B} {L : B | Γ ⟶ Θ} (v : Value V) (w : Value W)
  → `⟨ V , W ⟩ • snd[ L ] ˢ⟶ᵛ W • L

β+₁ : ∀ {Γ Θ A B} {V : Γ ⟶ Θ | A} {K : A | Γ ⟶ Θ} {L : B | Γ ⟶ Θ} (v : Value V)
  → inl⟨ V ⟩ • `[ K , L ] ˢ⟶ᵛ V • K

β+₂ : ∀ {Γ Θ A B} {W : Γ ⟶ Θ | B} {K : A | Γ ⟶ Θ} {L : B | Γ ⟶ Θ} (w : Value W)
  → inr⟨ W ⟩ • `[ K , L ] ˢ⟶ᵛ W • L

β¬ : ∀ {Γ Θ A} {M : Γ ⟶ Θ | A} {K : A | Γ ⟶ Θ}
  → not[ K ] • not⟨ M ⟩ ˢ⟶ᵛ M • K

βL : ∀ {Γ Θ A} {V : Γ ⟶ Θ | A} {S : Γ , A ⟼ Θ} (v : Value V)

  → V • (μγ S) ˢ⟶ᵛ S ᵛ⟨ ⟨ V , v ⟩ /⟩ˢ

βR : ∀ {Γ Θ A} {K : A | Γ ⟶ Θ} {S : Γ ⟼ Θ , A}
  → (μθ S) • K ˢ⟶ᵛ S [ K /]ˢ
```

The most interesting examples of this relation are the reduction rules: βL and βR, which both include a substitution; substituting a left sequent in the case of βR and a TermValue in the case of βL.

I also defined two more inductive families, indexed by two centre sequents, to represent multi-step reduction: the reflexive-transitive closure of the reduction relation. An instance of this inductive family will type-check if one can prove that the first centre sequent will reduce to the second after an arbitrary number of reductions.

[2]Wadler defines a set of eleven reduction rules for each of these relations. However, three of these rules are 'expansion rules' and are not required for any of the properties I intend to prove. Therefore, I chose to omit these rules, defining only the congruence and reduction rules.

These proofs are built up through chains of reductions, much like with equational reasoning. A simple example of this relation follows.

```
example : ∀ {A B} → (V : ∅ ⟶ ∅ ∣ A) → (W : ∅ ⟶ ∅ ∣ B) → (K : A ∣ ∅ ⟶ ∅)
  → Value V → Value W
  → (not[ fst[ K ] ] ● not⟨ `⟨ V , W ⟩ ⟩) ˢ—↠ᵛ V ● K
example V W K v w = beginˢᵛ
  (not[ fst[ K ] ]) ● (not⟨ `⟨ V , W ⟩ ⟩) ˢ⟶ᵛ⟨ β¬ ⟩
  `⟨ V , W ⟩ ● fst[ K ] ˢ⟶ᵛ⟨ β×₁ v w ⟩
  V ● K                 ∎ˢᵛ
```

Many-step reduction invites us to consider the shared meaning of each sequent along the chain; a meaning that we now define formally.

3.4 Denotational Semantics

The following section defines a Denotational Semantics of the DC, interpreting DC constructs as Agda representations of sets and functions. This is done by implementing a version of the continuation-passing style transformations from the original paper. I then prove the soundness of said CPS transformations.

3.4.1 Continuation-Passing Style Transformation

The CPS transformation, as defined in the original paper, maps from the DC to the STLC. Implementing this would require us to formalise the STLC as well, so, to keep the focus on the DC, we decided to use Agda as the *target calculus*. This is valid because Agda has all the constructs of the target calculus that Wadler uses.

As we are implementing a denotational semantics of the DC, it is worth summarising the standard methodology in this field and how these must be modified for the DC.

A set-based model of a language like the STLC interprets types A as sets $[\![A]\!]$, and typing contexts $\Gamma = x_1 : A_1, ..., x_n : A_n$ are interpreted as products of the interpretations of the types of their component variables $[\![\Gamma]\!] = [\![A_1]\!] \times ... \times [\![A_m]\!]$. Terms $\Gamma \vdash M : A$ are interpreted as a function $[\![\Gamma]\!] \to [\![A]\!]$ mapping the interpretations of the term's free variables to some element of the denotation of the term's type. We can extend this approach to more intricate languages, which often requires more assumptions to be made about the structure of interpreted constructs: for example, the partial language PCF is interpreted as domains and continuous functions.

These definitions allow us to investigate properties of the language, such as whether the operational semantics preserve the meaning of a term: the soundness theorem. To establish this we first must prove the semantic substitution lemma, which characterises the denotation of $M[M'/x]$ as the composition of the denotations of M' and M.

While it may seem pretentious to call a CPS transformation "denotational semantics", the definitions and theorems I have outlined above fit into this framework well. With this in mind, we present the CbV denotational semantics of the DC (the CbN semantics are dual).

Again, we start by defining the operation for Types and Contexts. Types are interpreted as Sets, and Contexts as products of these Sets. We interpret the base type as the natural numbers to make evaluation easier, but we could use any set we wanted. The negated type $\neg A$ represents continuations, as such, it is interpreted as a function $A \to R$, where R is some arbitrary type representing the rest of the computation.

```
_ᵛᵀ : Type → Set
_ᵛˣ : Context → Set
```

$$\begin{aligned}
&\mathsf{X}^{\,vT} = \mathbb{N}\\
&(A \;\text{‘}\!\times B)^{\,vT} = (A^{\,vT}) \times (B^{\,vT})\\
&(A \;\text{‘}\!+ B)^{\,vT} = (A^{\,vT}) \uplus (B^{\,vT})\\
&(\text{‘}\neg\; A)^{\,vT} = (A^{\,vT}) \to R
\end{aligned}$$

$$\begin{aligned}
&\varnothing^{\,vx} = \top\\
&(\Gamma \;,\; A)^{\,vx} = \Gamma^{\,vx} \times A^{\,vT}
\end{aligned}$$

Now we define the interpretation of a var: a pointer to a Type in a Context. As such, the interpretation of a var projects the relevant interpreted Type from the interpreted Context.

$$\begin{aligned}
&\_^{\,vv} : \forall\; \{\Gamma\; A\} \to (\Gamma \ni A) \to ((\Gamma^{\,vx}) \to (A^{\,vT}))\\
&\_^{\,vv}\; \text{‘}\mathsf{Z}\; \langle\, \gamma \,,\, \theta \,\rangle = \theta\\
&\_^{\,vv}\; (\text{‘}\mathsf{S}\; x)\; \langle\, \gamma \,,\, \theta \,\rangle = ((x^{\,vv})\; \gamma)
\end{aligned}$$

Each DC sequent is indexed by two Contexts, therefore the interpretation of a sequent must be a function from a *pair* of interpreted Contexts. The type signatures of the interpretation of sequents are given below. $\text{‘}\neg^{x}$ is a simple implementation of the negation of a Context.

$$\begin{aligned}
&\_^{\,vLv} : \forall\; \{\Gamma\; \Theta\; A\} \to \mathsf{TermValue}\; \Gamma\; \Theta\; A \to (\Gamma^{\,vx} \times (\text{‘}\neg^{x}\; \Theta)^{\,vx}) \to (A^{\,vT})\\
&\_^{\,vL} : \forall\; \{\Gamma\; \Theta\; A\} \to (\Gamma \longrightarrow \Theta \mid A) \to (\Gamma^{\,vx} \times (\text{‘}\neg^{x}\; \Theta)^{\,vx}) \to ((A^{\,vT} \to R) \to R)\\
&\_^{\,vR} : \forall\; \{\Gamma\; \Theta\; A\} \to (A \mid \Gamma \longrightarrow \Theta) \to (\Gamma^{\,vx} \times (\text{‘}\neg^{x}\; \Theta)^{\,vx}) \to (A^{\,vT} \to R)\\
&\_^{\,vs} : \forall\; \{\Gamma\; \Theta\} \to (\Gamma \longmapsto \Theta) \to (\Gamma^{\,vx} \times (\text{‘}\neg^{x}\; \Theta)^{\,vx}) \to R
\end{aligned}$$

It is worth explaining the types of the CPS Agda programs that these transformations produce. TermValues are interpreted as programs of type $A^{\,VT}$ because they represent a completed computation that must be passed into a continuation. Left sequents contain coterms, which consume values, this means we interpret them as a function $A^{\,VT} \to R$ that consumes values of type $A^{\,VT}$. Right sequents are transformed into programs of type $(A^{\,VT} \to R) \to R$, this is the standard type of a CPS-transformed term: they take a continuation, and then return control to the rest of the computation represented by R. Statements are interpreted as programs of type R, this is because they do not actually execute anything, so they are of whatever type the rest of the program is.

The transformation of sequents is implemented in a way that closely mirrors the definition given in the original paper. ¬var is a lemma that derives $(\text{‘}\neg^{x}\; \Gamma) \ni (\text{‘}\neg\; A)$ from $\Gamma \ni A$. We must include this as the covariable α has type $\Theta \ni A$, so is interpreted as a function from $\Theta^{\,Vx}$ to $A^{\,VT}$, while we require a function from $\text{‘}\neg^{x}\; \Theta$ to $(\text{‘}\neg\; A)^{\,VT}$.

$$\begin{aligned}
&(\langle\, \text{‘}\; x \,,\, \mathsf{V\text{-}var} \,\rangle^{\,vLv})\; \langle\, \gamma \,,\, \theta \,\rangle \;= (x^{\,vv})\; \gamma\\
&(\langle\, \text{‘}\langle\, M \,,\, N \,\rangle \,,\, \mathsf{V\text{-}prod}\; V\; W \,\rangle^{\,vLv})\; c = \langle\, ((\langle\, M \,,\, V \,\rangle^{\,vLv})\; c) \,,\, (\langle\, N \,,\, W \,\rangle^{\,vLv})\; c \,\rangle\\
&(\langle\, \mathsf{inl}\langle\, M \,\rangle \,,\, \mathsf{V\text{-}inl}\; V \,\rangle^{\,vLv})\; c \;= \mathsf{inj}_1\; ((\langle\, M \,,\, V \,\rangle^{\,vLv})\; c)\\
&(\langle\, \mathsf{inr}\langle\, M \,\rangle \,,\, \mathsf{V\text{-}inr}\; V \,\rangle^{\,vLv})\; c = \mathsf{inj}_2\; ((\langle\, M \,,\, V \,\rangle^{\,vLv})\; c)\\
&(\langle\, \mathsf{not}[\; K \;] \,,\, \mathsf{V\text{-}not} \,\rangle^{\,vLv})\; c \quad = \lambda\; k \to (K^{\,vR})\; c\; k
\end{aligned}$$

$$\begin{aligned}
&((\text{‘}\; x)^{\,vL})\; \langle\, \gamma \,,\, \theta \,\rangle &&= \lambda\; k \to k\; ((x^{\,vv})\; \gamma)\\
&(\text{‘}\langle\, M \,,\, N \,\rangle^{\,vL})\; c &&= \lambda\; k \to (M^{\,vL})\; c\; (\lambda\; x \to (N^{\,vL})\; c\; (\lambda\; y \to k\; \langle\, x \,,\, y \,\rangle))\\
&(\mathsf{inl}\langle\, M \,\rangle^{\,vL})\; c &&= \lambda\; k \to (M^{\,vL})\; c\; (\lambda\; x \to k\; (\mathsf{inj}_1\; x))\\
&(\mathsf{inr}\langle\, M \,\rangle^{\,vL})\; c &&= \lambda\; k \to (M^{\,vL})\; c\; (\lambda\; x \to k\; (\mathsf{inj}_2\; x))
\end{aligned}$$

$$
\begin{aligned}
&(\mathsf{not}[\ K\ ]^{\ vL})\ c && = \lambda\ k \to k\ (\lambda\ z \to (K^{\ vR})\ c\ z)\\
&((\mu\theta\ S)^{\ vL})\ \langle\ \gamma\ ,\ \theta\ \rangle && = \lambda\ \alpha \to (S^{\ vs})\ \langle\ \gamma\ ,\ \langle\ \theta\ ,\ \alpha\ \rangle\ \rangle
\end{aligned}
$$

$$
\begin{aligned}
&((\text{‘}\ \alpha)^{\ vR})\ \langle\ \gamma\ ,\ \theta\ \rangle = \lambda\ z \to ((\neg\mathsf{var}\ \alpha)^{\ vv})\ \theta\ z\\
&(\text{‘}[\ K\ ,\ L\ ]^{\ vR})\ c && = \lambda\{\ (\mathsf{inj}_1\ x) \to (K^{\ vR})\ c\ x\ ;\ (\mathsf{inj}_2\ y) \to (L^{\ vR})\ c\ y\}\\
&(\mathsf{fst}[\ K\ ]^{\ vR})\ c && = \lambda\{\ \langle\ x\ ,\ \_\ \rangle \to (K^{\ vR})\ c\ x\}\\
&(\mathsf{snd}[\ L\ ]^{\ vR})\ c && = \lambda\{\ \langle\ \_\ ,\ y\ \rangle \to (L^{\ vR})\ c\ y\}\\
&(\mathsf{not}\langle\ M\ \rangle^{\ vR})\ c && = \lambda\ z \to (\lambda\ k \to (M^{\ vL})\ c\ k)\ z\\
&((\mu\gamma\ S)^{\ vR})\ \langle\ \gamma\ ,\ \theta\ \rangle && = \lambda\ x \to (S^{\ vs})\ \langle\ \langle\ \gamma\ ,\ x\ \rangle\ ,\ \theta\ \rangle
\end{aligned}
$$

$$
((M \bullet K)^{\ vs})\ c \quad = ((M^{\ vL})\ c)\ ((K^{\ vR})\ c)
$$

We also define a relation between the CPS translation of TermValues and right sequents based on Proposition 6.4 in the original paper. The type signature is given below, it is a simple inductive proof.

$$
\begin{aligned}
&\mathsf{cps\text{-}V} : \forall\ \{\Gamma\ \Theta\ A\}\ (V : \Gamma \longrightarrow \Theta \mid A)\ (v : \mathsf{Value}\ V)\ (c : \Gamma^{\ vx} \times (\text{‘}\neg^{x}\ \Theta)^{\ vx})\ k\\
&\quad \to (V^{\ vL})\ c\ k \equiv k\ ((\langle\ V\ ,\ v\ \rangle^{\ vLv})\ c)
\end{aligned}
$$

3.4.2 CPS Transformation of Renamings and Substitutions

It is standard for substitution to be interpreted as composition within denotational semantics. A DC substitution is a function on a sequent and two context maps, which we wish to interpret as the composition of the interpretations of the sequent and the context maps. As such, we must define the interpretation of context maps.

A context map from Γ to Γ' is a product of terms indexed by Γ', so its interpretation is a product of these terms' interpretations. This is equivalent to a map out of a product, as such the interpretation of a context map from Γ to Γ' is a function from Γ' to Γ.

The definition of the Call-by-Value interpretation of a renaming map is given below.

$$
\mathsf{ren\text{-}int\text{-}cbv} : \forall\ \Gamma\ \Gamma' \to \Gamma \rightsquigarrow \Gamma' \to (\Gamma'^{\ vx}) \to (\Gamma^{\ vx})
$$

$$
\begin{aligned}
&\mathsf{ren\text{-}int\text{-}cbv}\ \varnothing\ \Gamma'\ \rho\ \gamma = \mathsf{tt}\\
&\mathsf{ren\text{-}int\text{-}cbv}\ (\Gamma\ ,\ A)\ \Gamma'\ \rho\ \gamma =\\
&\quad \langle\ (\mathsf{ren\text{-}int\text{-}cbv}\ \Gamma\ \Gamma'\ (\lambda\ z \to \rho\ (\text{‘}\mathsf{S}\ z))\ \gamma)\ ,\ (((\rho\ \text{‘}\mathsf{Z})^{\ vv})\ \gamma)\ \rangle
\end{aligned}
$$

We also define the interpretation of a renaming map as a function between the interpretation of negated Contexts.

$$
\mathsf{neg\text{-}ren\text{-}int\text{-}cbv} : \forall\ \Theta\ \Theta' \to \Theta \rightsquigarrow \Theta' \to ((\text{‘}\neg^{x}\ \Theta')^{\ vx}) \to ((\text{‘}\neg^{x}\ \Theta)^{\ vx})
$$

The DC introduces a key difference in the interpretation of substitution maps, once again we interpret the substitution of every var that references Γ'. However, since the substitution map produces a sequent that is indexed by two Contexts, its interpretation must be applied to two interpreted Contexts. As such, the interpretation of a substitution map is a function from the interpretations of both of the Contexts that index the sequents it produces.

$$
\begin{aligned}
&\mathsf{sub\text{-}TV\text{-}int} : \forall\ \Gamma\ \Gamma'\ \Theta \to \Gamma \text{–[}\ (\mathsf{Fix}_2\ \mathsf{TermValue}\ \Theta)\ \text{]}\!\!\to \Gamma'\\
&\quad \to ((\text{‘}\neg^{x}\ \Theta)^{\ vx}) \to (\Gamma'^{\ vx}) \to (\Gamma^{\ vx})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{sub\text{-}C\text{-}int} : \forall\ \Gamma\ \Theta\ \Theta' \to \Theta \text{–[}\ (\mathsf{Fix}_1\ \mathsf{Coterm}\ \Gamma)\ \text{]}\!\!\to \Theta'\\
&\quad \to \Gamma^{\ vx} \to ((\text{‘}\neg^{x}\ \Theta')^{\ vx}) \to ((\text{‘}\neg^{x}\ \Theta)^{\ vx})
\end{aligned}
$$

3.4.3 The Renaming and Substitution Lemmas

Before we set about proving the soundness of the CPS transformation, we must prove the semantic substitution lemma, which itself requires us to prove the semantic renaming lemma. To state this explicitly, we wish to prove that the interpretation of a renamed/substituted sequent applied to a pair of interpreted Contexts is equivalent to the interpretation of the original sequent applied to a pair made up of the interpreted renaming/substitution maps applied to the interpreted Contexts.

Just like the definitions of these operations, these proofs are extremely involved and I do not wish to overwhelm the reader with detail, as such I give a high-level explanation of how the substitution lemma is proved. The renaming lemma follows the same pattern.

```
sub-lemma-T : ∀ {Γ Γ′ Θ Θ′ A}
  (s : Γ –[ (Fix₂ TermValue Θ′) ]→ Γ′) (t : Θ –[ (Fix₁ Coterm Γ′) ]→ Θ′)
  (M : Γ ⟶ Θ ∣ A) (γ : Γ′ ⱽˣ) (θ : (`¬ˣ Θ′) ⱽˣ )
  → ((sub-T TVK CK s t M) ⱽᴸ) ⟨ γ , θ ⟩
    ≡ (M ⱽᴸ) ⟨ sub-TV-int Γ Γ′ Θ′ s θ γ , sub-C-int Γ′ Θ Θ′ t γ θ ⟩

sub-lemma-C : ∀ {Γ Γ′ Θ Θ′ A}
  (s : Γ –[ (Fix₂ TermValue Θ′) ]→ Γ′) (t : Θ –[ (Fix₁ Coterm Γ′) ]→ Θ′)
  (K : A ∣ Γ ⟶ Θ) (γ : Γ′ ⱽˣ) (θ : (`¬ˣ Θ′) ⱽˣ )
  → ((sub-C TVK CK s t K) ⱽᴿ) ⟨ γ , θ ⟩
    ≡ (K ⱽᴿ) ⟨ sub-TV-int Γ Γ′ Θ′ s θ γ , sub-C-int Γ′ Θ Θ′ t γ θ ⟩

sub-lemma-S : ∀ {Γ Γ′ Θ Θ′}
  (s : Γ –[ (Fix₂ TermValue Θ′) ]→ Γ′) (t : Θ –[ (Fix₁ Coterm Γ′) ]→ Θ′)
  (S : Γ ↦ Θ) (γ : Γ′ ⱽˣ) (θ : (`¬ˣ Θ′) ⱽˣ )
  → ((sub-S TVK CK s t S) ⱽˢ) ⟨ γ , θ ⟩
    ≡ (S ⱽˢ) ⟨ sub-TV-int Γ Γ′ Θ′ s θ γ , sub-C-int Γ′ Θ Θ′ t γ θ ⟩
```

We prove this by induction on the sequent that is being substituted into, the interesting cases are for (co)variables and (co)variable abstraction.

The (co)variable cases require us to prove a similar substitution lemma for the interpretation of a substitution map applied to a var.

The (co)variable abstraction cases are interesting as substitution into (co)variable abstractions involves sub-weaken, sub-lift, and fmap. This means that we must prove lemmas that define the interpretation of *weakened* and *fmap-ed* substitutions [3] in terms that do not include sub-weaken, sub-lift, or fmap. This must be done for both TermValue and left sequent substitutions. Since the behaviour of these functions is defined by renaming, these lemmas require us to prove the renaming lemma. As an example, I give the type signature of the weakening lemma for TermValues below.

```
weaken-sub-TV-int-lemma : ∀ {Γ Γ′ Θ A} (σ : Γ –[ (Fix₂ TermValue Θ) ]→ Γ′) γ θ k
  → sub-TV-int Γ (Γ′ , A) Θ (sub-weaken (TermKit.kit TVK) σ) θ ⟨ γ , k ⟩
    ≡ sub-TV-int Γ Γ′ Θ σ θ γ
```

[3] We do not need to prove anything for sub-lift as the interpretation of a *lifted* substitution can be defined in terms of a *weakened* substitution.

3.4.4 Proof of Soundness

In the original paper, Wadler asserts that the CPS transformations preserve reductions from DC to the target calculus. I have taken a slightly different approach to defining the CPS transformation, as such, in our context, the natural formulation of the property of soundness is that the CPS transformations of DC programs related by many-step reduction are propositionally equal Agda terms.

```
S—↠ᵛT⇒Sᵛ≡Tᵛ : ∀ {Γ Θ} (S T : Γ ⟼ Θ) (c : (Γ ᵒˣ) ⁿˣ × `¬ˣ (Θ ᵒˣ) ⁿˣ)
  → S ˢ—↠ᵛ T → (S ᵛˢ) c ≡ (T ᵛˢ) c
```

We first prove that single-step reduction preserves the meaning of statements.

```
S⟶ᵛT⇒Sᵛ≡Tᵛ : ∀ {Γ Θ} (S T : Γ ⟼ Θ) (c : (Γ ᵒˣ) ⁿˣ × `¬ˣ (Θ ᵒˣ) ⁿˣ)
  → S ˢ⟶ᵛ T → (S ᵛˢ) c ≡ (T ᵛˢ) c
```

This lemma is proved by induction on the reduction relation $\_\,{}^{s}\!\!\longrightarrow^{V}\_$. The cases for the congruence rules are either definitional equalities or can be proved by appealing to the cps-V theorem. The (βL) and (βR) cases are more interesting as they involve substitution, I will present the TermValue substitution case below in equational reasoning form, and then explain it. Note the use of sym at the start of the proof, this reverses the direction of the equality to be proved.

```
S⟶ᵛT⇒Sᵛ≡Tᵛ {Γ} {Θ} (V ● μγ {Γ}{Θ}{A} S) .(S ᵛ⟨ ⟨ V , v ⟩ /⟩ˢ) ⟨ γ , θ ⟩ (βL v) = sym (
  begin
    ((S ᵛ⟨ ⟨ V , v ⟩ /⟩ˢ) ᵛˢ) ⟨ γ , θ ⟩
  ≡⟨⟩
    (sub-S TVK CK (add (Fix₂ TermValue Θ) ⟨ V , v ⟩ id-TV) id-C S ᵛˢ) ⟨ γ , θ ⟩
  ≡⟨ sub-lemma-S (add (Fix₂ TermValue Θ) ⟨ V , v ⟩ id-TV) id-C S γ θ ⟩
    (S ᵛˢ)
      ⟨ sub-TV-int (Γ , A) Γ Θ (add (Fix₂ TermValue Θ) ⟨ V , v ⟩ id-TV) θ γ
      , sub-C-int Γ Θ Θ id-C γ θ ⟩
  ≡⟨⟩
    (S ᵛˢ)
      ⟨ ⟨ sub-TV-int Γ Γ Θ id-TV θ γ , (⟨ V , v ⟩ ᵛᴸᵛ) ⟨ γ , θ ⟩ ⟩
      , sub-C-int Γ Θ Θ id-C γ θ ⟩
  ≡⟨ cong₂ (λ -₁ -₂ → (S ᵛˢ) ⟨ ⟨ -₁ , (⟨ V , v ⟩ ᵛᴸᵛ) ⟨ γ , θ ⟩ ⟩ , -₂ ⟩)
     (id-sub-TV Γ Θ γ θ) (id-sub-C Γ Θ γ θ) ⟩
    (S ᵛˢ) ⟨ ⟨ γ , (⟨ V , v ⟩ ᵛᴸᵛ) ⟨ γ , θ ⟩ ⟩ , θ ⟩
  ≡˘⟨ cong (λ - → - (λ x → (S ᵛˢ) ⟨ ⟨ γ , x ⟩ , θ ⟩)) (cps-V V v ⟨ γ , θ ⟩) ⟩
    (V ᵛᴸ) ⟨ γ , θ ⟩ (λ x → (S ᵛˢ) ⟨ ⟨ γ , x ⟩ , θ ⟩)
  ∎)
```

The first step of the proof is a definitional equality from the definition of $\_\,{}^{V}\langle\_/\rangle^{s}$, reducing the statement to the sub-statement form.

The second step appeals to the semantic substitution lemma.

The third step is another definitional equality, in this case derived from the inductive case of termvalue-sub-int. We can exploit this due to the guarantee that the first Context we pass to the interpretation function is non-empty.

The next step of the proofs uses two lemmas, they state that the interpretation of identity substitutions is the identity function.

The final step is to appeal to the cps-V theorem.

The proof of the right sequent substitution case is a dual of the proof given above, the only major difference being that we do not have to appeal to cps-V, so we omit it.

Multi-step reduction is simply the reflexive-transitive closure of single-step reduction, so its soundness can now be proved by repeated applications of the single-step soundness property given above.

```
S—↠ⱽT⇒Sⱽ≡Tⱽ S .S c (.S ∎ˢⱽ) = refl
S—↠ⱽT⇒Sⱽ≡Tⱽ S T c ( _ˢ⟶ⱽ⟨_⟩_ .S {S′} .{T} S⟶S′ S′↠T) =
  trans (S⟶ⱽT⇒Sⱽ≡Tⱽ S S′ c S⟶S′) (S—↠ⱽT⇒Sⱽ≡Tⱽ S′ T c S′↠T)
```

This completes our proof of the soundness of the Call-by-Value CPS transformation. The Call-by-Name CPS transformation requires a separate proof with its own renaming and substitution lemmas. This proof, however, is the dual of the proof I have outlined so is not included.

3.5 Duality

Duality is baked into the Dual Caluclus from the start; we explore it formally in the following section. We first define the dual translation, followed by the duality of the operational and denotational semantics.

3.5.1 The Duality of Syntax

Any construct of the DC that does not involve implication has a dual. The dual translation defined in the original paper is five different operations: a separate translation for Types, Terms, Coterms, Statements, and Contexts. It also assumes the existence of a bijective dual translation between variables and covariables, which we must define.

First, we define the dual translation of Type and Context with two mutually recursive functions. As expected, products and sums are duals of each other, and negation (and the base type) are self-dual.

```
_ᵒᵀ : Type → Type
_ᵒˣ : Context → Context

(A `+ B)ᵒᵀ = (A ᵒᵀ `× B ᵒᵀ)
(A `× B)ᵒᵀ = (A ᵒᵀ `+ B ᵒᵀ)
(`¬ A)ᵒᵀ   = (`¬ (A)ᵒᵀ)
(`ℕ)ᵒᵀ     = `ℕ

(∅ ᵒˣ) = ∅
(Γ , A) ᵒˣ = ((Γ ᵒˣ) , (A ᵒᵀ))
```

Next, we define the bijection between variables and covariables. The dual translation of a sequent makes the antecedent of the original the succedent of its dual, and vice versa. As such, variables are mapped to covariables by virtue of the Contexts they are in being swapped in the dual translation of a sequent. Therefore, the only condition of this bijection is to take the dual of the Context and Type that the var is indexed by.

```
_ᵒᵛ : ∀ {Γ A} → (Γ ∋ A) → (Γ ᵒˣ ∋ A ᵒᵀ)
```

Now we can define the dual translation of sequents, with separate functions for left, right, and centre sequents. Left sequents and right sequents are duals of each other, while centre sequents are self-dual.

$\_^{\,os} : \forall\ \{\Gamma\ \Theta\} \to (\Gamma \longmapsto \Theta) \to (\Theta^{\,ox} \longmapsto \Gamma^{\,ox})$
$\_^{\,oL} : \forall\ \{\Gamma\ \Theta\ A\} \to (\Gamma \longrightarrow \Theta \mid A) \to (A^{\,oT} \mid \Theta^{\,ox} \longrightarrow \Gamma^{\,ox})$
$\_^{\,oR} : \forall\ \{\Gamma\ \Theta\ A\} \to (A \mid \Gamma \longrightarrow \Theta) \to (\Theta^{\,ox} \longrightarrow \Gamma^{\,ox} \mid A^{\,oT})$

$(`\ x)^{oL} \qquad = `\ x^{\,ov}$
$(`\langle\ M\ ,\ N\ \rangle)^{\,oL} = `[\ M^{\,oL}\ ,\ N^{\,oL}\ ]$
$(\mathsf{inl}\langle\ M\ \rangle)^{\,oL} \ = \mathsf{fst}[\ M^{\,oL}\ ]$
$(\mathsf{inr}\langle\ M\ \rangle)^{\,oL} \ = \mathsf{snd}[\ M^{\,oL}\ ]$
$(\mathsf{not}[\ K\ ])^{\,oL} = \mathsf{not}\langle\ K^{\,oR}\ \rangle$
$(\mu\theta\ \{\Gamma\}\ \{\Theta\}\ \{A\}\ (S))^{\,oL} = \mu\gamma(\ \_^{\,os}\ \{\Gamma\}\ \{(\Theta\ ,\ A)\}\ S\ )$

$(`\ \alpha)^{\,oR} \qquad = `\ \alpha^{\,ov}$
$(`[\ K\ ,\ L\ ])^{\,oR} = `\langle\ K^{\,oR}\ ,\ L^{\,oR}\ \rangle$
$(\mathsf{fst}[\ K\ ])^{\,oR} \ = \mathsf{inl}\langle\ K^{\,oR}\ \rangle$
$(\mathsf{snd}[\ K\ ])^{\,oR} = \mathsf{inr}\langle\ K^{\,oR}\ \rangle$
$(\mathsf{not}\langle\ M\ \rangle)^{\,oR} = \mathsf{not}[\ M^{\,oL}\ ]$
$(\mu\gamma\ \{\Gamma\}\ \{\Theta\}\ \{A\}\ (S))^{\,oR} = \mu\theta(\ \_^{\,os}\ \{(\Gamma\ ,\ A)\}\ \{\Theta\}\ (S)\ )$

$(M \bullet K)^{\,os} = K^{\,oR} \bullet M^{\,oL}$

Despite this not being defined in the original paper, the proof the duality of the operational semantics requires a definition of the dual translation of renamings and substitutions. These definitions are simple induction, though I give their types below.

$\text{dual-ren} : \forall\ \Gamma\ \Gamma' \to \Gamma \rightsquigarrow \Gamma' \to (\Gamma^{\,ox}) \rightsquigarrow (\Gamma'^{\,ox})$

$\text{dual-sub-C} : \forall\ \Gamma\ \Theta\ \Theta' \to \Theta\ \text{–[}(\mathsf{Fix}_1\ \mathsf{Coterm}\ \Gamma)\text{]→}\ \Theta' \to (\Theta^{\,ox})\ \text{–[}\ (\mathsf{Fix}_2\ \mathsf{Term}\ (\Gamma^{\,ox}))\ \text{]→}\ (\Theta'^{\,ox})$

$\text{dual-sub-TV} : \forall\ \Gamma\ \Gamma'\ \Theta \to \Gamma\ \text{–[}(\mathsf{Fix}_2\ \mathsf{TermValue}\ \Theta)\text{]→}\ \Gamma' \to (\Gamma^{\,ox})\ \text{–[}(\mathsf{Fix}_1\ \mathsf{CotermValue}\ (\Theta^{\,ox}))\text{]→}\ (\Gamma'^{\,ox})$

Wadler asserts that a sequent is derivable iff its dual is derivable. Since our intrinsically-typed representation of syntax guarantees that it is impossible to produce an ill-typed sequent, producing a sequent in our formalisation is a proof that it is derivable. The dual translation produces the dual of any given sequent, thus demonstrating that if a sequent is derivable then so is its dual. To prove the reverse we must prove that the dual translation is an involution.

The mathematical proof that the dual translation is an involution is simple, though there is some difficulty in translating it to a formal proof.

First, we prove the theorem for Types and Contexts, these are simple inductive proofs that we omit for brevity.

$[\mathsf{A}^{oT}]^{oT}{\equiv}\mathsf{A} : \forall\ \{A\} \to (A^{\,oT})^{\,oT} \equiv A$

$[\Gamma^{ox}]^{ox}{\equiv}\Gamma : \forall\ \{\Gamma\} \to (\Gamma^{\,ox})^{\,ox} \equiv \Gamma$

I now present the involution proof for vars and will discuss the issue with translating the mathematical proof into a formal proof.

$\mathsf{[x^{ov}]^{ov}{\equiv}x} : \forall\ \{\Gamma\ A\}\ (x : \Gamma \ni A) \rightarrow ((x\ ^{ov})\ ^{ov}) \equiv x$
$\mathsf{[x^{ov}]^{ov}{\equiv}x}\ (\mathsf{`Z}) = \mathsf{refl}$
$\mathsf{[x^{ov}]^{ov}{\equiv}x}\ (\mathsf{`S}\ x) = \mathsf{cong}\ \mathsf{`S}\ (\mathsf{[x^{ov}]^{ov}{\equiv}x}\ x)$

This proof looks exactly like we would expect, however, Agda's type checker reports the following error in the type signature of the proof.

> Γ != (Γ ^{ox}) ^{ox} of type Context when checking that expression x has type (Γ ^{ox}) ^{ox} ∋ (A ^{oT}) ^{oT}.

The issue here is that we are trying to prove the equality of two terms, $(x\ ^{oV})\ ^{oV}$ and x, that have different types: $(\Gamma\ ^{ox})\ ^{ox} \ni (A\ ^{oT})\ ^{oT}$ and $\Gamma \ni A$. Despite the fact that we have proved that the constructs indexing these two types are equal, Agda will not accept this. In principle, it is possible to use Agda's heterogeneous equality, a notion of equality between terms of different types, however, the proofs are unnecessarily cumbersome. Instead, we make use of a REWRITE pragma, this marks a given propositional equality as a *rewrite rule*, meaning Agda will "automatically rewrite all instances of the left-hand side to the corresponding instance of the right-hand side during reduction" [44]. We use a REWRITE pragma to mark the involution equalities for Types and Contexts as rewrite rules. While using REWRITE is unsafe in general, in this specific case we have only used them with proven propositions, so there is nothing logically sinful about this.

$\{\text{-\#}\ \mathsf{REWRITE}\ \mathsf{[A^{oT}]^{oT}{\equiv}A}\ \text{\#-}\}$
$\{\text{-\#}\ \mathsf{REWRITE}\ \mathsf{[\Gamma^{ox}]^{ox}{\equiv}\Gamma}\ \text{\#-}\}$

With these rewrite rules, we can prove that the dual translation of sequents is also an involution. These are simple mutually inductive proofs so I only include the type signatures.

$\mathsf{[K^{oR}]^{oL}{\equiv}K} : \forall\ \{\Gamma\ \Theta\ A\}\ (K : A \mid \Gamma \longrightarrow \Theta) \rightarrow (K\ ^{oR})\ ^{oL} \equiv K$
$\mathsf{[M^{oL}]^{oR}{\equiv}M} : \forall\ \{\Gamma\ \Theta\ A\}\ (M : \Gamma \longrightarrow \Theta \mid A) \rightarrow (M\ ^{oL})\ ^{oR} \equiv M$
$\mathsf{[S^{os}]^{os}{\equiv}S} : \forall\ \{\Gamma\ \Theta\}\ (S : \Gamma \longmapsto \Theta) \rightarrow (S\ ^{os})\ ^{os} \equiv S$

3.5.2 The Duality of Operational Semantics

I now prove the original paper's titular claim, that Call-by-Value is dual to Call-by-Name, by demonstrating the duality in the DC's operational semantics. Specifically, we prove that a sequent S reduces by Call-by-Value to another sequent T iff the dual of S reduces by Call-by-Name to the dual of T. For brevity I demonstrate only one direction of this proof, the other direction is dual.

$\mathsf{S{\longrightarrow}^{v}T{\Rightarrow}S^{o}{\longrightarrow}^{n}T^{o}} : \forall\ \{\Gamma\ \Theta\}\ (S\ T : \Gamma \longmapsto \Theta) \rightarrow S\ ^{s}{\longrightarrow}^{v}\ T \rightarrow (S\ ^{os})\ ^{s}{\longrightarrow}^{n}\ (T\ ^{os})$

This proof is remarkably similar in structure to the proof of the soundness of the denotational semantics, as both proofs seek to prove that the reduction relation preserves some property. In the case of soundness, we demonstrated that it preserves meaning, here we seek to prove that it preserves duality. Again, much of the proof consists of proving a variety of lemmas about the behaviour of the duals of lifted, weakened, fmap-ed etc. substitutions and renamings.

Most of the cases of the duality of the operational semantics are relatively straightforward; requiring only that we prove that the dual of a value is a covalue. The type signature of this proof is given below.

$\mathsf{V^{o}{\equiv}P} : \forall\ \{\Gamma\ \Theta\ A\}\ (V : \Gamma \longrightarrow \Theta \mid A) \rightarrow \mathsf{Value}\ V \rightarrow (\mathsf{Covalue}\ (V\ ^{oL}))$

However, the cases for the βL and βR reduction rules require us to prove a new set of renaming and substitution lemmas. Stating that the dual of a renamed/substituted sequent is equivalent to renaming/ substituting the the dual of the original sequent with the duals of the original context maps. I outline the proof of the substitution lemma; it relies on the proof of the renaming lemma that I do not detail. Its type signature of the substitution lemma is given below.

```
dual-sub-lemma-T : ∀ {Γ Γ' Θ Θ' A} (M : Γ ⟶ Θ ∣ A)
  (s : Γ –[ (Fix_2 TermValue Θ') ]→ Γ') (t : Θ –[ (Fix_1 Coterm Γ') ]→ Θ')
  → (sub-T TVK CK s t M) ^{oL} ≡ sub-C TK CVK (dual-sub-C Γ' Θ Θ' t) (dual-sub-TV Γ Γ' Θ' s) (M ^{oL})
dual-sub-lemma-C : ∀ {Γ Γ' Θ Θ' A} (K : A ∣ Γ ⟶ Θ)
  (s : Γ –[ (Fix_2 TermValue Θ') ]→ Γ') (t : Θ –[ (Fix_1 Coterm Γ') ]→ Θ')
  → (sub-C TVK CK s t K) ^{oR} ≡ sub-T TK CVK (dual-sub-C Γ' Θ Θ' t) (dual-sub-TV Γ Γ' Θ' s) (K ^{oR})
dual-sub-lemma-S : ∀ {Γ Γ' Θ Θ'} (S : Γ ⟼ Θ)
  (s : Γ –[ (Fix_2 TermValue Θ') ]→ Γ') (t : Θ –[ (Fix_1 Coterm Γ') ]→ Θ')
  → (sub-S TVK CK s t S) ^{os} ≡ sub-S TK CVK (dual-sub-C Γ' Θ Θ' t) (dual-sub-TV Γ Γ' Θ' s) (S ^{os})
```

Again we prove a corresponding lemma for the (co)variable cases, as well as lemmas for each of the different substitution combinators for the (co)variable abstraction cases. I give the types of some of these lemmas for TermValues below.

```
dual-sub-TV-lift-lemma : ∀ Γ Γ' Θ' A {B} (σ : Γ –[ (Fix_2 TermValue Θ') ]→ Γ') (x : (Γ , A) ^{ox} ∋ B)
  → dual-sub-TV (Γ , A) (Γ' , A) Θ' (sub-lift (TVK.kit) σ) x
    ≡ sub-lift (CVK.kit) (dual-sub-TV Γ Γ' Θ' σ) x


dual-sub-TV-id-lemma : ∀ Γ Θ A (x : Γ ^{ox} ∋ A)
  → dual-sub-TV Γ Γ Θ id-TV x ≡ id-CV x
```

The interesting parts of the proofs of these lemmas is proving that the sequent that results from the substitution is a Covalue. We can do this by demonstrating that to prove that two CotermValues are equal, it is sufficient to show that the underlying coterms are equal.

```
CTV-eq : ∀{Γ Θ A}{K L : Coterm Γ Θ A}(K-V : Covalue K)(L-V : Covalue L)
  → (e : K ≡ L) → ⟨ K , K-V ⟩ ≡ ⟨ L , L-V ⟩
CTV-eq K-V L-V e = Inverse.f Σ-≡,≡↔≡ ⟨ e , CV-eq K-V L-V e ⟩
```

To prove this we make use of the Agda standard library's Σ−≡, ≡↔≡, this allows us to express the equality of a dependent pair as a dependent pair of equalities. The first equality in the pair is our assumption that the coterms are equal. The second appeals to a lemma stating that if two left sequents are equal then the proofs that they are covalues are equal. This is proved by simple induction.

```
CV-eq : ∀ {Γ Θ A} {K L : Coterm Γ Θ A} (K-V : Covalue K) (L-V : Covalue L)
  → (e : K ≡ L) → subst Covalue e K-V ≡ L-V
```

We make use of subst in the type of CV-eq because L-V is of type Covalue L so the left-hand side of the equation must be too. Using subst takes K-V, of type Covalue K, and e, a proof that K and L are equal, returning a value of type Covalue L.

With CTV-eq proved, proving the rest of the lemmas required for the substitution lemma is simple enough. Proving the abstraction cases of the substitution lemma, and the duality of the operational semantics simply involves appealing to these lemmas, hence they are omitted.

3.5.3 The Duality of Denotational Semantics

I now outline the proof of another instance of the original paper's titular claim, in this case for the denotational semantics of the DC.

We start with proving the statement for Types and Contexts by induction.

$$\mathsf{A}^{v}\!\equiv\!\mathsf{A}^{on} : \forall\ \{A\} \to A^{\ vT} \equiv (A^{\ oT})^{\ nT}$$

$$\Gamma^{v}\!\equiv\!\Gamma^{on} : \forall\ \{\Gamma\} \to \Gamma^{\ vx} \equiv (\Gamma^{\ ox})^{\ nx}$$

We then add rewrite rules for these two equalities, for the same reason we did in the proof that the dual translation is involutive.

We now prove the duality of the transformations for vars.

$$\begin{aligned}
&\mathsf{x}^{v}\!\equiv\!\mathsf{x}^{on} : \forall\ \{\Gamma\ A\}\ (x : \Gamma \ni A)\ (c : \Gamma^{\ vx}) \to (x^{\ vv})\ c \equiv ((x^{\ ov})^{\ nv})\ c\\
&\mathsf{x}^{v}\!\equiv\!\mathsf{x}^{on}\ \text{`}\mathsf{Z}\ c = \mathsf{refl}\\
&\mathsf{x}^{v}\!\equiv\!\mathsf{x}^{on}\ (\text{`}\mathsf{S}\ x)\ c = \mathsf{x}^{v}\!\equiv\!\mathsf{x}^{on}\ x\ (\mathsf{proj}_1\ c)
\end{aligned}$$

Then follows the proof for sequents.

$$\begin{aligned}
&\mathsf{M}^{v}\!\equiv\!\mathsf{M}^{on} : \forall\ \{\Gamma\ \Theta\ A\}\ (M : \Gamma \longrightarrow \Theta \mid A)\ (c : \Gamma^{\ vx} \times (\text{`}\neg^{x}\ \Theta)^{\ vx})\ (k : ((\text{`}\neg\ A)^{\ vT}))\\
&\quad \to (M^{\ vL})\ c\ k \equiv ((M^{\ oL})^{\ nR})\ c\ k\\
&\mathsf{K}^{v}\!\equiv\!\mathsf{K}^{on} : \forall\ \{\Gamma\ \Theta\ A\}\ (K : A \mid \Gamma \longrightarrow \Theta)\ (c : \Gamma^{\ vx} \times (\text{`}\neg^{x}\ \Theta)^{\ vx})\ (k : (A)^{\ vT})\\
&\quad \to (K^{\ vR})\ c\ k \equiv ((K^{\ oR})^{\ nL})\ c\ k\\
&\mathsf{S}^{v}\!\equiv\!\mathsf{S}^{on} : \forall\ \{\Gamma\ \Theta\}\ (S : \Gamma \longmapsto \Theta)\ (c : \Gamma^{\ vx} \times (\text{`}\neg^{x}\ \Theta)^{\ vx})\\
&\quad \to (S^{\ vs})\ c \equiv ((S^{\ os})^{\ ns})\ c
\end{aligned}$$

The (co)variable cases of this proof simply appeal to the duality of the var transformation, as well as a simple lemma in the case of covariables. We can use clever pattern matching to make all the other cases simple induction. A slight exception to this are the (co)variable abstraction cases, which require us to extend the interpreted Context with the newly bound (co)variable.

$$\begin{aligned}
&\mathsf{K}^{v}\!\equiv\!\mathsf{K}^{on}\ (\text{`}\ \alpha)\ \langle\ \_\ ,\ c\ \rangle\ k = \mathsf{trans}\\
&\qquad\qquad\qquad\qquad (\mathsf{cong}\ (\lambda\ \text{-} \to \text{-}\ k)\ (\mathsf{x}^{v}\!\equiv\!\mathsf{x}^{on}\ (\neg\mathsf{var}\ \alpha)\ c))\\
&\qquad\qquad\qquad\qquad (\mathsf{cong}\ (\lambda\ \text{-} \to (\text{-}^{\ nv})\ c\ k)\ ([\neg\mathsf{varx}]^{o}\!\equiv\!\neg\mathsf{var}[\mathsf{x}^{o}]\ \alpha))
\end{aligned}$$

$$\mathsf{M}^{v}\!\equiv\!\mathsf{M}^{on}\ (\mu\theta\ S)\ \langle\ c_1\ ,\ c_2\ \rangle\ k = \mathsf{S}^{v}\!\equiv\!\mathsf{S}^{on}\ S\ \langle\ c_1\ ,\ \langle\ c_2\ ,\ k\ \rangle\ \rangle$$

$$\mathsf{K}^{v}\!\equiv\!\mathsf{K}^{on}\ (\mu\gamma\ S)\ \langle\ c_1\ ,\ c_2\ \rangle\ k = \mathsf{S}^{v}\!\equiv\!\mathsf{S}^{on}\ S\ \langle\ \langle\ c_1\ ,\ k\ \rangle\ ,\ c_2\ \rangle$$

In the preceding chapter we have defined the syntax, operational semantics, and denotational semantics of the Dual Calculus. We proved the soundness of the denotational semantics over the operational semantics, forcing us to engage with the complexity of the formalisation of substitution. We then revisited each of the three parts in turn: formally defining the duality of DC constructs, and proving that the Call-by-Value semantics, both operational and denotational, are dual to Call-by-Name semantics.

Chapter 4

Evaluation

4.1 Work Completed

The project was a success. As well as completing the core requirements, I had to time to investigate several planned and unplanned extensions.

4.1.1 Core

I outlined six key pieces of work to be completed as part of the project core.

The first of these was to learn Agda to the standard necessary to fulfil my other objectives, by reading and working through the exercises in PLFA [48]. This was a great introduction to the fundamentals, though the finer details that were required as the implementation grew in complexity were learnt from various sources, including the Agda documentation [44] and my supervisor.

The next three pieces of work I outlined were to formalise the syntax of the DC, as well as the dual translation and the CPS transformations. While the learning curve was steep, I was able to complete these objectives in good time. After this, I completed the proof that the CPS transformations were dual.

Due to some limitations of the Dual Calculus (outlined in Section 4.3), we deemed a performance-based evaluation to be less informative than exploring the system's utility to model other computational calculi.

4.1.2 Extensions

My project proposal outlined three possible extensions: I completed two of these, taking one further than originally planned.

The first extension I completed was to define the operational semantics of the DC, requiring the implementation of renaming and substitution. I took this extension further by proving the duality of these operational semantics, as I believed this to be an important conclusion of the original paper.

This then allowed me to complete the second extension I had planned, proving the soundness of the CPS transformations over the operational semantics.

4.2 Unit Tests for the Dual Calculus Formalisation

The proofs of the various properties of the DC validate the formalisation is correct, as such, unit tests for the completed formalisation would be superfluous. Despite this, to make sure that my formalisation was behaving

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{\dfrac{}{x : A \to \varnothing \mid x : A}\,\textsc{IdR}}{x : A \to \varnothing \mid \langle x\rangle\mathrm{inl} : A \vee \neg A}\,\vee\mathrm{R1}
            \quad
            \dfrac{}{\gamma : A \vee \neg A \mid \varnothing \to \gamma : A \vee \neg A}\,\textsc{IdL}
          }{x : A \mid \langle x\rangle\mathrm{inl} \bullet \gamma \mapsto \gamma : A \vee \neg A}\,\textsc{Cut}
        }{x.(\langle x\rangle\mathrm{inl} \bullet \gamma) : A \mid \varnothing \to \gamma : A \vee \neg A}\,\mathrm{LI}
      }{\varnothing \to \gamma : A \vee \neg A \mid [x.(\langle x\rangle\mathrm{inl} \bullet \gamma)]\mathrm{not} : \neg A}\,\neg\mathrm{R}
    }{\varnothing \to \gamma : A \vee \neg A \mid \langle [x.(\langle x\rangle\mathrm{inl} \bullet \gamma)]\mathrm{not}\rangle\mathrm{inr} : A \vee \neg A}\,\vee\mathrm{R1}
    \quad
    \dfrac{}{\gamma : A \vee \neg A \mid \varnothing \to \gamma : A \vee \neg A}\,\textsc{IdL}
  }{\varnothing \mid \langle [x.(\langle x\rangle\mathrm{inl} \bullet \gamma)]\mathrm{not}\rangle\mathrm{inr} \bullet \gamma \mapsto \gamma : A \vee \neg A}\,\textsc{Cut}
}{\varnothing \to \varnothing \mid (\langle [x.(\langle x\rangle\mathrm{inl} \bullet \gamma)]\mathrm{not}\rangle\mathrm{inr} \bullet \gamma).\gamma : A \vee \neg A}\,\mathrm{RI}
$$

```
lem-proof : ∀ {A} → ∅ ⟶ ∅ ∣ A `+ `¬ A
lem-proof = μθ (inr⟨ not[ μγ (inl⟨ γ 0 ⟩ ● (θ 0) ) ] ⟩ ● (θ 0))
```

Figure 4.2.1: Derivation tree for the DC proof of LEM, alongside its encoding in my formalisation. Note that I remove some unnecessary variables from typing environments in the derivation tree, this is purely to save space.

correctly while it was in development, I used the in-progress formalisation to prove simple lemmas as well as producing results given in the original paper.

Upon completing the formalisation of the syntax, I produced a proof of *the law of the excluded middle* – a characteristic property of classical logic. Wadler gives a derivation of the DC term that proves this law in the original paper (see in Figure 4.2.1); converting this term into the syntax of my formalisation and type-checking it gave me confidence that I had defined the syntax correctly.

The dual translation has two important properties that my formalisation had to validate: that a sequent is derivable iff its dual is derivable and that the duality is an involution. The dual translation I produced is separated into multiple different functions, with the dual translation of sequents relying on the dual translation of Types and Contexts. I wanted to be sure that the dual translation functions I was relying on were correct before I used them, therefore I decided to prove that each dual translation was an involution before defining the next one.

The original paper has some examples of the CPS transformations of DC terms. To validate that the CPS transformations I had defined were correct, I replicated the examples given in the paper within my formalisation. To do this I produced a DC sequent equivalent to the one given in the paper, applied my CPS transformation to it, and then compared the normalised Agda program this produced to the CPS λ-terms in the original paper. I found that my CPS transformation produced equivalent terms for four of the five examples, one of these is given below.

$$
\begin{aligned}
&((z \bullet \mathrm{fst}[\alpha]).\alpha)^v \\
\equiv\ & \lambda\alpha.(\lambda\gamma.\gamma\, z)(\lambda z'.\mathtt{case}\ z'\ \mathtt{of}\ \langle x, -\rangle \\
&\quad \to (\lambda z''.\alpha\, z'')x) \\
\equiv\ & \lambda\alpha.\mathtt{case}\ z\ \mathtt{of}\ \langle x, -\rangle \Rightarrow \alpha\, x
\end{aligned}
$$

```
ex2 : ∀ {A B} → ∅ , (A `× B) ⟶ ∅ ∣ A
ex2 = μθ (γ 0 ● fst[ θ 0 ])

ex2ᵛ : ∀ {A B} z → ((A ᵛᵀ → R) → R)
ex2ᵛ {A}{B} z = (ex2 {A}{B} ᵛᴸ) ⟨ ⟨ tt , z ⟩ , tt ⟩

_ : ∀ {A B} z → (ex2ᵛ {A}{B} z) ≡ λ α → α (proj₁ z)
_ = λ z → refl
```

The example that did not give an equivalent term was initially quite worrying. I worked through the example by hand, using Wadler's CPS transformation and the DC term that he provided, and realised he had made a minor error in applying the CPS transformation. This meant that my CPS transformation did in fact produce the correct term. This is a helpful example of the usefulness of language formalisation. Though the error is minor and appeared only in a simple example, it is hard for a human to spot and it likely would have gone unnoticed if not for my formalisation. The example, as it appears in the original paper, as well as a corrected version, is given below.

Original

$$\begin{aligned}
&([\alpha]\mathrm{not} \bullet \mathrm{not}\langle x\rangle)^v \\
\equiv\, &(\lambda\gamma.\gamma\,(\lambda z.(\lambda z.\alpha\, z) z))(\lambda z.(\lambda\gamma.(\lambda\gamma.x\,\gamma)\gamma) z) \\
\equiv\, &(\lambda\gamma.x\,\gamma)(\lambda z.\alpha\, z) \\
\equiv\, &x\,\alpha
\end{aligned}$$

Corrected

$$\begin{aligned}
&([\alpha]\mathrm{not} \bullet \mathrm{not}\langle x\rangle)^v \\
\equiv\, &(\lambda\gamma.\gamma\,(\lambda z.(\lambda z.\alpha\, z) z))(\lambda z.(\lambda\gamma.(\lambda\gamma.\gamma\, x)\gamma) z) \\
\equiv\, &(\lambda\gamma.(\lambda\gamma.\gamma\, x)\gamma)(\lambda z.(\lambda z.\alpha\, z) z) \\
\equiv\, &\alpha\, x
\end{aligned}$$

To test the operational semantics I showed that a proof of the law of the excluded middle (LEM), $A \vee \neg A$, contradicted with a refutation of LEM would reduce to a contradiction between the proof of A and the refutation of A. This proof is given below.

```
lem-comp : ∀ {A} → (M : ∅ ⟶ ∅ ∣ A) → Value M → (K : A ∣ ∅ ⟶ ∅)
  → (lem-proof • lem-ref M K) ˢ—↠ᵛ M • K
lem-comp M M:V K = beginˢᵛ
    μθ (inr⟨ not[ μγ (inl⟨ γ 0 ⟩ • (θ 0) ) ] ⟩ • (θ 0))
    • ‘[ K , not⟨ M ⟩ ]
  ˢ⟶ᵛ⟨ βR ⟩
    inr⟨ not[ μγ (inl⟨ γ 0 ⟩ • ‘[ wkΓᶜ K , not⟨ wkΓᵗ M ⟩ ] ) ] ⟩
    • ‘[ K , not⟨ M ⟩ ]
  ˢ⟶ᵛ⟨ β+₂ V-not ⟩
    not[ μγ (inl⟨ γ 0 ⟩ • ‘[ wkΓᶜ K , not⟨ wkΓᵗ M ⟩ ] ) ]
    • not⟨ M ⟩
  ˢ⟶ᵛ⟨ β¬ ⟩
    M
    • μγ (inl⟨ γ 0 ⟩ • ‘[ wkΓᶜ K , not⟨ wkΓᵗ M ⟩ ] )
  ˢ⟶ᵛ⟨ βL M:V ⟩
    ((inl⟨ γ 0 ⟩ • ‘[ (wkΓᶜ K) , not⟨ (wkΓᵗ M) ⟩ ]) ᵛ⟨ ⟨ M , M:V ⟩ /⟩ˢ)
  ˢ⟶ᵛ⟨ {! !} ⟩
    inl⟨ M ⟩
    • ‘[ K , not⟨ M ⟩ ]
  ˢ⟶ᵛ⟨ β+₁ M:V ⟩
    M • K
  □ˢᵛ
```

‘{! !}’, used in the second last step constructor, is Agda syntax for a ‘hole’: something you have not proved. I include this to denote that the above proof relies on the unproven property that a renaming (a weakening is just an instance of renaming) followed by a substitution is equivalent to a substitution made by composing the context map of the original substitution with the renaming map of the renaming. This does not mean that my operational semantics is incorrect; just that a simplification is possible – a simplification that I include above without proof. The type signature of the lemma required is given below.

Benton [7] demonstrates a series of lemmas, including the lemma detailed above, on the behaviour of renamings and substitutions for a simply typed language. I decided that proving these lemmas for the DC was of lower priority than continuing with the extensions as I had planned, as the soundness of the CPS transformation is a significantly more important, and more interesting, property. At the time of writing these proofs have not been completed, this is the reason for the ‘hole’ in the above example.

4.3 Simulating other Calculi

The DC represents a fully expressive system of classical logic, as such, we can derive the axioms of many other logic systems in the DC. This means that the constructs of a variety of other calculi can be defined in terms of the DC. We can then use this to produce standard results of these calculi, and if these standard results behave correctly when simulated by the DC we can have confidence that our formalisation is correct.

The following 3 subsections present DC implementations of three different calculi, which I then use to demonstrate some standard results.

4.3.1 Simply-Typed Lambda Calculus

Since intuitionistic logic can be seen as a “special case” of classical logic that does not rely axioms like LEM or DNE, we expect that we should be able to embed intuitionistic computational calculi into classical ones. We demonstrate this by showing that any term of the STLC can be compiled to a DC program. To do this, we axiomatise the syntax of STLC types and terms using Agda records, detailing the operations that an Agda object must support in order to be interpreted as a STLC type or term. For example, if a type T has some inhabitant B and a binary _ ⇒ _ operator, then it can be used to generate STLC types.

```
record λ-Type (T : Set) : Set where
  infixr 7 _⇒_
  field
    B : T
    _⇒_ : T → T → T

  CN : T
  CN = (B ⇒ B) ⇒ (B ⇒ B)
```

The type declarations inside the field block define the interface that must be implemented when a record is instantiated. Definitions outside the field block may refer to the record fields (cf. interfaces and abstract classes in OOP).

The record for STLC terms is parameterised on an abstract implementation of λ-Type, an abstract inductive family for terms, as well as an abstract type of types. It has three fields for the three λ-calculus term constructors: variable, abstraction, and application. Note that variables and typing environments use the same definition as the DC, replacing Type with the abstract T.

```
record λ-Term (T : Set) (TΛ : λ-Type T) (Λ : DC.Context T → T → Set): Set where
  open DC T
  open λ-Type TΛ

  field
    ` : ∀ {Γ A} → Γ ∋ A → Λ Γ A
    λ : ∀ {Γ A B} → Λ (Γ , A) B → Λ Γ (A ⇒ B)
    _·_ : ∀ {Γ A B} → Λ Γ (A ⇒ B) → Λ Γ A → Λ Γ B
```

```
z : ∀ {Γ} → Λ Γ CN
z = λ (λ (# 0))

s : ∀ {Γ} → Λ Γ (CN ⇒ CN)
s = λ (λ (λ (# 1 · ((# 2 · # 1) · # 0))))
```

Within these records we have also defined Church numerals, deriving a Church numeral type (CN) with zero and successor constructors (z and s), alongside addition and multiplication functions (sum and mult). These definitions are all standard. Note that # is a shorthand allowing us to refer to variables by their de Bruijn index.

We can now use the DC to implement these STLC interfaces. λ-Type is implemented by instantiating T to Type and providing the base type B and function type _⇒_ with DC implementations: the base type X and Call-by-Value function type $\_\Rightarrow^{V}\_$.

We then implement λ-Term by instantiating T to Type, providing our implementation of λ-Type, and instantiating the abstract inductive family to a right sequent with a fixed, empty, succedent Context. The STLC variable and abstraction constructors are handled by the DC's variable constructor and derived Call-by-Value abstraction. Function application is slightly harder, as application in the DC is an operation on coterms, while our implementation of the STLC uses right sequents, which can only contain terms. We work around this by abstracting on M and N, proofs of $A \supset B$ and A respectively, and then using proof by contradiction to establish B. This is done by abstracting on a covariable, a refutation of B, and then contradicting M, the proof of $A \supset B$, with a refutation of this proposition. We can derive this refutation, by applying N to the covariable we abstracted on.

```
DC-λ-Term : λ-Term Type DC-λ-Type (λ Γ A → Γ ⟶ ∅ | A)
DC-λ-Term = record {
  ‘ = ‘_ ;
  λ = λᵛ_ ;
  _·_ = λ M N → μθ (wkΘᵗ M ● wkΘᵗ N ·ᵛ θ 0)
  }
```

This implementation, alongside the definition of church numerals, allows us to demonstrate that the DC can simulate simple arithmetic, despite it not containing any numerals. We achieve this by using the CPS transformation to interpret Church numeral terms as natural numbers, in a rather convoluted way.

This, in fact, speaks to a fundamental weakness of the DC, in that we cannot use it to write "practical" programs. While it contains a base type X, this exists only to ensure that the type syntax is well-founded; it has no inhabitants and no operations are defined on it. This means that there are no values in the DC that we can operate on, as such, it should not be surprising that it is difficult to write interesting programs. We could get around this by extending the language with natural numbers and primitive recursion like in Gödel's System T, however we run into an interesting issue: one cannot add new terms to the DC without adding dual coterms. This suggests that we could extend the DC with inductive and coinductive types, however this is a large piece of work in its own right and is outside the scope of my project. This is why we employ the convoluted workaround with Church numerals. This issue also impacted my aim to time the execution times of CbV and CbN evaluation, as this was based on the assumption that I would be able to produce programs that were complex enough to demonstrate the difference, which turned out not to be the case.

We first declare a module, in which we will produce natural numbers from DC programs; importing λ-Type and λ-Term, instantiated with their respective DC implementations, as well as the CPS transformation, with R instantiated to ℕ. This means that a CPS transformation of a term with type A has type (A → ℕ) → ℕ, – it takes a continuation and returns a natural number.

```
module STLC-DC where
  open λ-Type DC-λ-Type
  open λ-Term DC-λ-Term
```

We then define a church numeral term using the s and z constructors we defined, and sum two of them using the sum function.

```
z : ∀ {Γ} → Λ Γ CN
z = λ (λ (# 0))

s : ∀ {Γ} → Λ Γ (CN ⇒ CN)
s = λ (λ (λ (# 1 · ((# 2 · # 1) · # 0))))
```

We then convert CN-2+2 into a term of type X. Note that CN is shorthand for (B → B) → (B → B), and that in our DC implementation of the STLC B is instantiated to X. This means that if we apply a term of type CN to terms of type X → X (a successor function) and X (zero), we will get a term of type ℕ. This is shown below, with the aforementioned terms derived from variables of the same type.

```
2+2 : (∅ , X , X ⇒ X) ⟶ ∅ | X
2+2 = CN-2+2 · (# 0) · (# 1)
```

We can check the correctness of this arithmetic by applying the CPS transformation, resulting in a term of type ℕ that we can normalise. After applying the CPS transformation we must pass a pair of interpreted Context and a continuation. The first interpreted Context consists of the CPS transformations of zero and the successor function, the second is empty. The continuation is the identity in this case.

```
2+2ᵛ : ℕ
2+2ᵛ = (2+2 ᵛᴸ) ⟨ ⟨ ⟨ tt , ℕ.zero ⟩ , (λ{ ⟨ n , k ⟩ → k (ℕ.suc n) }) ⟩ , tt ⟩ (λ x → x)
```

We can use an Agda command to normalise this term, revealing that it is equal to 4. This shows us that the DC formalisation is capable of simulating arithmetic correctly, I derived a similar result for the mult function that I do not show here.

4.3.2 Simply-Typed Lambda Calculus with letcont and throw

The Types lecture course [26] introduces a simple extension of the STLC, which I will call the STLC+. This extends the STLC to classical logic through the inclusion of two new term constructors: `letcont`, which abstracts on the current continuation, and `throw`, which contradicts a term and a continuation. This requires that we include the negated type to represent continuations. The interface and implementation of this is a simple extension of the STLC so I will not describe it in detail, I simply present the interface for, and implementation of, `letcont` and `throw`.

```
letcont : ∀ {Γ A} → Λ (Γ , ¬' A) A → Λ Γ A
throw[_,_] : ∀ {Γ A B} → Λ Γ (¬' A) → Λ Γ A → Λ Γ B
```

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \neg\neg A, \mu : \neg A \vdash x : \neg\neg A}\,(\text{Id}) \qquad \dfrac{}{x : \neg\neg A, \mu : \neg A \vdash \mu : \neg A}\,(\text{Id})}{x : \neg\neg A, \mu : \neg A \vdash \texttt{throw}[x, \mu] : A}\,(\text{throw})}{x : \neg\neg A \vdash \texttt{letcont}\,\mu : \neg A.\texttt{throw}[x, \mu] : A}\,(\text{Cont})}{\varnothing \vdash \lambda x : \neg\neg A.\texttt{letcont}\,\mu : \neg A.\texttt{throw}[x, \mu] : \neg\neg A \supset A}\,(\to\text{I})$$

```
DNE : ∀ {Γ A} → Γ ⟶ ∅ | (¬' (¬' A)) ⇒ A
DNE = λ (letcont throw[ (# 1) , (# 0) ])

_ : ∀ {Γ A} →
    DNE {Γ}{A}
    ≡ not[ μγ (γ 0 • fst[ μγ (γ 1 • snd[ not⟨
        μθ (not[ θ 0 ] • μγ (μθ (γ 0 • μγ (γ 2 • not⟨ γ 0 ⟩)) • θ 0)) ⟩ ]) ]) ]
_ = refl
```

Figure 4.3.1: Derivation tree for an STLC+ proof of the Law of Double Negation Elimination, as well as its encoding in the DC implementation of STLC+

```
; letcont = λ M → μθ (not[ (θ 0) ] • (μγ ((wkΘ^t M) • (θ 0))))
; throw[_,_] = λ M N → μθ (wkΘ^t N • μγ ((wkΘ^t (wkΓ^t M)) • not⟨ (γ 0) ⟩))
```

We can now use this calculus to prove some simple theorems of classical logic, namely the law of double negation elimination (DNE) and Peirce's law. Figure 4.3.1 presents a derivation tree of the proof of DNE as well as its encoding in Agda and the DC term it normalises to. Figure 4.3.2 does the same for Peirce's law.

It is also possible to define conjunction and disjunction in terms of implication in classical logic, I demonstrate this by proving the introduction and elimination rules for a derived definition of conjunction.

```
and : ∀ A B → Type
and A B = ¬' (A ⇒ ¬' B)

and-I : ∀ {Γ A B} → Γ , A , B ⟶ ∅ | and A B
and-I = letcont throw[ ((DNE · (# 0)) · (# 2)) , (# 1) ]

and-E₁ : ∀ {Γ A B} → Γ , and A B ⟶ ∅ | A
and-E₁ = letcont throw[ (# 1) , (λ throw[ (# 1) , (# 0) ]) ]

and-E₂ : ∀ {Γ A B} → Γ , and A B ⟶ ∅ | B
and-E₂ = letcont throw[ (# 1) , (λ (# 1)) ]
```

4.3.3 Lambda-Mu Calculus

We can also simulate Parigot's $\lambda\mu$-Calculus [35] within the Dual Calculus. This interface and implementation are slightly more complex as the $\lambda\mu$ contains both terms and commands, however it is conceptually closer to the DC as it has dual Contexts. In fact, the $\lambda\mu$-Calculus is essentially a version of the DC in which coterms can only be covariables. We define terms in the same way as we did for the STLC, though we no longer fix the

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Id)}
\UnaryInfC{$f : (A \supset B) \supset A \vdash f : (A \supset B) \supset A$}
\AxiomC{}
\RightLabel{(Id)}
\UnaryInfC{$\mu : \neg A \vdash \mu : \neg A$}
\AxiomC{}
\RightLabel{(Id)}
\UnaryInfC{$x : A \vdash x : A$}
\RightLabel{(throw)}
\BinaryInfC{$x : A, \mu : \neg A \vdash \texttt{throw}[\mu, x] : B$}
\RightLabel{(→I}
\UnaryInfC{$\mu : \neg A \vdash \lambda x : A.\texttt{throw}[\mu, x] : A \supset B$}
\RightLabel{(→E)}
\BinaryInfC{$f : (A \supset B) \supset A, \mu : \neg A \vdash f\,(\lambda x : A.\texttt{throw}[\mu, x]) : A$}
\RightLabel{(Cont)}
\UnaryInfC{$f : (A \supset B) \supset A \vdash \texttt{letcont}\,\mu : \neg A.f\,(\lambda x : A.\texttt{throw}[\mu, x]) : A$}
\RightLabel{(→I)}
\UnaryInfC{$\varnothing \vdash \lambda f : (A \supset B) \supset A.\texttt{letcont}\,\mu : \neg A.f\,(\lambda x : A.\texttt{throw}[\mu, x]) : ((A \supset B) \supset A) \supset A$}
\end{prooftree}
$$

```
peirce : ∀ {Γ A B} → Γ ⟶ ∅ | ((A ⇒ B) ⇒ A) ⇒ A
peirce = λ (letcont (# 1 · (λ throw[ # 1 , # 0 ])))

_ : ∀ {Γ A B} →
    peirce {Γ}{A}{B}
    ≡ not[ μγ (γ 0 • fst[ μγ (γ 1 • snd[ not⟨ μθ (
        not[ θ 0 ] • μγ ( μθ (γ 1
        • not⟨ ‘⟨ not[ μγ (γ 0 • fst[ μγ (γ 1 • snd[ not⟨ μθ (γ 0 •
        μγ ( γ 3 • not⟨ γ 0 ⟩)) ⟩ ]) ]) ]
        , not[ θ 0 ] ⟩ ⟩) • θ 0)) ⟩ ]) ]) ]
_ = refl
```

Figure 4.3.2: Derivation tree for an STLC+ proof of Peirce's law, as well as its encoding in the DC implementation of STLC+

succedent Context. We also include a μ binder field implemented with the DC's $\mu\theta$ constructor. Commands are implemented with DC statements, they have one constructor that can be seen below. It is interesting to note that the implementation provided matches the equational correspondence between the DC and the $\lambda\mu$-Calculus that Wadler defines in his 2005 follow-up [47] to the original paper.

```
record λμ-Term (T : Set) (TΛ : λ+-Type T) (Λ-Term : DC.Context T → DC.Context T → T → Set)
  (Λ-Comm : DC.Context T → DC.Context T → Set): Set where
  open DC T
  open λ+-Type TΛ
  field
    ‘ : ∀ {Γ Δ A} → Γ ∋ A → Λ-Term Γ Δ A
    λ : ∀ {Γ Δ A B} → Λ-Term (Γ , A) Δ B → Λ-Term Γ Δ (A ⇒ B)
    _·_ : ∀ {Γ Δ A B} → Λ-Term Γ Δ (A ⇒ B) → Λ-Term Γ Δ A → Λ-Term Γ Δ B
    μ : ∀ {Γ Δ A} → Λ-Comm Γ (Δ , A) → Λ-Term Γ Δ A
```

```
record λμ-Command (T : Set) (TΛ : λ+-Type T) (Λ-Term : DC.Context T → DC.Context T → T → Set) (Λ-Comm : DC.Con
  open DC T
  open λ+-Type TΛ
  field
    [_]_ : ∀ {Γ Δ A} → Δ ∋ A → Λ-Term Γ Δ A → Λ-Comm Γ Δ
```

```
DC-λμ-Command : λμ-Command Type DC-λ+-Type _⟶_|_ _⟼_
DC-λμ-Command = record
  {
    [_]_ = λ α M → M ● (‘ α)
  }
```

The $\lambda\mu$-Calculus is, in fact, isomorphic to *minimal* classical logic, this means it enforces Peirce's law, though not the Law of Double Negation Elimination [3]. This means that we cannot derive a DC term of type $\neg\neg A \supset A$ using only the constructs of the $\lambda\mu$ that we have defined. It is, however, possible to validate Peirce's law, this proof is given below.

```
λμ-peirce : ∀ {Γ Δ A B} → Γ ⟶ Δ | ((A ⇒ B) ⇒ A) ⇒ A
λμ-peirce = λ (μ ([ ‘Z ] ((# 0) · (λ (μ ([ ‘S ‘Z ] (# 0)))))))

_ : ∀ {Γ Δ A B} →
    λμ-peirce {Γ}{Δ}{A}{B}
    ≡ not[ μγ (γ 0 ● fst[ μγ (γ 1 ● snd[ not⟨ μθ (
        μθ (γ 0
        ● not⟨ ‘⟨ not[ μγ (γ 0 ● fst[ μγ (γ 1 ● snd[ not⟨ μθ (γ 0 ●
        θ 2) ⟩ ]) ]) ]
        , not[ θ 0 ] ⟩ ⟩) ● θ 0) ⟩ ]) ]) ]
_ = refl
```

Note that the DC term it reduces to is similar to that of the STLC+ proof of Peirce's law, with the 1st, 3rd and 5th lines being identical.

4.4 Experience with Agda

A consistent experience of producing formal proofs is how much time must be spent proving properties that are implicitly assumed in mathematical proofs. A prime example of this was handling substitution. The original paper does not provide a definition of substitution in the DC, its existence is declared and all manners of implicit assumptions about its behaviour are made, for example, that it is type preserving. This is sensible in the context of the paper, substitution is well understood and is not the paper's main focus. The word 'substitution' appears ten times in the original paper; 'renaming' is not mentioned. In contrast to this, sub or ren appear 1333 times in my source code, and 'substitution' or 'renaming' appear 111 times in my dissertation. I believe this effectively highlights the point: to formally prove only a subset of the propositions of the original paper, I had to write hundreds of lines of code to formally define, and prove properties of, something that is hardly mentioned by Wadler. Any proof of any property over the operational semantics required engaging with renaming and substitution, and as a result, proving a series of lemmas about how this property interacted with *weakened*, *lifted* and *fmap-ed* substitutions and renamings.

cps-V is an example of the exact opposite: the original paper making a point that did not initially seem particularly important or relevant to our formalisation, that ended up being critical. I had been stuck on the proof of the soundness of the denotational semantics, and the lemmas I had left to prove appeared to be impossible. While working on a separate part of the formalisation I noticed that implementing cps-V, a proof of Proposition 6.4 from the original paper, would be necessary. This proposition then transpired to be a generalisation of the majority of the lemmas that the soundness proof had been blocked on, despite not seeming particularly important.

Chapter 5

Conclusion

5.1 Overview of Results

The Dual Calculus is a product of research into both extending the Curry-Howard correspondence to classical logic, and establishing formally that Call-by-Value is the de Morgan dual of Call-by-Name. The aim of my project was to formalise the DC using Agda, as well as proving a series of propositions about it. This included the claim that its Call-by-Value semantics, both operational and denotational, are dual to its Call-by-Name semantics. This had not been attempted with a proof assistant before.

The first challenge I faced was learning Agda, a language I had no experience of using before starting this project. Another difficulty of the implementation was the complexity of defining renaming and substitution, and the resultant complexity of proving lemmas about these operations. If I could redo the project, I would spend more time looking into standard methods of implementing common operations, like substitution and renaming; trying to learn these standard methods at the same time as extending them to the DC proved difficult. Despite this, I was able to complete the core parts of the formalisation, as well as multiple extensions.

One early design choice that was validated throughout my project was the use of an intrinsically-typed representation of syntax. While this definitely made some definitions and proofs significantly more involved, the cast-iron guarantee that any DC sequent produced was both type- and scope-safe, and the requirement to define operations that maintain that guarantee, was of significant help throughout.

In hindsight, the original method of evaluation that I laid out in my proposal was poor. This decision was made out of a desire to have some quantitative data that I could present and compare, however, it would have said little about whether the project was successful. This is because transforming the DC into *efficient* Agda programs was not a goal of this project, nor would it have been sensible or worthwhile. I believe an analysis of some of the meaningful properties of my formalisation is much more valuable than producing graphs and tables of data for the sake of it.

This means that I achieved all my success criteria bar one, for which I carried out a different, and I would argue better, method of evaluation. I also implemented several extensions. As such, I believe my project to have been a success.

5.2 Future Work

The research in this field has evolved a lot since the publication of the original paper, however, there remains some interesting properties of the Dual Calculus that could be formalised. This includes both strong normalisation and confluence, both of which are claimed without proof in the original paper. I believe it would also be interesting to formalise the ‘target calculus’ of the CPS transformation, before implementing a CPS transformation into, and back out of, this calculus. One could then demonstrate a strong relationship between the reduction relations of, and transformations between, the DC and target calculus. Completing the proofs of the lemmas outlining the behaviour of substitution and renaming laid out by Benton [7] would also be worthwhile.

In his follow-up to the original paper [47], Wadler demonstrates an equational correspondence between the DC and the $\lambda\mu$-Calculus. Formalising the $\lambda\mu$ with the aim of demonstrating this correspondence would certainly be an interesting task.

Extending the DC with inductive and coinductive types would allow the definition of much more interesting data types, such as natural numbers. The lack of interesting data structures in the DC by default is currently the main limitation in using it to express anything particularly meaningful, so this extension would certainly be worthwhile.

Bibliography

[1] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. Proceedings of the ACM on Programming Languages, 2(ICFP):1–30, 2018.

[2] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In International Workshop on Computer Science Logic, pages 453–468. Springer, 1999.

[3] Zena M Ariola and Hugo Herbelin. Minimal classical logic and control operators. In International Colloquium on Automata, Languages, and Programming, pages 871–885. Springer, 2003.

[4] Franco Barbanera and Stefano Berardi. A symmetric lambda calculus for classical program extraction. Information and computation, 125(2):103–117, 1996.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. PhD thesis, Inria, 1997.

[6] Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. Science of computer programming, 23(2-3):287–311, 1994.

[7] Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly typed term representations in coq. Journal of automated reasoning, 49(2):141–159, 2012.

[8] Richard Bird. De bruijn notation as a nested datatype. Journal of functional programming, 9(1), 1999.

[9] Arthur Charguéraud. The locally nameless representation. Journal of automated reasoning, 49(3):363–408, 2012.

[10] Alonzo Church. A formulation of the simple theory of types. The journal of symbolic logic, 5(2):56–68, 1940.

[11] Tristan Crolard. A confluent lambda-calculus with a catch/throw mechanism. Journal of Functional Programming, 9(6):625–647, 1999.

[12] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. ACM sigplan notices, 35(9):233–243, 2000.

[13] Haskell B Curry. Functionality in combinatory logic. Proceedings of the National Academy of Sciences of the United States of America, 20(11):584, 1934.

[14] Haskell Brooks Curry. Grundlagen der kombinatorischen logik. American journal of mathematics, 52(4):789–834, 1930.

[15] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. Combinatory logic, volume 1. North-Holland Amsterdam, 1958.

[16] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In Indagationes Mathematicae (Proceedings), volume 75, pages 381–392. Elsevier, 1972.

[17] Peter Dybjer. Inductive families. Formal aspects of computing, 6(4):440–465, 1994.

[18] Gergő Érdi. Generic description of well-scoped, well-typed syntaxes. arXiv preprint arXiv:1804.00119, 2018.

[19] Andrzej Filinski. Declarative continuations and categorical duality. Citeseer, 1989.

[20] Gerhard Gentzen. Investigations into logical deduction. American philosophical quarterly, 1(4):288–306, 1964.

[21] Timothy G Griffin. A formulae-as-type notion of control. In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 47–58, 1989.

[22] Robert Harper. Practical foundations for programming languages. Cambridge University Press, 2016.

[23] William A Howard. The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism, 44:479–490, 1980.

[24] Chantal Keller. The category of simply typed λ-terms in agda, 2008.

[25] Daisuke Kimura et al. Computation in classical logic and dual calculus. 2007.

[26] Neel Krishnaswami. Types. Cambridge University lecture notes, 2020.

[27] Cristina Matache. Formalisation of the $\lambda\mu$ t-calculus in isabelle/hol computer science tripos–part ii fitzwilliam college may 16, 2017. 2017.

[28] Conor McBride. Type-preserving renaming and substitution. 2005.

[29] Conor McBride and James McKinna. Functional pearl: i am not a number–i am a free variable. In Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, pages 1–9, 2004.

[30] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. Formal aspects of computing, 10(2):171–186, 1998.

[31] Ulf Norell. Towards a practical programming language based on dependent type theory, volume 32. Citeseer, 2007.

[32] Ulf Norell. Dependently typed programming in agda. In International school on advanced functional programming, pages 230–266. Springer, 2008.

[33] C-HL Ong and Charles A Stewart. A curry-howard foundation for functional computation with control. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 215–227, 1997.

[34] CHL Ong. A semantic view of classical proofs: Type-theoretic. Categorical, and Denotational Characterizations, Linear Logic, 96, 1996.

[35] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In International Conference on Logic for Programming Artificial Intelligence and Reasoning, pages 190–201. Springer, 1992.

[36] Lawrence C Paulson. Natural deduction as higher-order resolution. The Journal of Logic Programming, 3(3):237–258, 1986.

[37] Lawrence C Paulson. Logic and proof. Cambridge University lecture notes, 2008.

[38] Lawrence C Paulsson and Jasmin C Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Proceedings of the 8th International Workshop on the Implementation of Logics (IWIL-2010), Yogyakarta, Indonesia. EPiC, volume 2, 2012.

[39] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. ACM sigplan notices, 23(7):199–208, 1988.

[40] John C Reynolds. The meaning of types from intrinsic to extrinsic semantics. BRICS Report Series, 7(32), 2000.

[41] Moses Schönfinkel. Über die bausteine der mathematischen logik. Mathematische annalen, 92(3):305–316, 1924.

[42] Peter Selinger. Control categories and duality: an axiomatic approach to the semantics of functional control. Talk presented at Mathematical Foundations of Programming Semantics, London, 1998.

[43] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. Mathematical Structures in Computer Science, 11(2):207–260, 2001.

[44] The Agda Team. Agda documentation, 2020.

[45] Nikos Tzevelekos. Investigations on the dual calculus. Theoretical computer science, 360(1-3), 2006.

[46] Philip Wadler. Call-by-value is dual to call-by-name. In Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, pages 189–201, 2003.

[47] Philip Wadler. Call-by-value is dual to call-by-name–reloaded. In International Conference on Rewriting Techniques and Applications, pages 185–203. Springer, 2005.

[48] Philip Wadler, Wen Kokke, and Jeremy G. Siek. Programming Language Foundations in Agda. July 2020.

[49] Stephanie Weirich, Brent A Yorgey, and Tim Sheard. Binders unbound. ACM SIGPLAN Notices, 46(9):333–345, 2011.

[50] Glynn Winskel. The formal semantics of programming languages: an introduction. MIT press, 1993.