Ted While

Formalisation of the Dual Calculus in Agda

Computer Science Tripos – Part II

Fitzwilliam College

DATE HERE

Declaration of Originality

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

Signed: Edward While

Date: DATE HERE

Proforma

Name: Edward While

College: Fitzwilliam College

Project Title : Formalisation of the Dual Calculus in Agda

Examination: Computer Science Tripos – Part II, June 2021

Word Count:

Line Count:

Project Originator: Dmitrij Szamozvancev

Project Supervisor: Dmitrij Szamozvancev

Original Aims

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.

Work Completed

All the main goals of the project have been achieved. A formalisation of the Dual Calculus was implemented using an intrinsically-typed representation of its syntax. This formalisation was then used to prove all the properties I originally aimed to prove. I also implemented an operational semantics of the Dual Calculus as an extension, this allowed me to carry out further extensions, proving the duality of the operational semantics as well as the soundness of the CPS denotational semantics. I then used the DC formalisation to simulate multiple different calculi.

Special Difficulties

None.

Contents

Chapter 1

Introduction

This chapter aims to describe the context the project exists in and introduce the project itself. It begins with a description of the structure of the various parts of the project, followed by the motivation behind undertaking the project, and a short summary of existing related work.

1.1 Motivation

1.1.1 The Dual Calculus

The Curry–Howard correspondence sets out a relationship between programming language theory and logic, its most well-known incarnation being the relationship between Church's Simply-Typed λ-Calculus [9] and Gentzen's Intuitionistic Natural Deduction [19]. It was, however, years after this relationship was first noted that it was first extended to classical logic. Interestingly, the prevailing thought at the time was that classical logic did not have a computational interpretation, and the discovery was essentially made by accident. This was when Griffin [20] demonstrated that the type call/cc – which gave explicit control over the continuation of a program – corresponded to Peirce's

Law, a law that holds only in classical logic.

It was Filinski [17] who first suggested that a duality existed between Call-by-Value and Call-by-Name evaluation strategies when programming with continuations. A series of papers of papers have been published since then, each trying to demonstrate this duality in a different way. The first to make this argument in a completely convincing manner was Wadler [42].

Wadler presented the DC, a calculus that corresponds to Gentzen's classical sequent calculus in the same way the STλC corresponds to intuitionistic natural deduction. The key contribution of the DC, however, is how explicitly, and cleanly, it demonstrates the duality between Call-by-Value and Call-by-Name.

1.1.2 Choice of Programming Language

The first reason I decided to use Agda [28] for this project, instead of any other proof assistant, is that it is one of the more widely used proof assistants, as such a good level of documentation and support exists online, including Wadler's excellent Programming Language Foundations in Agda (PLFA) [44]. Proofs in Agda are spelled out explicitly, rather than being hidden behind opaque tactics like in Coq [5] or Isabelle [33], and the syntax of Agda is a lot cleaner and more flexible, allowing mixfix operators and use of unicode characters. This means that proofs in Agda are significantly more readable than those in alternative proof assistant. Another key benefit is that the language is introduced in the Tripos and is easier to pick up if you have experience of functional programming, which appears regularly in the tripos. A final, more practical consideration was that it's the main language my supervisor (and the project originator) works in, and as such I would better be able to draw on his knowledge for assistance when needed.

1.1.3 Contribution of my Project

The original paper that sets out the DC presents all of the theorems and properties of the calculus without proof, as such formalising the language and then producing formal proofs of said theorems and properties can give us much greater confidence in the correctness of the paper. This is a well-documented benefit of formal verification, possibly the most-known example of this being Nipkow's paper 'Winskel is (almost) right' [27] in which he presents a formalisation of the first 100 pages of Winskel's textbook "The Formal Semantics of Programming Languages", uncovering one serious mistake in his completeness proof for Hoare Logic. Having looked through the literature I believe I am the only person to produce such a formalisation of the DC.

Producing such formal proofs, however, is "extremely laborious" [35] and time-consuming. Knowing this, and having no experience in writing in Agda or any other proof assistant before starting this project, I decided to exclude the formalisation of the operational semantics of the calculus and the proof of the soundness of the CPS transformations from the core of my project. Instead these were included as extensions, which I was fortunately able to complete.

1.2 Project Structure

The aim of the project was to formalise the Dual Calculus (DC), as set out by Phillip Wadler [42] that defines the language's syntax and semantics, as well as various properties of it. I have split up the formalisation of the language into three parts: the syntax, the operational semantics, and the denotational semantics. Definitions and proofs regarding these concepts are found in Sections 3.2, 3.3, and 3.4 respectively. Section 3.5 then discusses, and demonstrates the proofs of, the duality of each of the concepts defined in the three previous parts.

Note that the definition of an operational semantics for the DC, which necessitated a formal

definition of substitution, was completed as an extension. As was the proof of 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 in Agda is itself an evaluation of the project, though I used other methods to evaluate whether I was successful. This included using the formalised DC to produce 'unit tests' to check that my formalisation was behaving correctly (in Section 4.2). As well as this I used the DC formalisation to simulate a variety of other Calculi, both classical and intuitionistic, in 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.

1.3 Related Work

Tzevelekos [41] 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 [20] first introduced the idea that there was a correspondence between languages with control operators and classical logic, and several calculi that exemplify this have been produced. The most well-known is Parigot's $\lambda\mu$-calculus [32], which extends the Simply Typed λ-calculus 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. Both Call-by-Name and Call-by-Value semantics for $\lambda\mu$ have been investigated by Ong [31], and Ong and Stewart [30] respectively. Crolard then derived a λ-calculus with a catch/throw mechanism from this, which he calls the λ_{ct}-calculus [10]. Separately, Barbanera produced a λ-calculus with symmetric reduction rules that corresponded to classical logic [4], as well as proving the property of strong normalisation for the calculus.

There have also been various investigations into the duality of Call-by-Value and Call-by-Name

since Filinski first introduced the idea [17]. Sellinger, in two separate papers [38, 39], modeled the Call-by-Name and Call-by-Value semantics of the $\lambda\mu$-calculus in a control category and a dual co-control category respectively, the duality between these categories however is not an involution. This work was then improved upon by Curien and Herbelin who, exploiting the fact that the sequent calculus better displays the duality of classical logic, derived a computational calculus from the classical sequent calculus to demonstrate the duality [11]. In doing this, however, they introduce a difference operator as the dual of implication; the computational interpretation of this operator is far from intuitive. Barbenera's symmetric λ-calculus [4] also has a clear notion of duality, however they do not consider Call-by-Name and Call-by-Value evalutation strategies. Wadler's DC demonstrates the duality without any of the limitations of the above.

Chapter 2

Preparation

provides

on

This chapter aims to provide the necessary background knowledge of ~~the~~ various aspects of the

you can refer to it as DC in the body, but it's good to write it out with the first mention in a chapter or section

project, primarily Agda and the DC, as well as outlining the starting point of the project and the

tools I used for my implementation.

I also give the point of the project, and list the tools I used for the implementation.

do a find and replace of
Curry-Howard
to
Curry--Howard

2.1 Curry-Howard Correspondence

This section outlines the Curry—Howard correspondence, ...

This section serves as a brief introduction to the Curry-Howard correspondence, a concept that appears frequently throughout my project.

"relates programs and mathematical proofs"

"first observed this relationship"

The Curry-Howard correspondence, also known as the formulae-as-types interpretation, is a relationship between 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 variation of intuitionistic logic [12]. Over 20 years later Curry discovered another example of this relationship [14], a correspondence between Hilbert-style deduction systems and combinatory logic [37, 13] (a model of computation, not a logical system as the name might suggest). One could argue, however, that the first person to spell out this

axiom schemes
(I know Wikipedia has the hyphen but me no likey)

I don't think the lengthy historical discussion is really needed, other than mentioning the two milestones in 1-2 sentences. But do give a brief summary of the correspondence itself (referring to some lines in the table), and it's probably better to give the specifics of extending it to classical logic in this section, rather than the intro.

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

correspondence explicitly was William Howard, who, in 1969, demonstrated a direct and specific relationship between Intuitionistic Natural Deduction and the Simply-Typed λ-calculus [22]. The correspondence has since been extended to Classical Logic, as was detailed in Section 1.3.

2.2 The Dual Calculus

The DC represents the computational interpretation of Gentzen's classical sequent calculus. Taking inspiration from the original formulation of duality in logic, conjunction, disjunction, and negation are primitive, while implication is defined in terms of these connectives. This definition of implication is different depending on whether Call-by-Value or Call-by-Name semantics are used. The following section outlines the calculus.

better to have in the implication sect

2.2.1 Syntax

The syntax of the DC is as follows:

Definition 2.2.1 (Syntax)**.**

Term	$M, N ::= x \mid \langle M, N\rangle \mid \langle M\rangle\text{inl} \mid \langle N\rangle\text{inr} \mid [K]\text{not} \mid (S).\alpha$
Coterm	$K, L ::= \alpha \mid [K, L] \mid \text{fst}[K] \mid \text{snd}[L] \mid \text{not}\langle M\rangle \mid x.(S)$
Statement	$S ::= M \bullet K$

A DC program is in the form of either a term, coterm, or statement. Terms produce values and are in the form of a variable; a pair of terms; a left or right injection of a term; the negation of a coterm; a λ-abstraction of a term; or a covariable abstraction of a statement. Coterms consume values and occur as a covariable; a left or right projection of a coterm; a case of two coterms; the negation of a term; a functional application of a term to a coterm; or a variable abstraction of a statement. A statement is the cut of a term against a coterm. These will be explained in greater detail later.

Variables, covariables, terms and coterms, all have types, the types of the DC are defined as:

Definition 2.2.2 (Grammar of Types)**.**

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

The types of the DC are familiar, a type is either the base type X; a product of types; a sum of types; the negation of a type; or a function from one type to another.

2.2.2 Typing Relation

The existence of (co)variable abstractions requires the tracking of the types of ~~the various~~ free (co)variables. This is done in two separate typing environments: an antecedent is a list of pairs of free *variables* and types; while a succedent is a list of pairs of free *covariables* and types.

Here is another example of how the text could be tightened up and made less like a specification (as would be given in an academic paper) and more like an explanation. You are free to adopt/adapt this but more importantly it should give you an idea on how to approach editing the rest of the text yourself. Often it's not a matter of small phrasing tweaks but an entire rewrite of a paragraph or section, condensing things into something just as informative, but shorter and more to the point. This takes time and some thinking, but is very much worthwhile to cut down on words and streamline the overall writing.

2.2 The Dual Calculus

The Dual Calculus is a formal system that can be interpreted as a language of proof terms for Gentzen's classical sequent calculus, or a programming language with explicit control of the call stack. This section outlines the syntax of DC, along with its logical and computational content.

2.2.1 Syntax

"business card" of the DC: what it is and what is it trying to do

As its name suggests, the motivating idea behind the DC is capturing duality: namely, the de Morgan duality between connectives, and the computational duality between producers and consumers of values.

DC has a minimal type system consisting of a base type, products, sums, and negation:
[type def]
The types are introduced by terms, e.g. pairs <M,N> for products, and injections <M>inl and <N>inr for sums. Elimination forms become coterms that consume values of some type: for example, first projection fst[K] consumes a product, and case analysis [K,L] consumes a sum. A term and coterm of the same type can be combined into a statement, and a statement can be turned into a term or a coterm by abstracting on a variable or covariable. The logical and computational meaning of these constructs will be explained later.
[syntax def]

brief comments on some features of the syntax: reading the text and looking at the definition for a few seconds should give the reader an idea of the general symmetric and mutually inductive nature of the syntax, and the last sentence reassures them that things will be explained in detail later.

Terms, coterms, and statements each have an associated typing judgment called a left, right, and centre sequent, respectively. Since all three syntactic forms may contain both free variables and covariables, we must keep track of their types in two typing environments in all three judgments. Wadler's notation for the three sequents is as follows, where Γ is a variable environment (antecedent) and Θ is a covariable environment (succeedent):
[sequent def]
The typing judgments are mutually inductively defined by the system of rules and axioms given on Figure 2.2.1. The rules are pleasingly symmetric and involve no syntactic elimination forms: every term introduction rule (e.g. &R) has an associated coterm introduction rule for the dual type constructor (e.g. VL). Negation converts between left and right sequents, and (co)variable abstraction turns a statement with a free (co)variable of type A into a (co)term of type A.
[fig 2.2.1 here or wherever LaTeX puts it]
At first sight, some of the rules may seem unintuitive: for example, fst[K] takes a coterm of type A and produces a coterm of type A & B, which is the opposite of what first projection would do in e.g. the simply-typed λ-calculus with products. Another unusual feature is that the typing judgment for statements doesn't actually involve a type. All these issues become clear, however, when one considers the logical and computational interpretation of the DC.

a reader with base understanding of how type systems work will understand that you need a judgment for each syntactic form, and a typing environment for each kind of variable. I decided to not mention the type-less centre sequent here and leave it as something the reader might notice and question themselves.

Anticipate the reader's questions about the direction of typing rules and the centre judgment, then end with a glue sentence that leads into the next section.

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

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

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

There are three different types of *sequent* in the DC: Left Sequents, Right Sequents, and Centre Sequents, each consists of an antecedent and a succedent. The difference between the three sequents is the location and forms of their *distinguished elements*: for right sequents this features in the succedent and takes the form of a `term:type` pair; for left sequents the antecedent contains the distinguished element, a `coterm:type` pair. Centre sequents have no distinguished element, instead they contain a statement.

Definition 2.2.4 (Sequents)**.**

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

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

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

The above sequents can be read as typing judgements: the term M has type A given typing environments Γ and Θ; the coterm K has type A given typing environments Γ and Θ; the statement S is valid given typing environments Γ and Θ. Note that statements do not have types.

Definition 2.2.5 (Typing)**.** The typing rules for the Dual Calculus are given in Figure 2.2.1

Note that all right rules result in a right sequent, and vice versa for left rules. It should also be noted that & and $\vee$ rules have the same type of sequent in the hypothesis and the conclusion whereas $\neg$ rules swap from a left to a right sequent or vice versa.

In a vacuum, 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. The reasons for this, however, become clear when one considers how to interpret the constructs of the DC, both computationally and logically.

Figure 2.2.1: The typing rules for the Dual Calculus

$$\frac{}{x\colon A \rightarrow \blacksquare\, x\colon A}\ (\textsc{Id}\mathrm{R}) \qquad \frac{}{\alpha\colon A \blacksquare \rightarrow \alpha\colon A}\ (\textsc{Id}\mathrm{L})$$

if you're doing the Reloaded-style system (without explicit structural rules), you should have the Γ and Θ contexts here too

$$\frac{\Gamma \rightarrow \Theta \blacksquare M\colon A \qquad \Gamma \rightarrow \Theta \blacksquare N\colon B}{\Gamma \rightarrow \Theta \blacksquare \langle M, N\rangle\colon A \,\&\, B}\ (\&\mathrm{R}) \qquad \frac{K\colon A \blacksquare \Gamma \rightarrow \Theta}{\mathrm{fst}[K]\colon A \,\&\, B \blacksquare \Gamma \rightarrow \Theta}\ (\&\mathrm{L1}) \qquad \frac{L\colon B \blacksquare \Gamma \rightarrow \Theta}{\mathrm{snd}[L]\colon A \,\&\, B \blacksquare \Gamma \rightarrow \Theta}\ (\&\mathrm{L2})$$

$$\frac{\Gamma \rightarrow \Theta \blacksquare M\colon A}{\Gamma \rightarrow \Theta \blacksquare \langle M\rangle \mathrm{inl}\colon A \vee B}\ (\vee\mathrm{R1}) \qquad \frac{\Gamma \rightarrow \Theta \blacksquare N\colon B}{\Gamma \rightarrow \Theta \blacksquare \langle N\rangle \mathrm{inr}\colon A \vee B}\ (\vee\mathrm{R2}) \qquad \frac{K\colon A \blacksquare \Gamma \rightarrow \Theta \qquad L\colon B \blacksquare \Gamma \rightarrow \Theta}{[K, L]\colon A \vee B \blacksquare \Gamma \rightarrow \Theta}\ (\vee\mathrm{L})$$

put VL under &R

also add some horizontal space between the rules

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

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

$$\frac{\Gamma \rightarrow \Theta \blacksquare M\colon A \qquad K\colon A \blacksquare \Gamma \rightarrow \Theta}{\Gamma \blacksquare M \bullet K \mapsto \Theta}\ (\textsc{Cut})$$

2.2.3 Computational Content

better to start with logical interpretation first I think (especially since implication is now not part of the syntax)

And maybe also explain evaluation strategies in the computational content section (without details of the evaluation relation itself)

Types The computational content of the types of the DC is clear for products $A \,\&\, B$, sums $A \vee B$, and functions $A \supset B$. The only computationally interesting type is the negated type $\neg A$. When one considers that the the logical proposition $\neg A$ is equivalent to $A \supset \bot$ it becomes clear that the type $\neg A$ is the type of a function that takes a value of type A and returns nothing. Such a function is known as a *continuation*.

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 one of the covariables in the succedent. Considering this, one can see that the interpretation of the right sequent

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

is that 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 will pass a value of type B_i to some continuation variables α_i. Similarly, the interpretation of the left sequent

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

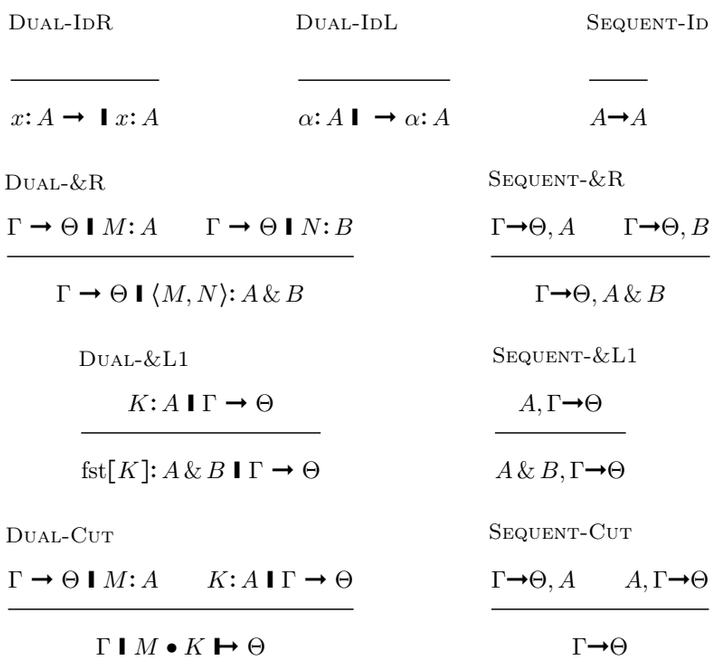
is that if one supplies a value of type A_i to each x_i, and a value of type A to the coterm K then evaluation will pass a value type B_i to some α_i. The interpretation of the centre sequent

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

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

Terms, Coterms, and Statements The computational meaning of terms, coterms and statements are detailed in Table 2.2. Terms produce values, while coterms consume them, statements simply pipe together a term and a coterm.

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



fix sequent spacing

2.2.4 Curry-Howard Correspondence for the Dual Calculus

"Philip Wadler is fond of using colours to typeset syntax definitions in his papers, writing types and contexts in blue and terms in red. By putting on red-tinted glasses, all terms are "erased" and we are left with a purely logical inference system in accordance with the Curry–Howard correspondence. If we apply this idea to the Dual Calculus, the obtained proof system is precisely Gentzen's Classical Sequent Calculus ..."

As has been mentioned, the DC corresponds to Gentzens's Classical Sequent Calculus, a side-by-side comparison of the inference rules of the Sequent Calculus and the typing relation of the DC demonstrates this clearly. This subsection discusses how to interpret the DC logically.

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. Variables and covariables simply label antecedents and succedents, which themselves should be interpreted as a sequence of propositions (under conjunction in the case of antecedents, and under disjunction for succedents). This means that left, right, and center sequents are interpreted as sequents of the Sequent Calculus:

meaning that the conjunction of the propositions in the antecedent implies the disjunction of the propositions in the succedent. For example the right sequent

$$x_1 : A_1, ..., x_m : A_m \rightarrowtail \alpha_1 : B_1, ..., \alpha_m : B_m \mathbin{\blacktriangleright} M : A$$

is interpreted to state that if all of the propositions A_i are true then one of the propositions B_i *or* A is true.

A term proves a proposition in the succedent.

The coterm K proves the proposition A in the antecedent, this is equivalent to refuting A in the succedent.

A statement represents a contradiction between a term and a coterm. Using these contradictions for proofs, as (co)variable abstractions do, is a key feature of classical logic. It relies on the law of double negation elimination ($\neg\neg AA \supset$) as we assume $\neg A$ to derive some kind of contradiction, thus demonstrating $\neg\neg A$, which, in classical logic, entails A.

The logical interpretations of terms, coterms, and statements defined in more detail in Table 2.2.

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 $\rightarrowtail \mathbin{\blacktriangleright} M : A$.*

2.2.5 Implication

As we have already mentioned, implication in the DC can be defined in terms of the other connectives, however we require two separate definitions of implication, due to the nature of the DC's reduction relation. The DC has two separate reduction relations, one Call-by-Value, the other Call-by-Name, these will be defined in the implementation chapter. These relations depend on a subset of terms and coterms, known as values and covalues, which are defined as follows:

In the discussion above, I deliberately chose not to address an important notion: functions and implication. There are several reasons for this:
* The dual of implication is not a well-known or natural construction and therefore does not fit nicely into the dual framework
* It can be defined in terms of the other constructs in different ways
* The way it is derived depends on what computational behaviour we expect from the DC
Previous literature attempted to resolve this somewhat awkward issue in many ways, none of them fully satisfactorily. Wadler outlines the drawbacks of existing approaches, but still includes implication types, abstraction and application in the syntax of DC — most likely to connect it very clearly to the sequent calculus with implication. This, however, forces him to restrict definitions and theorems to a subset of DC programs that do not involve implication, before concluding that implication is in fact derivable from other connectives depending on the evaluation strategy. It therefore made sense for me to treat implication as a purely derived notion from the start and not include it in the syntax at all.

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

	Computational Interpretation	Logical Interpretation
Terms		
x	~~The term x~~ produces the value given to the variable x.	~~The term 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.
$\langle M \rangle$inl/$\langle N \rangle$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.
$[K]$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	The coterm α consumes a value, giving it out to the covariable α.	The coterm 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 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.

help

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 $P, Q ::= \alpha \mid [P, Q] \mid \text{fst}[P] \mid \text{snd}[Q] \mid \text{not}\langle M\rangle \mid M \mathbin{@} Q$

This highlights the reason that we require two separate definitions of implication.The definition of the Call-by-Value reduction relation requires a function abstraction to be a value, however under Call-by-Name we require that a function application is a covalue.

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

$$A \supset B \equiv \neg A \,\&\, \neg B$$

$$\lambda x.N \equiv [z.(z \bullet \mathit{fst}[x.(z \bullet \mathit{snd}[\mathit{not}\langle N\rangle])])]\mathit{not}$$

$$M@L \equiv \mathit{not}\langle\langle M, [L]\mathit{not}\rangle\rangle$$

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

$$A \supset B \equiv \neg A \vee B$$

$$\lambda x.N \equiv (\langle[x.(\langle N\rangle \mathit{inr} \bullet \gamma)]\mathit{not}\rangle \mathit{inl} \bullet \gamma).\gamma$$

$$M@L \equiv [\mathit{not}\langle M\rangle, L]$$

The typing inference rules for function application and abstraction are given below, it is possible to derive these rules from the rules given in Figure 2.2.1.

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

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

Similarly, one can also derive the reduction rules for implication from those of the other connectives.

2.3 Agda

This section introduces ~~the reader to~~ the dependently-typed functional programming language Agda, and the core concepts behind it. Through the Curry-Howard correspondence and intutionistic type theory Agda can also be used as a proof assistant [29].

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. Pattern matching over algebraic data types is an important part of programming in Agda. Basic data types can be declared like so, data Name : Set where with the constructors of said data type immediately following it. Here Set is the 'type of types' [1] The classic example of data type in Agda is the natural numbers, defined below with two constructors: zero and suc.

this whole paragraph is clunky

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

To define a function over a data type we must pattern match over *all* of the possible cases. Functions are defined in agda by declaring the type of the function (dependent types make ML-style inference impossible) followed by defining its behaviour over the possible cases. Agda syntax is very flexible, allowing mixfix operators by using '_' where an argument should be expected, as well as allowing the use of unicode characters. Addition over the natural numbers is recursively defined below.

[1] This may raise the question what is the type of Set, well its Set1, which has type Set2, and so on and so on. This gets complicated, however, and we do not need to explore it here.

```
_+_ : ℕ → ℕ → ℕ

zero  + n = n

suc m + n = suc (m + n)
```

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

Curly braces can used around a function argument to declare it to be implicit, this means that the type checker will try to work out the value of the argument by itself, though it is still possible to explicitly define the value of an implicit argument. It is also possible to parameterise data types by other types, as an example the list of elements of an arbitrary type A is defined below.

explain implicit arguments when you actually have an example for them

```
data List (A : Set) : Set where

  [] : List A

  _::_ : A → List A → List A
```

2.3.2 Dependent Types

Dependent types are what give Agda much of its power, they can be formally defined as follows. Say we have a type $A : U$ where U is a universe of types, we can define a family of types $B : A \to U$, assigning each term $x : A$ a type $B(x) : U$. The family of types given by B is known as a dependent type. The most basic dependent type is the dependent function type, from the 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 just equivalent to $A \to B$, as B does not depend on x.

In Agda these dependent functions are represented as $(x :$ A$) \to$ B x where A is a type and B is a type dependent on A. A simple example of this is the polymorphic map function given below.

blue letters matter

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

Dependent types can do much more than this though, a classic example of their power is the ability to define the types of lists of specific length. This definition is given below.

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

Note that the type of Vec A is $\mathbb{N} \to$ Set, a family of types indexed by the natural numbers. This means that for each $n \in \mathbb{N}$, Vec A n is a type. We describe this Vec datatype as parameterised on type A and indexed by the natural numbers. The type of _::_ is also an example of the dependent function type 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, say, for example, we wanted to define the head operation on only non-empty lists. The Vec data type will let us do this.

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

Despite only using one case the type checker recognises that this is 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, allowing such errors to be corrected at compile-time.

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, and this is how we use Agda as a proof assistant.

Equality is an important concept in proofs, and many of the proofs I aimed to complete were proofs of equality. There are multiple types of equality in Agda, the two most important ones for this project are definitional equality and propositional equality. Definitional equality is simply 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 ($m + n$). This definitional equality implies the second type of equality that we're interested in, propostional equality. This is defined as:

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

This can be interpreted as follows: for a type A and term x, refl proves that $x \equiv x$, this means that every term is equal to itself, and that this is the only way to prove terms are equal. From this definition we can easily derive the other properties necessary for equality: substitution, transitivity, and symmetry.

It is also worth discussing universal quantification, the logical interpretation of dependent functions. The ∀ syntax in Agda is simply 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 : A) → B x if, for every term M : A we can produce a term of type B M. Therefore, a proof of the proposition ∀ (x : A) → B x takes the form of λ (x : A) → 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:

no need to confuse things with ∀ if you're also giving the type

```
+-assoc : ∀ (m n p : ℕ) → (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 requires proving a couple of lemmas, the first is that zero is the identity when on the right hand side of an addition, $\forall n \in \mathbb{N}.n + 0 \equiv n$. The first thing we must do is provide a type signature for the lemma, this looks a lot like the mathematical formulation.

typeset this as latex

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

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

```
+-identityʳ zero = refl
```

For the inductive step, m is instantiated as (suc m), therefore we must prove (suc m) + zero ≡ suc M. The definition of _+_ 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. cong states that if two terms are equivalent then the result of a given function being applied to them will also be equivalent, it takes as arguments a function and a proof of equality of two terms, returning a proof of equality of the function applied to those two terms. 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.

easier to give the type signature

```
+-identity^r (suc m) = cong suc (+-identity^r m)
```

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

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

can probably skip the details of this proof, just say that it's done similarly by induction and congruence

The base case requires us to prove that zero + suc n is equivalent to suc (zero + n). Since both these terms are definitionally equal to suc n this can be proved with refl.

```
+-suc zero n = refl
```

The inductive case requires a proof that suc m + suc n is equal to suc (suc m + n). Both terms have a definitional equality that makes this equivalent to proving suc (m + suc n) is equal to suc (suc (m + n)). Again, using congruence with suc and an appeal to the inductive hypothesis is sufficient to prove this.

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

latex

Now we can actually 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, meaning our base case needs a proof that m + zero equals zero + m, since zero + m is definitionally equal to m we prove this by our +-identity^r lemma.

```
+-comm m zero = +-identity^r m
```

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

```
+-comm m (suc n) = begin
  m + (suc n) ≡⟨ +-suc m n ⟩
  suc (m + n) ≡⟨ cong suc (+-comm m n) ⟩
  (suc(n + m))                    move box here, align vertically
  □
```

Equational reasoning allows us to write a series of equivalent equations, like a human might, separated by proofs of their equivalence inside the ≡⟨ _ ⟩ operator. The inductive case requires us to 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 again uses cong and our inductive hypothesis, this demonstrates that suc (m + n) is equivalent to suc (n + m), which itself is definitionally equal to (suc n) + m, thus completing the proof.

2.4 Software Engineering

2.4.1 Starting Point

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

As far as I am aware, no-one has produced a formalisation of the DC in 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 on this topic was the brief discussion they receive in the Part IB Logic and Proof course [34]. As soon as I knew my project would involve using Agda I started studying PLFA [44] to teach myself how to write in Agda. I also studied sections of the textbook *Practical Foundations for Programming Languages* [21] to familiarise myself with the Curry-Howard correspondence. The intrinsically-typed formalisation of the λ-calculus in Agda provided in PLFA proved useful to me throughout the duration of my project.

2.4.2 Tools and Design

I used *banacorn's agda-mode* extension for Visual Studio Code for writing all my code, this provides the ability to execute Agda commands and allows for simple unicode input in VSCode, making it an effective IDE for writing in Agda. The Agda distribution I used also has an LaTeXback-end allowing easy typesetting of Agda code, which I have used to prepare this dissertation.

mention back-ups and github and all that jazz

The DC formalisation is not a standard software project in that it does not require testing in the normal sense of the word, the proofs of various DC properties I have produced guarantee that the formalisation is correct, and the Agda type checker guarantees that the proofs are correct. Despite this, I used my DC formalisation to prove some theorems of classical logic to give me greater confidence in its correctness. As well as this I simulated the Simply-Typed Lambda Calculus (STLC) in the DC formalisation, demonstrating 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, the next two define the semantics of the DC, one from an operational standpoint, the other denotational. The final section of this chapter discusses and proves the duality of each the previous sections.

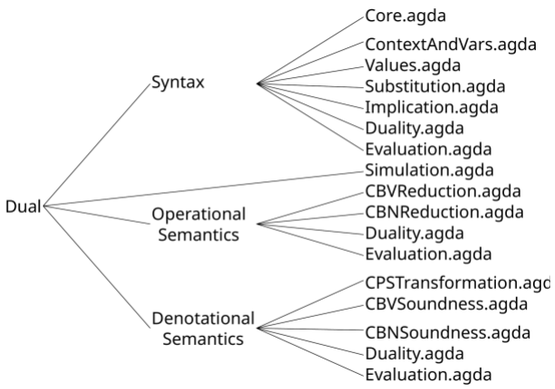
3.1 Repository Overview

modules

Most of the final source code for my project is split into three main folders, 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 particular 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. ~~Each file imports other files as necessary, for example almost all files import the core syntax of the DC, defined in `Dual.Syntax.Core`, though the proof of the Duality of the~~

~~Denotational Semantics does not need to be used by any other file.~~

The code for this project was written from scratch, though in some cases is based off standard methods for certain common problems that I extended to the DC [26, 2].



any way to remove the .agda endings

Figure 3.1.1: Tree outlining the structure of the source code repository

3.2 Syntax

This section presents the important definitions of my DC formalisation, making notice of any interesting issues, and discussing and justifying 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, often referred to as extrinsic and intrinsic encoding.

Definitions of a typed language can be *extrinsic* or *intrinsic* [36]. Terms in an extrinsic definition have a meaning that is 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 terms themselves, that meaning is assigned. These typing judgements include typing environments that track which variables are in scope of

ill-formed

the term. Representing syntax in this way makes ill-typed or ill-scoped terms meaningless. Using inductive *families* [16], rather than inductive types, 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 allows us to track the scoping information at the type level, meaning that type checking will guarantee that a term is well-scoped. This can be used to enforce scope-safety in an untyped language, and can be extended to enforce type- and scope-safety

by

in typed languages. Indexing a family on a type (a type in the language we are defining, not an Agda type), as well as a set representing the scope, will guarantee 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 either naming variables, in which case maintaining α-equivalence becomes a difficult issue, or using numeric de Bruijn indices [15] which can result into running into difficult to debug errors from having to correctly modify all de Bruijn indices each time the scope changes [25]. In an intrinsically typed representation one can define an inductive family to represent well-typed and well-scoped de Bruijn indices. As such, when

the user

defining operations that modify the scope of a term, Agda's type system will force you to modify

them

In addition,

the indices correctly, and assist you in doing so. As well as this, intrinsically-typed representations of languages tend to require significantly less code [44].

For all the above reasons I decided to use an intrinsically-typed representation of the syntax of the DC, following the formalisation of the STLC laid out in PLFA [44], which itself is based on the work of Altenkirch and Reus [2].

new section here

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

natural numbers as base type X.

can you change ‘N to something else (like B for base type), otherwise it’s misleading

```
data Type : Set where
  ‘ℕ : Type
  _‘×_ : Type → Type → Type
  _‘+_ : Type → Type → Type
  ‘¬_ : Type → Type
```

For an intrinsically-typed representation of the DC, terms, coterms and statements must be indexed by two sets representing the scope of the program. In Agda, these contexts are represented as a list of Types:

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

We can then define free variables and covariables as an inductive family indexed by a Context and a Type. There are two constructors for the datatype: ‘Z proves that the variable exists by showing it is the most recently bound variable, ‘S uses evidence of a variable existing in a context to prove it exists in a context with one more variable bound. 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, as previously mentioned, variables can be seen as type- and scope-safe de Bruijn indices.

is in a context

`S uses evidence that the variable is in a context Γ to prove that it is in Γ extended with another variable

```
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 _∋_ data type is a variable or a covariable is decided entirely by whether the Context that the instance (I will call such an instance a var from now on) is indexed by is used as an antecedent or a succedent.

We now have all we need to define DC terms, coterms, and statements. With our intrinsically-typed representation these are all indexed by two Contexts (the antecedent and succedent) and terms and coterms are indexed by a Type. This means we don't really define terms, coterms and statements, we define left sequents, right sequents, and centre sequents, that cannot exist independently of their Type and Contexts. Note that the constructors of these sequents are essentially the typing relation of the DC. This makes it impossible to construct a sequent that is not well-typed, as intrinsic-typing requires.

```
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
```

you can even remove the line break here

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

3.2.2 Values and Covalues

give one or two cases as an example

We introduce both a Value and Covalue inductive family, indexed on right sequents and left sequents respectively. The Agda term Value M provides evidence that the DC term represented by M is a value. The definitions of these families is as expected, with the constructors matching the definitions of values and covalues from the original paper.

We also introduce two new inductive families, called TermValue and CotermValue they are simply products of a sequent and the proof that it is a Value or Covalue. These are helpful constructs which we will make use of throughout the implementation.

dependent

```
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
```

3.2.3 Renaming and Substitution

~~While the motivation for implementing substitution (we have (co)variable abstraction) should be clear,~~ the motivation for renaming may be less so and is worth expanding on. When we substitute into a (co)variable abstraction we must recursively substitute into the statement it abstracts on. To avoid variable capture, however, we must rename the variables we are mapping from, incrementing the de Bruijn indices to account for the newly bound variable. Renaming and substitution are both sequent traversals, they take an operation on variables and lift it to an operation on sequents. They differ only in what they map variables to, whereas renaming maps a variable to a variable, substitution maps variables to sequents [26].

The Dual Calculus is a language that involves variable binding, so evaluation of DC programs relies on the operation of substitution. While relatively easy to define on paper (taking care to only substitute for free variables and avoid variable capture), substitution is notoriously difficult to formalise in a proof assistant even for the simplest calculus like STLC. [refs] Indeed, dealing with the tedious nuances of substitution often makes up the bulk of a language formalisation task, and this was no different in the case of my project. Even though overcoming the difficulties required considerable effort – not in small part because existing approaches had to be adapted to work with a mutually inductively defined syntax with two distinct kinds of variables – the technical details of the solutions are rather involved and discussing them would detract from the main themes of this dissertation. Below I give a very high-level overview of how substitution is defined [and defer the details to the Appendix].

The only operations we require for the reduction relation is the substitution of a term or coterm into a statement.
[type]
Defining this operation from scratch is a futile exercise, as it cannot be recursively applied to the body of an abstraction due to the presence of a new variable. We must therefore derive it from the general simultaneous substitution operation sub-S
[def]
<provide a summary of what each part of the def is>

Since our intrinsically typed representation statically enforces both well-typedness and well-scopedness, we must implement both substitution and renaming in a type and scope safe manner [1]. My implementation is based on the example McBride lays out for the STLC in his 2005 paper [26], though with significant changes to account for the intricacies of the DC. McBride defines an abstract family indexed by a Type and Context, and shows how this family must support three operations: mapping in from variables, mapping out to terms, and a weakening map that extends the Context. We can group these functions together into a *kit*, parameterised by this inductive family. We can then use this for both renaming and substitution: in the case of renaming we instantiate the family with the data type for the variables of the STLC, and for substitution we instantiate it with the data type of terms. Keller [23] demonstrates how we can use Agda record types to represent these kits, with fields for each of the three required functions, the functions provided by these kits are used throughout the implementations of substitution and renaming.

However, things are more complex with the DC. Firstly, sequents are indexed by a Type and two Contexts, whereas vars are indexed by a Type and one Context, this means we cannot generalise vars and sequents to one family. Secondly, the DC has three different types of sequents and a var can represent *either* a variable *or* a covariable. This means that a kit parameterised by a generic T, that itself is indexed by two Contexts and a Type, would have to support mapping in from variables and covariables, and mapping out to both right sequents and left sequents. This is impossible, say we mapped a covariable into T, there would be no way to map this out to a right sequent, as we cannot construct a right sequent from a covariable. All this means we must define three different types of kits, one for vars, one for right sequents, and one for left sequents. The kit for right sequents is parameterised by a generic T, which can be mapped into by variables and mapped out to right sequents, while also supporting weakening in either context. The kit for the left sequents, parameterised by a generic C, does the same but with covariables and left sequents. The kit for

vars is parameterised by a generic T, indexed by a Context and a Type, this supports mapping into from a var and weakening the Context it is indexed by. We can actually use these var kits to support mapping into from vars and weakening of one of the Contexts for the left and right sequent kits, by including a var kit with a fixed context as a field of the sequent kits.

```
record VarKit (T : Context → Type → Set) : Set where
  field
    vr : ∀ {Γ A} → Γ ∋ A → T Γ A
    wk : ∀ {Γ A B} → T Γ A → T (Γ , B) A

record TermKit (T : Context → Context → Type → Set) : Set where
  field
    tm : ∀ {Γ Θ A} → T Γ Θ A → Γ ⟶ Θ ∣ A
    kit : ∀ {Θ}       → VarKit (Fix₂ T Θ)
    wkΘ : ∀ {Γ Θ A B} → T Γ Θ A → T Γ (Θ , B) A
  open module K {Θ} = VarKit (kit {Θ}) renaming (wk to wkΓ) public

record CotermKit (C : Context → Context → Type → Set) : Set where
  field
    tm : ∀ {Γ Θ A} → C Γ Θ A → A ∣ Γ ⟶ Θ
    wkΓ : ∀ {Γ Θ A B} → C Γ Θ A → C (Γ , B) Θ A
    kit : ∀ {Γ}       → VarKit (Fix₁ C Γ)
  open module K {Γ} = VarKit (kit {Γ}) renaming (wk to wkΘ) public
```

I will also introduce the idea of context maps. A context map represents the substitution or renaming itself, it is indexed by two Contexts and a Sorted-Family (a family indexed by a Type and a Context). A context map indexed by Sorted-Family T, and Contexts Γ and Δ, represents a

substitution that, for an arbitrary Type A, transforms elements of $\Gamma \ni A$ to $T\,\Delta\,A$. A renaming map is a special instance of a context map for which the Sorted-Family is instantiated to the datatype $\_\ni\_$.

```
_–[_]→_ : Context → Sorted-Family → Context → Set
Γ –[ X ]→ Δ = {A : Type} → Γ ∋ A → X Δ A

_⇝_ : Context → Context → Set
Γ ⇝ Δ = Γ –[ _∋_ ]→ Δ
```

Now we have defined these constructs we will explain the definition and derive the behaviour of the general substitution function in a top-down manner. Below is the type signature of this function and an example of how it is used in the syntax of the DC. The example given below is the substitution of a value into a statement, which arises in the CBV operational semantics of the DC, this is discussed in more detail in the next section. The function is mutually inductive so the type signature for substitution into left and right sequents is also given.

```
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′
  → 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′

_^V⟨_/⟩^s {Γ}{Θ} S V =
```

sub-S TVK CK (add (Fix$_2$ TermValue Θ) V id-TV) id-C S

The substitution function takes a TermKit T, a CotermKit C, two arbitrary substitutions, and a sequent. One substitution modifies the antecedent Context and produces instances of the Sorted-Family T, the other modifies the succedent and produces instances of C. The function returns a sequent with both the Contexts it is indexed by modified according to the subsitutions given.

First we explain how the substitution function is used in the above example, discussing renaming as we do this, before detailing how substitution is implemented.

First we define instances of the kits: CK and TVK. These instances instantiate the generic family each kit is indexed by to a left sequent and a TermValue respectively, as well as implementing the necessary functions. Mapping out is simple, since the generic parameter is instantiated to the required construct. Mapping in, defined in the VarKit field, also simply invokes the `_ constructor. The TVK implementation is slightly more complex in these cases as a TermValue contains both a sequent and a proof that it is a value, so we must construct/project out of this pair as necessary. The weakening of both Contexts, however, requires us to define renaming, this is used directly in the implementation of CK and indirectly, by way of TK (the dual of CK) in TVK.

We define the renaming of sequents by mutual induction. These type signatures are very similar to the type signatures for substitution, note the use of renaming maps rather than general context maps.

ren-T : $\forall\ \{\Gamma\ \Gamma'\ \Theta\ \Theta'\ A\} \to \Gamma \rightsquigarrow \Gamma' \to \Theta \rightsquigarrow \Theta' \to \Gamma \longrightarrow \Theta \mid A \to \Gamma' \longrightarrow \Theta' \mid A$

ren-C : $\forall\ \{\Gamma\ \Gamma'\ \Theta\ \Theta'\ A\} \to \Gamma \rightsquigarrow \Gamma' \to \Theta \rightsquigarrow \Theta' \to A \mid \Gamma \longrightarrow \Theta \to A \mid \Gamma' \longrightarrow \Theta'$

ren-S : $\forall\ \{\Gamma\ \Gamma'\ \Theta\ \Theta'\} \to \Gamma \rightsquigarrow \Gamma' \to \Theta \rightsquigarrow \Theta' \to \Gamma \longmapsto \Theta \to \Gamma' \longmapsto \Theta'$

Most of the cases of this renaming function are simple induction, the interesting cases are vars and (co)variable abstraction. In the var cases we use the `_ to construct a sequent from the application

of the relevant renaming to the var.

```
ren-T ρ ϱ (` x) = ` (ρ x)
```

```
ren-C ρ ϱ (` α) = ` (ϱ α)
```

For the variable abstraction case we must *lift* the antecedent Context renaming from $\Gamma \rightsquigarrow \Gamma'$ to $\Gamma\ ,\ A \rightsquigarrow \Gamma'\ ,\ A$, this is because the variable abstraction results in a new variable being bound in S. Covariable abstraction requires the same, except we must lift the succedent Context instead. This *lift* operation is provided by a function that extends both contexts in a renaming map: ren-lift.

```
ren-T ρ ϱ (μθ S) = μθ (ren-S ρ (ren-lift ϱ) S)
```

```
ren-C ρ ϱ (μγ S) = μγ (ren-S (ren-lift ρ) ϱ S)
```

The last requirement is to define the weakening of a renaming, in which the Context the renaming is mapping into is extended. The type signature of the function that implements this is given below.

```
ren-weaken : ∀ {Γ Δ A} → Γ ⇝ Δ → Γ ⇝ (Δ , A)
```

This gives us all we need to define CK, note that id-var simply represents the identity renaming, and is just a synonym for the identity function.

```
CK : CotermKit λ Γ Θ A → A | Γ ⟶ Θ
CK = record
  { tm = λ a → a
  ; wkΓ = ren-C (ren-weaken id-var) id-var
  ; kit = record { vr = `_ ; wk = ren-C id-var (ren-weaken id-var) }
  }
```

For TVK we must also implement V-rename, a proof that the arbitrary renaming of a Value is still a Value.

```
TVK : TermKit TermValue
TVK = record
  { tm = λ x → proj₁ x
  ; wkΘ = λ x → ⟨ (TermKit.wkΘ TK (proj₁ x))
                , V-ren (proj₂ x) ⟩
  ; kit = record
    { vr = λ x → ⟨ ` x , V-var ⟩
    ; wk = λ x → ⟨ (VarKit.wk (TermKit.kit TK) (proj₁ x))
                , V-ren (proj₂ x) ⟩ }
  }
```

The other definitions used in the subsitution example are add, id-termvalue, and id-coterm. add is a simple function that adds a term to a context map, while id-termvalue and id-coterm simply represent the identity termvalue and coterm substitutions respectively.

We now discuss the implementation of substitution, as a reminder, this is the type signature of the mutually inductive substitution function.

```
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′
  → 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′
```

As is the case with renaming, the interesting cases are (co)variables and (co)variable abstraction; the other cases are simple induction. For variables we apply the TermKit instance's mapping of T to a right sequent to the application of the antecedent substitution on the variable, the covariable case is dual.

```
sub-T k₁ k₂ s t (` x) = TermKit.tm k₁ (s x)
```

```
sub-C k₁ k₂ s t (` α) = CotermKit.tm k₂ (t α)
```

The variable abstraction case is a slightly more complex version of the same case for renaming, we must lift the substitution to account for the newly bound variable. However, unlike renaming, we also must modify the succedent substitution such that the Sorted-Family C that it produces instances of is indexed by Γ' , A instead of Γ', this is done with the fmap function. The covariable abstraction case is the dual of this, lifting the succedent substitution, and applying fmap to antecedent substitution.

```
sub-T {T}{A}{C}{Γ}{Θ}{Γ′}{Θ′} k₁ k₂ s t (μθ S) = μθ (sub-S k₁ k₂
  (fmap {λ - → T - Θ′} {λ - → T - (Θ′ , A)} (λ x → TermKit.wkΘ k₁ x) s)
  (sub-lift (CotermKit.kit k₂) t) S)
```

```
sub-C {T}{A}{C}{Γ}{Θ}{Γ′}{Θ′} k₁ k₂ s t (μγ S) = μγ (sub-S k₁ k₂
  (sub-lift (TermKit.kit k₁) s)
  (fmap {C Γ′} {C (Γ′ , A)} (λ x → CotermKit.wkΓ k₂ x) t) S)
```

The lifting operation is implemented by the function sub-lift, which makes use of a VarSubstKit paramterised by a Sorted-Family T. Its implementation requires us to define weakening as well, this

sub-weaken function simply applies the VarSubstKit's implementation of weakening to the substituted var. The fmap function takes a family of functions from $T\,\Gamma\ A$ to $T'\,\Gamma\ A$ and a substitution from Γ to Γ' that produces instances of T. This returns a substitution that produces instances of T'.

$$\mathsf{fmap} : \forall\ \{T\ T'\ \Gamma\ \Gamma'\}\ (f : \forall\ \{\Gamma\ A\} \to T\,\Gamma\ A \to T'\ \Gamma\ A) \to \Gamma \mathrel{-[}\ T\ \mathrel{]\!\to} \Gamma' \to \Gamma \mathrel{-[}\ T'\ \mathrel{]\!\to} \Gamma'$$

$$\mathsf{fmap}\ f\ \sigma\ \ `x = f\ (\sigma\ \ `x)$$

With this final definition we have produced a definition of substitution that preserves type and scope-safety.

3.2.4 Implication

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

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

$$\underline{\lambda}^{V}\ N = \mathsf{not}[\ \mu\gamma(\gamma\ 0 \bullet \mathsf{fst}[\ \mu\gamma\ (\gamma\ 1 \bullet \mathsf{snd}[\ \mathsf{not}\langle\ \mathsf{int}\Gamma^{t}\ (\mathsf{wk}\Gamma^{t}\ N)\ \rangle\ ])\ ])\ ]$$

gib type signatur pls

$$\underline{\lambda}^{N}\ N = \mu\theta\ (\mathsf{inl}\langle\ \mathsf{not}[\ \mu\gamma(\mathsf{inr}\langle\ \mathsf{wk}\Theta^{t}\ N\ \rangle \bullet \theta\ 0)\ ]\ \rangle \bullet \theta\ 0)$$

$$M\ \cdot^{V}\ N = \mathsf{not}\langle\ `\langle\ M\ ,\ \mathsf{not}[\ N\ ]\ \rangle\ \rangle$$

$$M\ \cdot^{N}\ N = `[\ \mathsf{not}\langle\ M\ \rangle\ ,\ N\ ]$$

3.3 Operational Semantics

This section defines the operational semantics of the DC, giving its reduction relations.

3.3.1 Reduction Relations

The DC has two reduction relations, one Call-by-Value, the other Call-by-Name. 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, in fact, when they are omitted the Calculus is strongly normalising. Therefore, I chose to omit these rules from the formalised operational semantics, defining only the congruence and reduction rules. This means that every reduction rule takes a statement to a statement, thus we require two inductive families, one for the Call-by-Value reduction relation, and one for Call-by-Name. Each of these inductive families is indexed by two centre sequents, the sequent both before and after the rule is applied, I give the type signature of the Call-by-Value relation as an example.

if you choose to discuss this, better to keep it as a remark somewhere at the end (rather than the second sentence!)

```
data _ ⟶ˢ^V _ : ∀ {Γ Θ} → (Γ ⟼ Θ) → (Γ ⟼ Θ) → Set where
```

perhaps you can trick latex into displaying the superscript V as a superscript v so it's ~ nicer ~

Note the type of both centre sequents is the same, this means that any constructor of this inductive family must preserve types. Therefore, thanks to intrinsic typing, the definition of the reduction relation doubles up as a proof of the Type Preservation theorem.

there are no types so it's not applicable. Still might be worth mentioning that "For standard calculi like STLC, the intrinsically-typed reduction relation is defined between terms of the same type and context and therefore doubles as a proof of the Type Preservation theorem. Since statements in DC do not have types, we only get a – less interesting, but still valuable – "context-preservation" theorem.

The constructors of this inductive family correspond directly to the reduction rules laid out in the original paper. Here is the $\beta\times_1$ constructor as an example. Note how it also requires evidence that both terms V and W are *Value*s.

i should be blue da ba dee da ba di

```
β×₁ : ∀ {Γ Θ A B} {V : Γ ⟶ Θ | A} {W : Γ ⟶ Θ | B} {K : A | Γ ⟶ Θ}
  → Value V → Value W
    ---------------
  → ‘⟨ V , W ⟩ • fst[ K ] ⟶ˢ^V V • K
```

The most interesting examples of this relation are the reduction rules: βL and βR. These are given below.

$\beta\mathsf{L} : \forall \{\Gamma\ \Theta\ A\}\ \{V : \Gamma \longrightarrow \Theta \mid A\}\ \{S : \Gamma\ ,\ A \longmapsto \Theta\}\ (v : \mathsf{Value}\ V)$

$\rightarrow V \bullet (\mu\gamma\ S) \overset{s}{\longrightarrow}^{V} S \overset{V}{\langle}\ \langle\ V\ ,\ v\ \rangle\ /\rangle^{s}$

$\beta\mathsf{R} : \forall \{\Gamma\ \Theta\ A\}\ \{K : A \mid \Gamma \longrightarrow \Theta\}\ \{S : \Gamma \longmapsto \Theta\ ,\ A\}$

$\rightarrow (\mu\theta\ S) \bullet K \overset{s}{\longrightarrow}^{V} S\ [\ K\ /]^{s}$

Both rules include a substitution, substituting a left sequent in the case of $\beta\mathsf{R}$ and a TermValue in the case of $\beta\mathsf{L}$. ~~These operations are shorthand for the general substitution functions defined in the previous section.~~

~~$\_\overset{V}{\langle}\_/\rangle^{s} : \forall \{\Gamma\ \Theta\ A\}$~~

~~$\rightarrow \Gamma\ ,\ A \longmapsto \Theta$~~

~~$\rightarrow \mathsf{TermValue}\ \Gamma\ \Theta\ A$~~

~~---~~

~~$\rightarrow \Gamma \longmapsto \Theta$~~

will be defined in the prev section

~~$\_\overset{V}{\langle}\_/\rangle^{s}\ \{\Gamma\}\{\Theta\}\ S\ V =$~~

~~$\mathsf{sub\text{-}S\text{-}TVK\text{-}CK}\ (\mathsf{add}\ (\mathsf{Fix}_2\ \mathsf{TermValue}\ \Theta)\ V\ \mathsf{id\text{-}TV})\ \mathsf{id\text{-}C}\ S$~~

~~$\_[\_/]^{s} : \forall \{\Gamma\ \Theta\ A\}$~~

~~$\rightarrow \Gamma \longmapsto \Theta\ ,\ A$~~

~~$\rightarrow A \mid \Gamma \longrightarrow \Theta$~~

~~---~~

~~$\rightarrow \Gamma \longmapsto \Theta$~~

~~_[_/]^s {Γ}{Ø} S K =~~

~~sub-S TVK-CK id-TV (add (Fix$_1$ Coterm Γ) K id-C) S~~

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 the 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. 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
```

align things nicely

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 defined in the original paper maps from the DC to the STLC, if I were to implement this it would require a formalisation of the STLC as well, which is a serious job in its own right. To keep focus on the DC we decided to use Agda as the *target calculus*, this is valid as Agda has all the constructs of the version of the STLC that Wadler uses as his target calculus.

I will present only the implementation of the Call-by-Value CPS transformation as the Call-by-Name CPS transformation is dual.

As we are implementing a denotational semantics of the DC it is worth summarising the standard results in this field and how these must be modified for the DC. In a standard denotational semantics of a language, types are interpreted as sets and contexts as products of these sets. The denotation of a term is a function from the denotation of contexts to the denotation of types, this is because we need to interpret the free variables within a term to be able interpret to it. After defining these standard interpretations, one would prove the semantic substitution lemma, which essentially states that the interpretation of a substitution, $M'[M/x]$, is equivalent to the of the interpretation of M' applied to the interpretation M. With this proved, one would then prove the soundness of the denotational semantics, demonstrating that if one term evaluates to another in the languages operational semantics, then the terms have the same denotation [18]. With this in mind, we present the denotational semantics of the DC.

methodology

how it can be applied to the DC

what things get interpreted as depends on the syntax (e.g. PCF types are interpreted as domains, not sets)

see below for some ideas on how to summarise densem

Again we start with defining the operation for Type and Context, here we can implement the denotational semantics in a completely standard way. Types are interpreted as Sets, and Contexts as products of these Sets, ~~the implementation is as defined in the original paper.~~

they won't have read the paper and the definitions are not spelled out in the preparation so worth expanding upon a bit (esp. negation)

$\_^{VT}$: Type → Set

$\_^{Vx}$: Context → Set

A set-based model of a simple language like STLC interprets types A as sets ⟦A⟧, and typing contexts Γ = x1 : A1, ..., xn : An as products of the interpretations of the variable types ⟦Γ⟧ = ⟦A1⟧ × ⋯ × ⟦An⟧. A term Γ ⊢ M : A is interpreted as a function from ⟦Γ⟧ → ⟦A⟧ that maps the interpretation of free variables to an element of the denotation of the type. In particular, if M is closed, ⟦M⟧ : {*} → ⟦A⟧ is isomorphic to an element of ⟦A⟧. This general approach can be adapted to more involved languages, which often require one to assume more structure on the target of the interpretation: for example, the partial language PCF is interpreted in domains and continuous functions.

Once the meaning of terms is established, we may investigate properties such as soundness with respect to the operational semantics: does reduction preserve the meaning of programs? A central property required for this is the semantic substitution lemma that characterises the denotation of M' [M/x] as the composition of the denotations of M' and N.

Even though calling a CPS transformation "denotational semantics" may sound overly grandiose (Wadler doesn't use the term either), the definitions and theorems I stated fit well into the framework outlined above. With this in mind, I present the denotational semantics of the Dual Calculus in [the category of] sets and functions.

$$
\begin{aligned}
&\text{`}\mathbb{N}\ ^{VT} &&= \mathbb{N} \\
&(A\ \text{`}\times\ B)\ ^{VT} &&= (A\ ^{VT}) \times (B\ ^{VT}) \\
&(A\ \text{`}+\ B)\ ^{VT} &&= (A\ ^{VT}) \uplus (B\ ^{VT}) \\
&(\text{`}\neg\ A)\ ^{VT} &&= (A\ ^{VT}) \to R \\
&\varnothing\ ^{Vx} &&= \top \\
&(\Gamma\ ,\ A)\ ^{Vx} &&= \Gamma\ ^{Vx} \times A\ ^{VT}
\end{aligned}
$$

remark that the base type could be interpreted as any set, and $\mathbb{N}$ was chosen to make evaluation easier

Now we define the interpretation of vars. A var is simply a pointer to a Type in a Context as such the interpretation of a var projects the relevant interpreted Type from the interpreted Context.

$$
\begin{aligned}
&\_^{VV} : \forall\ \{\Gamma\ A\} \to (\Gamma \ni A) \to ((\Gamma\ ^{Vx}) \to (A\ ^{VT})) \\
&\_^{VV}\ \text{`Z} = \lambda\ c \to \mathsf{proj}_2\ c \\
&\_^{VV}\ (\text{`S}\ x) = \lambda\ c \to ((x\ ^{VV})\ (\mathsf{proj}_1\ c))
\end{aligned}
$$

pattern-match instead of project

Sequents provide the first interesting difference in the denotational semantics of the dual calculus. Each DC sequent is indexed by two Contexts, therefore the interpretation of a sequent must instead be a function from a *pair* of interpreted Contexts. The type signatures of the interpretation of sequents is given below, they closely resemble Proposition 6.1 from the original paper, which asserts that the CPS transformation preserves types. Note that $\text{`}\neg^x$ 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 these transformations produce. Note that R is an arbitrary type representing the rest of the computation. TermValues are interpreted

as programs of type $A\ ^{VT}$, this is because when we obtain a value the computation has completed, as such it must be fed into a continuation. Continuations are of type $A\ ^{VT} \to R$, and are how we interpret left sequents. This is because right sequents contain coterms, and coterms consume values. Right sequents are transformed into programs of type $(A\ ^{VT} \to R) \to R$, this is the standard type of a CPS-transformed term, taking a continuation, and then returning control to the rest of the computation, represented by R. Statements are interpreted simply as programs of type R, this is because they do not actually execute anything so are of whatever type the rest of the program is.

The transformation itself is then implemented in a way that closely mirrors the definition given in the original paper, the key difference is that we abstract on the pair of transformed Contexts and must, in some cases, modify this pair before recursing. We do this in the variable abstraction cases by adding the newly bound variable to the transformed antecedent before making the recursive call. We do the same for covariable abstractions but with the succedent instead. Note also the inclusion of $\Gamma \ni A \Rightarrow \neg\Gamma \ni \neg A$ in the covariable case, this is just a lemma allowing us to derive $(`\neg^x\ \Gamma) \ni (`\neg\ A)$ from $\Gamma \ni A$. We must include this because the covariable α has type $\Theta \ni A$, which is interpreted as a function from $\Theta\ ^{Vx}$ to $A\ ^{VT}$, while we require a function from $`\neg^x\ \Theta$ to $(`\neg\ A)\ ^{VT}$. I omit the other cases as other than the inclusion of abstracting on the transformed Context they essentially match the definition from the original paper.

not worth mentioning

rename to something nicer

I still think you could include the other cases, maybe split into two columns if you need space

$$(\langle\ `\ x\ ,\ \text{V-var}\ \rangle\ ^{VLV})\ \langle\ \gamma\ ,\ \theta\ \rangle = (x\ ^{VV})\ \gamma$$

$$((`\ x)\ ^{VL})\ \langle\ \gamma\ ,\ \theta\ \rangle = \lambda\ k \to k\ ((x\ ^{VV})\ \gamma)$$

$$((`\ \alpha)\ ^{VR})\ \langle\ \gamma\ ,\ \theta\ \rangle = \lambda\ z \to ((\Gamma \ni A \Rightarrow \neg\Gamma \ni \neg A\ \alpha)\ ^{VV})\ \theta\ z$$

$$((\mu\theta\ S)\ ^{VL})\ \langle\ \gamma\ ,\ \theta\ \rangle = \lambda\ \alpha \to (S\ ^{Vs})\ \langle\ \gamma\ ,\ \langle\ \theta\ ,\ \alpha\ \rangle\ \rangle$$

$$((\mu\gamma\ S)\ ^{VR})\ \langle\ \gamma\ ,\ \theta\ \rangle = \lambda\ x \to (S\ ^{Vs})\ \langle\ \langle\ \gamma\ ,\ x\ \rangle\ ,\ \theta\ \rangle$$

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

```
cps-V : ∀ {Γ Θ A} (V : Γ ⟶ Θ ∣ A) (v : Value V) (c : Γ ⁽ Vx ⁾ × (‘¬ x Θ) ⁽ Vx ⁾)
  → (V ⁽ VL ⁾) c ≡ λ x → x (((⟨ V , v ⟩ ⁽ VLV ⁾) c)
```

apply pointwise to a continuation

3.4.2 CPS Transformation of Renamings and Substitutions

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

only discuss this if the notions are explained in the syntax section (which they probably will be but maybe not)

Consider M of type $\Gamma \rightarrowtriangle \Theta \mid A$, and two renamings ρ and ϱ of types $\Gamma \rightsquigarrow \Gamma'$ and $\Theta \rightsquigarrow \Theta'$ respectively. The interpretation of renaming M by ρ and ϱ must be a function from the interpretations of Γ' and Θ' to the interpretation of A. To create this function by composition of the interpretation of M and the interpretations of ρ and ϱ, the renamings must be interpreted as functions from Γ' to Γ and Θ' to Θ respectively. The type signature of the Call-by-Value interpretation of a renaming is given below. We define this by building up a product of interpreted types by interpreting the renaming of each var that references Γ' and applying that to the interpretation of Γ'.

```
ren-int-cbv : ∀ Γ Γ' → Γ ⇝ Γ' → (Γ' ⁽ Vx ⁾) → (Γ ⁽ Vx ⁾)
```

```
ren-int-cbv ∅ Γ' ρ γ = tt
ren-int-cbv (Γ , A) Γ' ρ γ =
  ⟨ (ren-int-cbv Γ Γ' (λ z → ρ (‘S z)) γ) , (((ρ ‘Z) ⁽ VV ⁾) γ) ⟩
```

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

neg-ren-int-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 substitutions, once again we interpret the substitution of every var that references Γ'. However, since the substitution produces a sequent that is indexed by two Contexts, its interpretation must be applied to a *pair* of interpreted Contexts. As such, the interpretation of a substitution is a function from the interpretations of *both* of the indexing Contexts of the sequents that the substitution produces.

sub-TV-int : $\forall\ \Gamma\ \Gamma'\ \Theta \to \Gamma \mathrel{-[}\ (\mathsf{Fix}_2\ \mathsf{TermValue}\ \Theta)\ \mathrel{]\to} \Gamma'$
$\to ((\text{`}\neg^{x}\ \Theta)^{Vx}) \to (\Gamma'^{Vx}) \to (\Gamma^{Vx})$

sub-C-int : $\forall\ \Gamma\ \Theta\ \Theta' \to \Theta \mathrel{-[}\ (\mathsf{Fix}_1\ \mathsf{Coterm}\ \Gamma)\ \mathrel{]\to} \Theta'$
$\to \Gamma^{Vx} \to ((\text{`}\neg^{x}\ \Theta')^{Vx}) \to ((\text{`}\neg^{x}\ \Theta)^{Vx})$

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 that is made up of the interpreted renamings/substitutions applied to the interpreted Contexts.

Just like the definition of the operations, these proofs are ...

These proofs are extremely involved and I do not wish to overwhelm the reader with detail, as such I present the type signature of the substitution lemma and explain how it is proved. The renaming lemma follows the exact same pattern.

throw the detailed explanation in an appendix??

sub-lemma-T : $\forall$ {Γ Γ' Θ Θ' A}

(s : Γ –[(Fix$_2$ TermValue Θ')]→ Γ') (t : Θ –[(Fix$_1$ 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 : $\forall$ {Γ Γ' Θ Θ' A}

(s : Γ –[(Fix$_2$ TermValue Θ')]→ Γ') (t : Θ –[(Fix$_1$ 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 : $\forall$ {Γ Γ' Θ Θ'}

(s : Γ –[(Fix$_2$ TermValue Θ')]→ Γ') (t : Θ –[(Fix$_1$ 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 directly 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 [1] in terms that do not include sub-weaken,

[1] 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.

sub-lift, or fmap. This must be done for both TermValue and left sequent substitutions. Since the behaviour of these functions is given by renaming, it is these lemmas that require us to prove the renaming lemma. As an example, I give the type signature of the weakening lemma for TermValues below.

3.4.4 Proof of Soundness

The preceding proofs allow us to go about proving the soundness of the Call-by-Value CPS transformation. In the original paper, Wadler asserts that the CPS transformations preserve reductions. from DC to the target calculus.

In our setting — where the target calculus is Agda itself — the natural statement of the property is that the CPS translation of terms related by many-step reduction are propositionally equal Agda terms.

I have taken a slightly different approach to defining the CPS transformation. Rather then embedding the DC into the STLC, I have defined a denotational semantics of the DC. This means that the notion of preserving reductions from the original paper does not make sense in the context of my CPS transformation. As such, I instead prove a composite proposition: if a sequent reduces to another sequent after an arbitrary number of reductions, then the Agda programs that they are interpreted as are equivalent.

well it's not that it does not make sense, it's just that the reduction in the target calculus is normalisation of Agda terms and definitional equality is β-convertibility

```
S—↠ᵛ T⇒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⟶ᵛ T⇒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 can prove this lemma by induction on the reduction relation _ ˢ⟶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 the most 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 simply reverses the direction of the equality to be proved.

```
S⟶^V T⇒S^V≡T^V {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                                                    squish squish
      (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 ⟩
  ≡⟨ cong (λ - → (S ^Vs) ⟨ ⟨ sub-TV-int Γ Γ Θ id-TV θ γ , (⟨ V , v ⟩ ^VLV) ⟨ γ , θ ⟩ ⟩ , - ⟩)
    (id-sub-C Γ Θ γ θ) ⟩                                  see if Agda accepts an _ here (it should do, since it's just tuple components)
    (S ^Vs) ⟨ ⟨ sub-TV-int Γ Γ Θ id-TV θ γ , (⟨ V , v ⟩ ^VLV) ⟨ γ , θ ⟩ ⟩ , θ ⟩
  ≡⟨ cong (λ - → (S ^Vs) ⟨ ⟨ - , (⟨ V , v ⟩ ^VLV) ⟨ γ , θ ⟩ ⟩ , θ ⟩) (id-sub-TV Γ Θ γ θ) ⟩
    (S ^Vs) ⟨ ⟨ γ , (⟨ V , v ⟩ ^VLV) ⟨ γ , θ ⟩ ⟩ , θ ⟩
  ≡⟨ sym (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 ⟩)
  □)
```

maybe combine these two steps into one with cong₂, since they do similar things? The cong locations then are ⟨ ⟨ −₁ , _ ⟩ −₂ ⟩

can use the symmetric equational step here, spelled as: \==\u{}\<

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 substitution lemma, ~~the result of this step should not be a surprise.~~

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

The next two steps of the proofs use two lemmas, they state that the interpretation of identity substitutions is the identity function. ~~Their types are given below though the proofs are omitted.~~

This leaves us with a formulation that looks much like our desired answer, in fact the final step is to appeal to the cps-V theorem and the proof is complete.

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, as such, we omit it.

With the soundness of the CPS transformation over one reduction step proved, we have the final tool we need to prove the soundness of the CPS transformation. As a reminder, the type signature of this theorem is given below.

$$\mathsf{S}\longrightarrow\!\!\!\!\!\rightarrow^{V}\mathsf{T}\Rightarrow\mathsf{S}^{V}\equiv\mathsf{T}^{V} : \forall\ \{\Gamma\ \Theta\}\ (S\ T : \Gamma \longmapsto \Theta)\ (c : (\Gamma\ ^{ox})\ ^{Nx} \times {}^{\backprime}\neg^{x}\ (\Theta\ ^{ox})\ ^{Nx})$$
$$\rightarrow S\ ^{s}\!\!\longrightarrow\!\!\!\!\!\rightarrow^{V}\ T \rightarrow (S\ ^{Vs})\ c \equiv (T\ ^{Vs})\ c$$

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

$$\mathsf{S}\longrightarrow\!\!\!\!\!\rightarrow^{V}\mathsf{T}\Rightarrow\mathsf{S}^{V}\equiv\mathsf{T}^{V}\ \ S\ .S\ c\ (.S\ \Box^{sV}) = \mathsf{refl}$$
$$\mathsf{S}\longrightarrow\!\!\!\!\!\rightarrow^{V}\mathsf{T}\Rightarrow\mathsf{S}^{V}\equiv\mathsf{T}^{V}\ \ S\ T\ c\ (\ \_^{s}\!\longrightarrow^{V}\langle\_\rangle\_\ .S\ \{S'\}\ .\{T\}\ S\longrightarrow S'\ S'\!\!\rightarrow\!\!\!\!\!\rightarrow T) =$$
$$\mathsf{trans}\ (\mathsf{S}\longrightarrow^{V}\mathsf{T}\Rightarrow\mathsf{S}^{V}\equiv\mathsf{T}^{V}\ \ S\ S'\ c\ S\longrightarrow S')\ (\mathsf{S}\longrightarrow\!\!\!\!\!\rightarrow^{V}\mathsf{T}\Rightarrow\mathsf{S}^{V}\equiv\mathsf{T}^{V}\ \ S'\ T\ c\ S'\!\!\rightarrow\!\!\!\!\!\rightarrow T)$$

This completes our proof of the soundness of the Call-by-Value CPS transformation, note that the Call-by-Name CPS transformation requires a separate proof with separate renaming and substitution lemmas. This proof, however, is the dual of the proof I have outlined in the last two sections so is not included.

3.5 Duality

This section discusses the duality of the DC, we first define the dual translation – demonstrating the duality of the syntax of the calculus – followed by the duality of the operational and denotational semantics.

3.5.1 The Duality of Syntax

The dual translation defined in the original paper is actually five different operations, a separate translation for each of the following: 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 as two mutually recursive functions, ~~their implementation's closely match the definitions given in the original paper.~~

"As expected, products and sums are duals of each other, and negation (and the base types) are self-dual."

$$\_^{oT} : \mathsf{Type} \to \mathsf{Type}$$

$$\_^{ox} : \mathsf{Context} \to \mathsf{Context}$$

$$(A \; `+ \; B)^{oT} = (A^{oT} \; `\times \; B^{oT})$$

$$(A \; `\times \; B)^{oT} = (A^{oT} \; `+ \; B^{oT})$$

don't need the parentheses after the = (I think)

$$(`\neg \; A)^{oT} = (`\neg \; (A)^{oT})$$

$$(`\mathbb{N})^{oT} = `\mathbb{N}$$

$$(\varnothing\ ^{ox}) = \varnothing$$

$$(\Gamma\ ,\ A)\ ^{ox} = ((\Gamma\ ^{ox})\ ,\ (A\ ^{oT}))$$

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, the dual translation of a var maps from variables to covariables simply by virtue of the Contexts 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.

$$\_^{oV} : \forall\ \{\Gamma\ A\} \to (\Gamma \ni A) \to (\Gamma\ ^{ox} \ni A\ ^{oT})$$

Now we can define the dual translation of sequents, with separate functions for left, right, and centre sequents. These are all exactly as the original paper would suggest.

again the readers will more appreciate some inline explanations (which, in this case, do not need to be very elaborate – it's just dual, bruh) than referring to a paper they haven't and won't read

$$\_^{os} : \forall\ \{\Gamma\ \Theta\} \to (\Gamma \longmapsto \Theta) \to (\Theta\ ^{ox} \longmapsto \Gamma\ ^{ox})$$

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

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

$$(`\ x)^{oL} = `\ x\ ^{oV}$$

$$(`\langle\ M\ ,\ N\ \rangle)\ ^{oL} = `[\ M\ ^{oL}\ ,\ N\ ^{oL}\ ]$$

$$(\mathsf{inl}\langle\ M\ \rangle)\ ^{oL} = \mathsf{fst}[\ M\ ^{oL}\ ]$$

$$(\mathsf{inr}\langle\ M\ \rangle)\ ^{oL} = \mathsf{snd}[\ M\ ^{oL}\ ]$$

$$(\mathsf{not}[\ K\ ])\ ^{oL} = \mathsf{not}\langle\ K\ ^{oR}\ \rangle$$

$$(\mu\theta\ \{\Gamma\}\ \{\Theta\}\ \{A\}\ (S))\ ^{oL} = \mu\gamma(\ \_^{os}\ \{\Gamma\}\ \{(\Theta\ ,\ A)\}\ S\ )$$

I think this is where some photoshopping might be beneficial to make it less ugly (i.e. manually editing the exported agda code)

$$(`\ \alpha)\ ^{oR} = `\ \alpha\ ^{oV}$$

$$(`[\ K\ ,\ L\ ])\ ^{oR} = `\langle\ K\ ^{oR}\ ,\ L\ ^{oR}\ \rangle$$

$$(\mathsf{fst}[\ K\ ])\ ^{oR} = \mathsf{inl}\langle\ K\ ^{oR}\ \rangle$$

$(\mathsf{snd}[\ K\ ])^{oR} = \mathsf{inr}\langle\ K^{oR}\ \rangle$

$(\mathsf{not}\langle\ M\ \rangle)^{oR} = \mathsf{not}[\ M^{oL}\ ]$

$(\mu\gamma\ \{\Gamma\}\ \{\Theta\}\ \{A\}\ (S))^{oR} = \mu\theta(\ \_^{os}\ \{(\Gamma\ ,\ A)\}\ \{\Theta\}\ (S)\ )$

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

We must also provide a definition for the dual translation of renaming and substitutions, despite this not being defined in the original paper. This is required to prove the duality of the operational semantics. These definitions are just simple induction, though I give their types below.

$\mathsf{dual\text{-}ren} : \forall\ \Gamma\ \Gamma' \to \Gamma \rightsquigarrow \Gamma' \to (\Gamma^{ox}) \rightsquigarrow (\Gamma'^{ox})$ is it possible to make the contexts here implicit? Or at least some of them

$\mathsf{dual\text{-}sub\text{-}C} : \forall\ \Gamma\ \Theta\ \Theta' \to \Theta \mathsf{\ -[(Fix_1\ Coterm\ \Gamma)]\to\ } \Theta' \to (\Theta^{ox}) \mathsf{\ -[\ (Fix_2\ Term\ (\Gamma^{ox}))\ ]\to\ } (\Theta'^{ox})$

$\mathsf{dual\text{-}sub\text{-}TV} : \forall\ \Gamma\ \Gamma'\ \Theta \to \Gamma \mathsf{\ -[(Fix_2\ TermValue\ \Theta)]\to\ } \Gamma' \to (\Gamma^{ox}) \mathsf{\ -[(Fix_1\ CotermValue\ (\Theta^{ox}))]\to\ } (\Gamma'^{ox})$

The definition of the dual translation is a great example of the power of intrinsic typing, as it is also a proof that if a sequent is derivable then its dual is derivable. Simply producing a sequent of a given type is proof that it is derivable, as our intrinsically typed represenation of syntax makes it impossible to produce an ill-typed sequent. So the fact that dual translation produces the dual of any sequent proves that any sequent's dual is derivable. That a sequent is derivable if its dual is derivable relies on the proof that the dual translation is an involution.

The mathematical proof of that the dual translation is an involution is very simple. The only issue is in translating it into a formal proof, explicitly spelling out every condition to satisfy Agda's type checker. Since we have defined six different dual translations (for Type, Context, var, Left Sequents, Right Sequents, and Centre Sequents) it should not be surprising that we have six different involution proofs.

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

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

This proof looks exactly like we would expect, however, Agda's type checker reports an error in the type signature of the proof. The error is as follows:

```
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, namely $(\Gamma\ ^{ox})\ ^{ox} \ni (A\ ^{oT})\ ^{oT}$ and $\Gamma \ni A$, and Agda will not accept this. This is despite the fact that we have proved that these two types are equal in our previous two proofs. 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*. This means that Agda will "automatically rewrite all instances of the left-hand side to the corresponding instance of the right-hand side during reduction" [40]. We use a REWRITE pragma just before the variable proof 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, as such, there is nothing logically sinful about this.

```
{-# REWRITE [A^oT]^oT≡A #-}
{-# REWRITE [Γ^ox]^ox≡Γ #-}
```

these

With this rewrite rule we can prove that the dual translation of sequents is also an involution. These are
This is a simple mutually inductive proof so I include only the type signatures.

```
[K^oR]^oL≡K : ∀ {Γ Θ A} (K : A | Γ ⟶ Θ) → (K ^oR) ^oL ≡ K
[M^oL]^oR≡M : ∀ {Γ Θ A} (M : Γ ⟶ Θ | A) → (M ^oL) ^oR ≡ M
[S^os]^os≡S : ∀ {Γ Θ} (S : Γ ⟼ Θ) → (S ^os) ^os ≡ 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 operational semantics of the DC. Specifically we seek to prove that a sequent S reduces by Call-by-Value to another sequent T iff the dual of S reduced by Call-by-Name to the dual of T. For brevity I demonstrate only one direction of this proof, the other direction is dual.

This proof is remarkably similar in structure to the proof of the soundness of the denotational semantics. This is because both proofs seek to prove that the reduction relation preserves a given property. In the case of soundness we demonstrated that it preserves meaning, here we seek to prove that it preserves duality. As such, 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.

As is the case with the proof of soundness, 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.

```
V^o≡P : ∀ {Γ Θ A} (V : Γ ⟶ Θ | A) → Value V → (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 a equivalent to the dual of the renamings/substitutions performed on the dual of the original sequent. I will outline the proof of the substitution lemma, note that it relies on the proof of the renaming lemma which I do not detail. The type signature of the substitution lemma is given below.

also might be nicer to put the implicit args under a variable declaration (which may let you combine the first two lines

```
dual-sub-lemma-T : ∀ {Γ Γ′ Θ Θ′ A} (M : Γ ⟶ Θ | A)
  (s : Γ –[ (Fix₂ TermValue Θ′) ]→ Γ′) (t : Θ –[ (Fix₁ 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)
```

put the expression on the LHS into parentheses so it's clear that the translation is applied to the whole thing

```
dual-sub-lemma-C : ∀ {Γ Γ′ Θ Θ′ A} (K : A | Γ ⟶ Θ)
  (s : Γ –[ (Fix₂ TermValue Θ′) ]→ Γ′) (t : Θ –[ (Fix₁ 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 –[ (Fix₂ TermValue Θ′) ]→ Γ′) (t : Θ –[ (Fix₁ 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, following a pattern that is familiar, we prove a corresponding lemma for both variables and covariables, and must prove various lemmas for each of the different operations on substitutions for the (co)variable abstraction cases. I give the types of these lemmas for TermValues below.

```
dual-sub-TV-weaken-lemma : ∀ Γ Γ′ Θ′ A {B} (σ : Γ –[ (Fix₂ TermValue Θ′) ]→ Γ′) (x : Γ ^ox ∋ B)
  → dual-sub-TV Γ (Γ′ , A) Θ′ (sub-weaken (TVK.kit) σ) x
    ≡ sub-weaken (CVK.kit) (dual-sub-TV Γ Γ′ Θ′ σ) x
```

```
dual-sub-TV-lift-lemma : ∀ Γ Γ′ Θ′ A {B} (σ : Γ –[ (Fix₂ 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-fmap-lemma : ∀ Γ Γ′ Θ′ A {B} (σ : Γ –[ (Fix₂ TermValue Θ′) ]→ Γ′) (x : Γ ᵒˣ ∋ B)
  → dual-sub-TV Γ Γ′ (Θ′ , A) (fmap-wkΘ^tV Θ′ A σ) x
    ≡ fmap-wkΓ^cV (Θ′ ᵒˣ) (A ᵒᵀ) (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

dual-sub-TV-add-lemma : ∀ Γ Θ A {B} (V : Term Γ Θ A) (v : Value V) (x : (Γ , A) ᵒˣ ∋ B)
  → dual-sub-TV (Γ , A) Γ Θ (add (Fix₂ TermValue Θ) ⟨ V , v ⟩ id-TV) x
    ≡ add (Fix₁ CotermValue (Θ ᵒˣ)) ⟨ V ᵒᴸ , Vᵒ≡P V v ⟩ id-CV x
```

Im not sure whether it worth listing them all here, what do u think

no, maybe show the id and lift ones

The interesting parts of these proofs is proving that the sequent that results from the substitution is a covalue. We can do this by proving that if two left sequents are equal, then a CotermValue derived from these left sequents must be equal.

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 pair as a pair of equalities. The first equality in the pair is simply 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.

dependent dependent

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

Note the use of subst in the type of CV-eq. This is because L-V is of type Covalue L so the left-hand side of the equality must be too. subst is a property of equality, if two values are equal and a predicate holds for the first then it also holds for the second [44]. 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.

show type of subst

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 proof that the operational semantics are dual, simply involves appealing to these lemmas, hence they are omitted.

at least state the type of one of the duality theorems

3.5.3 The Duality of Denotational Semantics

I now outline the proof of other 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.

We then add rewrite rules for these two equalities, for the same reason as in the proof that the dual translation is involutive; we need to demonstrate an equality been two constructs, one with type indexed by Γ^{Vx}, Θ^{Vx}, and A^{VT}, the other with type indexed by $\Gamma^{Nx\circ x}$, $\Theta^{Nx\circ x}$, and $A^{NT\circ x}$.

add parens

~~This proof also requires that we prove a couple of lemmas. The first lemma states that the dual of a negated Context is equivalent to the negation of the dual of a Context. The second lemma is similar, stating that the dual of negated var is equivalent to the negation of the dual of a var. The proofs of both are simple. We add the first lemma as a rewrite rule before proving the second lemma, this is required for the proof of the second lemma to be valid.~~

We now prove the duality of the transformations for vars. This proof is performed point-wise on

an arbitrary interpreted Context.

```
x^V≡x^oN : ∀ {Γ A} (x : Γ ∋ A) (c : Γ ^Vx) → (x ^VV) c ≡ ((x ^oV) ^NV) c

x^V≡x^oN ‘Z c = refl

x^V≡x^oN (‘S x) c = x^V≡x^oN x (proj₁ c)
```

Now follows the proof for sequents. ~~Each of the propositions are proved point-wise on the pair of transformed Contexts that are passed as an argument to the transformed sequent, this is simply to avoid the need to invoke ext on this argument for every single argument. The proof for left and right sequents are also performed point-wise on their continuations for exactly the same reason.~~

don't really need to elaborate on this, or if you do, mention it briefly at the first instance of the occurrence of this technique (probably somewhere in the densem section)

```
M^V≡M^oN : ∀ {Γ Θ A} (M : Γ ⟶ Θ | A) (c : Γ ^Vx × (‘¬^x Θ) ^Vx) (k : ((‘¬ A) ^VT))
  → (M ^VL) c k ≡ ((M ^oL) ^NR) c k
K^V≡K^oN : ∀ {Γ Θ A} (K : A | Γ ⟶ Θ) (c : Γ ^Vx × (‘¬^x Θ) ^Vx) (k : (A) ^VT)
  → (K ^VR) c k ≡ ((K ^oR) ^NL) c k
S^V≡S^oN : ∀ {Γ Θ} (S : Γ ⟼ Θ) (c : Γ ^Vx × (‘¬^x Θ) ^Vx)
  → (S ^Vs) c ≡ ((S ^os) ^Ns) c
```

The (co)variable cases of this proof simply appeal to duality of the var transformation, as well as the second of the two lemmas we proved above 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.

```
K^V≡K^oN (‘ α) ⟨ _ , c ⟩ k = trans
                               (cong (λ - → - k) (x^V≡x^oN (Γ∋A⇒¬Γ∋¬A α) c))
                               (cong (λ - → (- ^NV) c k) ([Γ∋A⇒¬Γ∋¬Ax]^o≡Γ∋A⇒¬Γ∋¬A[x^o] α))
```

and definitely rename these too

```
M^V≡M^oN (μθ S) ⟨ c₁ , c₂ ⟩ k = S^V≡S^oN S ⟨ c₁ , ⟨ c₂ , k ⟩ ⟩
```

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

end with a summary paragraph of the whole chapter

Chapter 4

Evaluation

4.1 Work Completed

several

As well as completing the core of my project, I was able to make time to complete ~~a few of the~~ extensions that I laid out in my project proposal.

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

4.1.1 Core

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

The first of these points was to learn Agda to the standard necessary to fulfil my other objectives, reading and working through the exercises in PLFA [44]. 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 [40] and my ever-helpful 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.

The final piece of work was to evaluate the project by measuring and comparing the execution times of various proofs of classical theorems after they had been CbV and CbN CPS transformed. While I did produce proofs of classical theorems, and measure the execution times of some Agda programs, we decided that using the DC formalisation as a base to simulate other calculi was a more informative evaluation. As such, we simulated three different calculi with the DC, showing that it could correctly represent simple arithmetic amongst other things.

Due to some limitations of the Dual Calculus itself (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 original project outlined three possible extensions, I completed two of these, taking one further than originally planned.

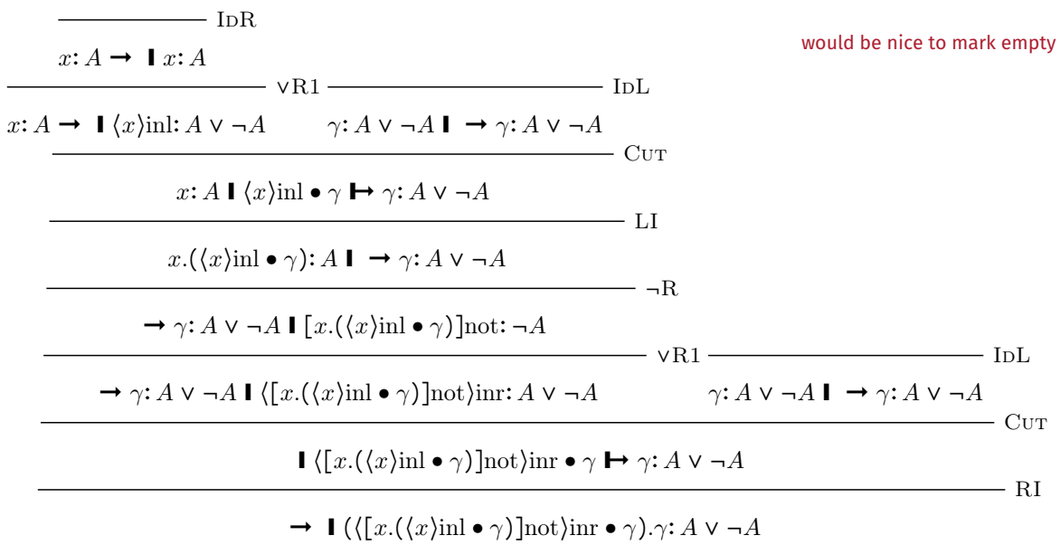
The first extension I completed was to define the operational semantics of the DC, requiring the implementation of renaming and substitution. I took this extension further by proving the duality of these operational semantics, as I believed this to be an important conclusion of the original paper.

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. However, these proofs required the formalisation to be complete to produce. To make sure that my formalisation was behaving correctly while it was in development, I used the in-progress formalisation to prove some simple lemmas and produce some results given in the original paper.

The first 'unit test' I produced was a proof of *the law of the excluded middle*, a characteristic



would be nice to mark empty contexts somehow, e.g. ∅

```
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.

also mention that γ and Θ is a shorthand for referring to variables by their numerical de Bruijn index

property of classical logic ~~systems~~, once I had completed the syntax of the formalisation. 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, for example, the dual translation of

sequents relying on the dual translation of Types and Contexts. I wanted to be confident that the 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 dual translation.

is there a code example for this?

not clunky enough to warrant a green underline on its own but you use the phrase three times

The original paper has some examples of the CPS transformations of DC terms. To give me confidence that the CPS transformation 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 the 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.

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

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

ex2^V : ∀ {A B} z → ((A ^VT → R) → R)
ex2^V {A}{B} z = (ex2 {A}{B} ^VL) ⟨ ⟨ tt , z ⟩ , tt ⟩

_ : ∀ {A B} z → (ex2^V {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 was 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	**Corrected**
$([\alpha]\text{not} \bullet \text{not}\langle x \rangle)^v$	$([\alpha]\text{not} \bullet \text{not}\langle x \rangle)^v$
$\equiv (\lambda\gamma.\gamma\,(\lambda z.(\lambda z.\alpha\, z)z))(\lambda z.(\lambda\gamma.(\lambda\gamma.x\,\gamma)\gamma)z)$	$\equiv (\lambda\gamma.\gamma\,(\lambda z.(\lambda z.\alpha\, z)z))(\lambda z.(\lambda\gamma.(\lambda\gamma.\gamma\, x)\gamma)z)$
$\equiv (\lambda\gamma.x\,\gamma)(\lambda z.\alpha\, z)$ this is then equal to x α	$\equiv \alpha\, x$ and add the intermediate step here as well

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 : $\forall\ \{A\} \to (M : \varnothing \longrightarrow \varnothing \mid A) \to \text{Value}\ M \to (K : A \mid \varnothing \longrightarrow \varnothing)$

$\to$ (lem-proof $\bullet$ lem-ref $M\ K$) ${}^{s}\!\!\longrightarrow\!\!\!\!\!\rightarrow^{V}$ $M \bullet K$

lem-comp $M\ M{:}V\ K =$ begin$^{s\,V}$

$\mu\theta$ (inr⟨ not[$\mu\gamma$ (inl⟨ γ 0 ⟩ $\bullet$ (θ 0))] ⟩ $\bullet$ (θ 0))

$\bullet$ ‘[K , not⟨ M ⟩] squish squish

${}^{s}\!\longrightarrow^{V}$⟨ βR ⟩

inr⟨ not[$\mu\gamma$ (inl⟨ γ 0 ⟩ $\bullet$ ‘[wkΓ^{c} K , not⟨ wkΓ^{t} M ⟩])] ⟩

$\bullet$ ‘[K , not⟨ M ⟩]

${}^{s}\!\longrightarrow^{V}$⟨ $\beta{+}_2$ V-not ⟩

not[$\mu\gamma$ (inl⟨ γ 0 ⟩ $\bullet$ ‘[wkΓ^{c} K , not⟨ wkΓ^{t} M ⟩])]

$\bullet$ not⟨ M ⟩

${}^{s}\!\longrightarrow^{V}$⟨ $\beta\neg$ ⟩

M

$\bullet$ $\mu\gamma$ (inl⟨ γ 0 ⟩ $\bullet$ ‘[wkΓ^{c} K , not⟨ wkΓ^{t} M ⟩])

${}^{s}\!\longrightarrow^{V}$⟨ βL $M{:}V$ ⟩

((inl⟨ γ 0 ⟩ $\bullet$ ‘[(wkΓ^{c} K) , not⟨ (wkΓ^{t} M) ⟩]) ${}^{V}\langle$ ⟨ 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 original substitution with the renaming. This does not mean that my operational semantics are 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 required above, on the behaviour of renamings and substitutions for a simply typed language alongside associated proofs in Coq. I decided that proving these lemmas for the DC was of lower priority than continuing with the extensions as I had planned. I made this decision because the soundness of the CPS transformation is a significantly more important, and more interesting, property than the behaviour of a substitution following a renaming. 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 system in the DC. This means that the constructs (types, terms etc.) 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, 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, and various standard results of said calculi.

Constructive logic may be seen as a "special case" of classical logic that does not rely on any classical axioms like LEM or DNE. Consequently, we expect that constructive computational calculi can be embedded into classical ones. Indeed, we can demonstrate that the Dual Calculus is a model of the simply-typed λ-calculus, that is, any λ-term can be compiled to a DC program.

4.3.1 Simply-Typed Lambda Calculus

calculus

The first calculi we simulated was the STLC. To do this we first must provide an interface for both the types and terms of the STLC. We do this using Agda's records, parameterised on abstractions that we will provide DC implementations for, defining fields for each of the constructors of that particular construct. As an example I present the record for types of the STLC below.

To do this, we axiomatise the type and term syntax of STLC using Agda records, listing the operations that an Agda object must support in order to be used as a λ-calculus type or term. For example, if a type T has a distinguished inhabitant B and a binary operator _=>_, it can be used to generate the infinite set of simple types from base type B.

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

you can add the derived definition of Church numerals here

Note that the record is paramaterised on some abstract T of type Set, this is the type of types for the STLC. The record has two fields, each representing a type constructor of the STLC. The first, B, represents the base type and the second, _⇒_ represents the function type.

The type declarations in a field block make up the interface that must be implemented when a record is instantiated. Definitions outside a 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.

add code

Within this record we also define church numerals, deriving a church numeral type (CN) with

Church Church

zero and successor constructors (z and s), alongside addition and multiplication functions (sum and mult). These definitions are all standard.

With interfaces for STLC types and terms prepared, we can use our DC formalisation to implement it. λ-Type is implemented by instantiating T to Type and providing implementations of the base type, B, and function type, $\_ \Rightarrow \_$. These are implemented using the DC's base type, $\mathbb{N}$, 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 implemented with the DC's variable constructor and derived Call-by-Value abstraction. Implementing 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 implement 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, a proof of A, to the covariable we abstracted on, a refutation of B.

```
DC-λ-Term : λ-Term Type DC-λ-Type (λ Γ A → Γ ⟶ ∅ | A)
DC-λ-Term = record {
  ‘ = ‘_ ;
  λ̲ = λ̲^V _ ;
  _·_ = λ M N → μθ (wkΘ^t M • wkΘ^t N ·^V θ 0)
  }
```

I think it's worth explaining the limitations of DC and how that meant that you weren't really able to do much quantitative evaluation. You can put the blame on DC and that extending it with new constructs while maintaining all the results you roved is very far from trivial. See some ideas below

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. While the DC includes the type ‘$\mathbb{N}$, there is no primitive to construct terms of this type. However, we can still

One limitation of the Dual Calculus is that it does not include much functionality to write "practical" programs: it assumes a base type X (to make the type syntax well-founded), but defines no inhabitants or operations on it. This is fairly common in the formal study of programming languages, where more emphasis is placed on interesting type operators and programming constructs than such mundane trivialities as numbers or Booleans. However, it becomes rather difficult to write interesting programs in a language that has no values we can operate on. Some practicality can be extracted from Church numerals, but they too are limited in a simply typed setting (no way to define predecessor, for example). This can often be remedied by simply extending the syntax with something like natural numbers with primitive recursion (also known as Gödel's System T). I tried to do the same for the Dual Calculus but encountered a not entirely unexpected problem: one does not simply add new terms to DC without also having to add dual coterms. Another classical system, the λμ-calculus, has been combined with System T [ref], but I could not find any similar work for the DC and λμT is not concerned with duality. There is a natural extension of DC with arbitrary inductive and coinductive types [ref], but restricting the extension to only natural and conatural numbers seems infeasible, and extending the Agda formalisation with (co)inductive types is a major endeavour outside the scope of my project. Consequently, I decided to work with what I have, and employ some workarounds to turn Church numeral computations into normalisable Agda programs. My original goal of comparing execution times of CBN and CBV evaluation assumed that I will be able to write complex enough programs to demonstrate the difference, which, due to the limitations of the syntax, sadly turned out to not be the case.

produce meaningful ‘ℕ terms, just in a rather convoluted way. tbc

We first declare a module to do this all in, 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 translation of a term with type A has type (A → ℕ) → ℕ, meaning 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

  open import Dual.CPSTransformation ℕ
```

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

We then convert this CN-2+2 term into a term of type ‘ℕ. Note that CN is simply shorthand for (B → B) → (B → B), and that in our DC implementation of the STLC B is instantiated to ‘ℕ. This means that if we apply a term of type CN to terms of type ‘ℕ → ‘ℕ (a successor function) and ‘ℕ (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 : (∅ , ‘ℕ , ‘ℕ ⇒ ‘ℕ) ⟶ ∅ | ‘ℕ
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 which we normalise to discover the value of. 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+2^V : ℕ
2+2^V = (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 the remarkable result that 2 + 2 = 4. This shows us that the DC formalisation is capable of simulating arithmetic correctly, I showed a similar result for the mult function which I do not show here.

4.3.2 Simply-Typed Lambda Calculus with letcont and throw

The Types lecture course [24] introduces a simple extenstion 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
```

```
; 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 that hold only in classical logic, this validates that the DC formalisation represents classical logic. The theorems we prove are the law of double negation elimination and Peirce's law. Figure 4.3.1 presents a derivation tree of the proof as well as its encoding in Agda alongside the DC term it normalises to, and figure 4.3.2 does the same for Peirce's law.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x\colon \neg\neg A, \mu\colon \neg A \vdash x\colon \neg\neg A}\,(\mathrm{Id}) \quad \dfrac{}{x\colon \neg\neg A, \mu\colon \neg A \vdash \mu\colon \neg A}\,(\mathrm{Id})}{x\colon \neg\neg A, \mu\colon \neg A \vdash \mathtt{throw}[x, \mu]\colon A}\,(\mathrm{throw})}{x\colon \neg\neg A \vdash \mathtt{letcont}\mu\colon \neg A.\mathtt{throw}[x, \mu]\colon A}\,(\mathrm{Cont})}{\varnothing \vdash \lambda x\colon \neg\neg A.\mathtt{letcont}\mu\colon \neg A.\mathtt{throw}[x, \mu]\colon \neg\neg A \supset A.}\,(\to\mathrm{I})$$

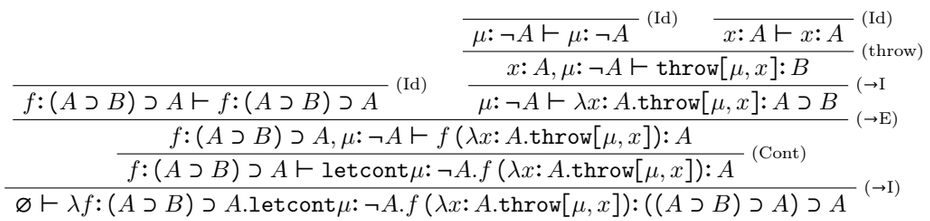
add some spacing in places with \, and ~

```
DNE : ∀ {Γ A} → Γ ⟶ ∅ | (¬ (¬ A)) ⇒ A

DNE = λ̲ (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+



```
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+

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) , (λ̲ throw[ (# 1) , (# 0) ]) ]

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

probably want to do (Γ → ∅ | A) → (Γ → ∅ | B) → (Γ → ∅ | and A B)?
Not much difference tho

4.3.3 Lambda-Mu Calculus

We can also simulate Parigot's [32] $\lambda\mu$-Calculus within the Dual Calculus. This interface and implementation is slightly more complex as the $\lambda\mu$ contains both terms and commands. We define terms in the same way as we did for the STLC, though we no longer fix the succedent Context and also include a μ binder field, this is implemented with the DC's $\mu\theta$ constructor. Commands are implemented with DC's statements, they have one constructor which 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 [43] to the original paper.

but is conceptually closer to DC because of the dual contexts

you need to show the code otherwise this explanation is useless

λμ ~ DC where coterms can only be covariables

```
[_]_ : ∀ {Γ Δ A} → Δ ∋ A → Λ-Term Γ Δ A → Λ-Comm Γ Δ
```

```
[_]_ = 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 = λ̲ (μ ([ ‘Z ] ((# 0) · (λ̲ (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 very 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 implicitly assumed properties, 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 the this, sub or ren appear, in some form, 1333 times in my source code, and 'substitution' and 'renaming' appear 159 times in my dissertation. I believe this effectively highlights my point, to formally prove only 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.

An interesting contrast to this however is cps-V. This 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, which ended up being critical. I had been stuck on the proof of the soundness of the denotational semantics, with the lemmas I had left to prove appearing 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 operation. If I could redo the project I would have spent more of my dedicated research time looking in to 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 very 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 derivable DC sequent was both type- and scope-safe, and requirement to define operations that maintain that guarantee, was of significant help throughout.

With 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 very 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 purely 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, as well as implementing 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, their 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 CPS transformation 'target calculus', before implementing a CPS transformation into, and back out of, this calculus. One could then demonstrate that this CPS transformation is a *Galois connection*. Completing the proofs of the

a strong relationship between the reduction relations of and translations between DC and the target calculus

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

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

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

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

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

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

[15] 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.

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

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

[18] Marcelo Fiore. Denotational semantics. Cambridge University lecture notes, 2020.

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

[20] 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.

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

[22] 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.

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

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

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

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

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

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

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

[30] 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.

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

[32] 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.

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

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

[35] 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.

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

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

[38] 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.

[39] 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.

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

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

[42] 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.

[43] 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.

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

Appendix A

Agda Code

Appendix B

Project Proposal