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

Work Completed

Special Difficulties

Contents

Chapter 1

need an engaging start for the introduction to get the reader engaged and thinking, so probably start with motivation and go into the project description next

this is usually one of the hardest parts of writing so you might need several iterations

Introduction

better to keep these "here's what this chapter is going to be about" bits to sections 2-3-4 – also, not worth spending too many words on it if it doesn't add much (i.e. just restates the section headings)

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 Project Structure

use hyperref to make references work and have a pdf table of contents

Philip

The aim of the project was to formalise the Dual Calculus, as set out by Wadler [39] in a paper that describes the syntax and semantics of the language, as well as various operations on and properties of said language. The formalisation of the core part of the Calculus can be found in Section 3.1, and formal proofs of the properties he lays out, including the paper's key conclusion: that the Call-by-Value CPS Transformation is the dual of the Call-by-Name CPS Transformation, are in Section 3.2. This constitutes the core part of my project.

can introduce the abbreviation DC here

I'd say the CBV is dual to CBN is about the operational semantics, rather than CPS (I know this is still under construction)

operational semantics

I was also able to work on some extensions, including defining an Operational Semantics for the Dual Calculus, which necessitated a formal definition of substitution ~~(a definition that Wadler~~

soundness transformations

can discuss this in the relevant section

~~omits)~~, and formally proving the Soundness of both of the CPS Transformations provided. These can be found in Section 3.3 and Section 3.4 respectively.

basically no need to capitalise so many things

The existence of the formalisation in Agda is itself an evaluation of the project. I also, however, used the formalised Dual Calculus to produce proofs of various theorems of classical logic (in Section 4.2). As well as this I used the Dual Calculus formalisation I produced 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 Dual Calculus formalisation.

yes kinda but if you state it like that it sounds like a cop-out (which it is) :D

will probably be revisited

1.2 Motivation

1.2.1 The Dual Calculus

Curry--Howard (en-dash between surnames of different people)

add a bit of high-level waffle about Curry–Howard at the beginning to situate the project, and put this section at the start of the chapter

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 [7] and Gentzen's Intuitionistic Natural Deduction [16]. It was, however, years after this relationship was first noted that it was first extended to classical logic. This was when Griffin [17] 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 ~~systems~~.

worth emphasising that it was more-or-less accidental – people did not expect there to be a computational content for classical logic so it wasn't even really investigated

-- (en-dash around sentence interruptions) ((((or em-dash --- but that's American and we don't like that)))

It was Filinski [15] 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 [39].

Wadler presented the Dual Calculus, 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 Dual Calculus, however, is how explicitly, and cleanly, it demonstrates the duality between Call-by-Value and Call-by-Name.

1.2.2 Choice of Programming Language

The first reason I decided to use Agda [23] for this project, instead of any other proof assistant, was that it's the main language my supervisor (and project originator) works in, and as such I would better be able to draw on his knowledge for assistance when needed. Agda is also 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) [40]. Another key benefit is that the language is taught in the tripos and is easier to pick up if you have experience of functional programming, which appears regularly in the tripos. Proofs in Agda are spelled out explicitly rather than being hidden behind opaque tactics like in Coq [4] or Isabelle [28], and the syntax of Agda is a lot cleaner and more flexible, allowing mixfix operators and use of unicode characters.

if you mention this, maybe put it at the end – it's an important practical consideration but reads a bit weird. The other reasons are convincing enough I think!

introduced Tripos

A better order would be
– Widely used proof assistant
– Support, libraries, etc
– readable proofs, syntax, etc
– mentioned in tripos
– (supervision)

1.2.3 Contribution of my Project

The original paper that sets out the Dual Calculus 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 benefit of formal verification that is well-known, the most famous example of this being Nipkow's paper 'Winskel is (almost) right' [22] 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

This is a well-known benefit of formal verification

is it that famous?

https://nautil.us/issue/24/Error/in-mathematics-mistakes-arent-what-they-used-to-be

a good article about verification

believe I am the only person to produce such a formalisation of the Dual Calculus.

Producing such formal proofs, however, is "extremely laborious" [30] 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.

1.2.4 Not sure where I will talk about "The Gap" yet, maybe here

I think that discussion would be perfect for the evaluation

1.3 Related Work

[...]

Tzevelekos [38] describes the Dual Calculus 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 [17] 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 [27], which this extends the Simply Typed λ-calculus with a μ binder that abstracts on the programs current program's 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 [26], and Ong and Stewart [25] respectively. Crolard then derived a λ-calculus with a catch/throw mechanism from this, which he calls the λ_{ct}-calculus [8]. Separately, Barbanera produced a λ-calculus with symmetric reduction rules that corresponded to classical logic [3], 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 [15]. Sellinger, in two separate papers [35, 36], modeled the Call-by-Name and Call-by-Value semantics of the $\lambda\mu$-calculus in a control categories 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 [9]. 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 [3] also has a clear notion of duality, however they do not consider Call-by-Name and Call-by-Value evalutation strategies. Wadler's Dual Calculus demonstrates the duality without any of the limitations of the above.

might be interesting to have a list of relevant properties and why they are desirable, and how they are handled by previous research e.g. connection to logical duality, involution, computational interpretation

Chapter 2

Preparation

This chapter aims to provide the necessary background knowledge of the various aspects of the project, primarily Agda and the Dual Calculus, as well as outlining the starting point of the project and the tools I used for my implementation.

2.1 Starting Point

To the extent of my knowledge no-one has produced a formalisation of the Dual Calculus 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 [29]. As soon as I knew my project would be formalising some calculi in Agda I started studying PLFA [40] to teach myself how to write in Agda. I also studied sections of the textbook *Practical Foundations for Programming Languages* [] 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.

2.2 Curry-Howard Correspondence

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

The Curry-Howard correspondence, also known as the formulae-as-types interpretation, is a relationship between programs and mathematical proofs. Curry made the first observation of this relationship when he, in 1934, he noted that the types of combinators, a kind of programming construct, can be interpreted as axiom-schemes for a variation of intuitionistic logic [10]. Over 20 years later Curry discovered another example of this relationship [12], a correspondence between Hilbert-style deduction systems and combinatory logic [34, 11] (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 correspondence explicitly was Howard, who, in 1969, demonstrated a direct and specific relationship between Intuitionistic Natural Deduction and the Simply-Typed λ-calculus [18]. The correspondence has since been extended to Classical Logic, as was detailed in Section 1.3.

Haskell

...relationship in 1934 when he noted ...

it's nice to give the full name of people who are influential or important for the research and mentioned repeatedly (Griffin, Wadler, Curry, Howard)

2.3 The Dual Calculus

The Dual Calculus 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 provided as primitives, and implication is defined in terms of these connectives, though this definition is different for the Call-by-Value and Call-by-Name calculi. The following

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 Elimination	Continuation-Passing Style Transformation

translation

section outlines the calculus.

2.3.1 Syntax

The syntax of the Dual Calculus is as follows:

Definition 2.3.1 (Syntax)**.**

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

I didn't completely understand why Wadler put implication as part of the syntax and then say that it's actually derivable in two different ways – I guess it might be just to establish a direct "proof term" language for sequent calculus. I'd say it's better not to have it in the syntax to begin with

A Dual Calculus 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 Dual Calculus are defined as:

Definition 2.3.2 (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 Dual Calculus 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.

2.3.2 Typing Relation

The existence of variable, covariable, and λ abstraction requires the tracking of the types of the various free variables and covariables. This is done in two separate typing environments: the antecedent is a list of pairs of free variables and types; and the succedent is a list of pairs of free covariables and types. They are defined below.

Definition 2.3.3 (Grammar of Antecedents and Succedents)**.**

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

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

just Γ = ...
::= is for BNF definitions (which this isn't)

many ideas came before him

~~Wadler defines~~ three different types of *sequent* in the Dual Calculus: Left Sequents, Right Sequents, and Centre Sequents. Each consists of an antecedent and a succedent though are slightly different from each other. A right sequent contains a distinguished element in the succedent,

this distinguished element is a `term:type` pair (rather than a `covariable:type` pair). A left sequent contains a distinguished element in the antecedent, a `coterm:type` pair (rather than a `variable:type` pair). A centre sequent has no distinguished element, though it contains a statement.

you are more-or-less paraphrasing Wadler's paper here, which is a bit dangerous

Definition 2.3.4 (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 as follows: 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 here that statements do not have types, they are simply the cut of a term against a coterm of the same type.

not really clear what this means to the uninitiated (and again directly lifted from the paper)

Definition 2.3.5 (Typing)**.** The typing rules for the Dual Calculus are given in Figure 2.3.1

Note that all right rules result in a right sequent, and equally all left rules result in a left sequent. As well as this, it should be noted that & and ∨ rules have the same type of sequent in the hypothesis and the conclusion whereas the ¬ rules swap from a left to a right sequent or vise versa.

vice

you will later explain these in more detail so these paragraphs are just restatements of the typing rules

The only typing rules that are particularly interesting are IR and IL. IR states that if the statement S is valid given antecedent Γ and succedent $\Theta, \alpha : A$ then the coterm $(S).\alpha$, where the covariable α is bound in S, is of type A given antecedent Γ and succedent Θ. IL is simply the dual for terms.

The rules for left sequents 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 you might expect. This is

:

,

one

usually prefer "we" or "one" (or passive voice), not "you"

Figure 2.3.1: The typing rules for the Dual Calculus

use \inferrule*[Right=~(IdR)] or something instead

$$\textsc{IdR}\ \frac{}{x\colon A \rightarrowtail \mid x\colon A} \qquad \textsc{IdL}\ \frac{}{\alpha\colon A \mid \rightarrowtail \alpha\colon A}$$

$$\&\mathrm{R}\ \frac{\Gamma \rightarrowtail \Theta \mid M\colon A \qquad \Gamma \rightarrowtail \Theta \mid N\colon B}{\Gamma \rightarrowtail \Theta \mid \langle M, N\rangle\colon A \,\&\, B} \qquad \&\mathrm{L}1\ \frac{K\colon A \mid \Gamma \rightarrowtail \Theta}{\mathrm{fst}[K]\colon A \,\&\, B \mid \Gamma \rightarrowtail \Theta} \qquad \&\mathrm{L}2\ \frac{L\colon B \mid \Gamma \rightarrowtail \Theta}{\mathrm{snd}[L]\colon A \,\&\, B \mid \Gamma \rightarrowtail \Theta}$$

$$\vee\mathrm{R}1\ \frac{\Gamma \rightarrowtail \Theta \mid M\colon A}{\Gamma \rightarrowtail \Theta \mid \langle M\rangle\mathrm{inl}\colon A \vee B} \qquad \vee\mathrm{R}2\ \frac{\Gamma \rightarrowtail \Theta \mid N\colon B}{\Gamma \rightarrowtail \Theta \mid \langle N\rangle\mathrm{inr}\colon A \vee B} \qquad \vee\mathrm{L}\ \frac{K\colon A \mid \Gamma \rightarrowtail \Theta \qquad L\colon B \mid \Gamma \rightarrowtail \Theta}{[K, L]\colon A \vee B \mid \Gamma \rightarrowtail \Theta}$$

$$\neg\mathrm{R}\ \frac{K\colon A \mid \Gamma \rightarrowtail \Theta}{\Gamma \rightarrowtail \Theta \mid [K]\mathrm{not}\colon \neg A} \qquad \neg\mathrm{L}\ \frac{\Gamma \rightarrowtail \Theta \mid M\colon A}{\mathrm{not}\langle M\rangle\colon \neg A \mid \Gamma \rightarrowtail \Theta}$$

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

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

$$\textsc{Cut}\ \frac{\Gamma \rightarrowtail \Theta \mid M\colon A \qquad K\colon A \mid \Gamma \rightarrowtail \Theta}{\Gamma \mid M \bullet K \mapsto \Theta}$$

because left sequents contain coterms, and coterms *consume* values rather than produce them. These rules become easier to understand if one has a grasp on the computational content of the constructs of the Dual Calculus.

2.3.3 Computational Content

Break the section up with \paragraph{...}: types, sequents, terms, coterms

The computational content of the types of the Dual Calculus is clear for products $A \,\&\, B$, sums $A \vee B$, and functions $A \supset B$. The only computationally interesting type is the negated type $\neg A$, the computational interpretation of which is not necessarily intuitive. When one considers that the the logical proposition $\neg A$ is equivalent to $A \supset \bot$ it becomes clearer to see 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.

no need to repeat saying that these are unintuitive, just explain what they mean

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 ~~computational~~ interpretation of the right sequent

avoid word repetition

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

is that if one supplies a value of type A_i to each x_i' then evaluating the term M either returns a value of type A or pass a value of type B_i to one of the continuation variables α_i. Similarly, the left sequent

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

is interpreted as supplying a value of type A_i to each variable x_i, a value of type A to the coterm K and then evaluating will pass a value type B_i to some continuation variable α_i. The interpetation

of the centre sequent

$$x_1 : A_1, ..., x_m : A_m \mathbin{\blacksquare} S \mapsto \alpha_1 : B_1, ..., \alpha_m : B_m$$

is simply if each variable x_i is given a value of type A_i then evaluation will return a value of type B_i to one continuation variable α_i.

... terms, statements and coterms are detailed below.

The computational meaning of some of the terms, statements, and especially coterms of the Dual Calculus can be unintuitive, so I detail them below.

Terms produce values, while coterms consume them

Terms *produce* values and come in the following forms with associated computational interpretation:

\begin{description}
\item[Variable] ...

Variable: The term x simply produces the value given to the variable x.

Pair of terms: The term $\langle M, N \rangle$ produces a value of type $A \,\&\, B$, a pair of the values of type A and B yielded by M and N.

A left or right injection of a term: The term $\langle M \rangle \text{inl}$ produces a value of type $A \vee B$ given by the left injection of M, a value of type A. Similar for the term $\langle N \rangle \text{inr}$.

Negation of a coterm: The term $[K]\text{not}$ produces a continuation of type $\neg A$ (equivalent to $A \supset \bot$), this takes a value of type A that is consumed by coterm K.

Function abstraction of a term: The term $\lambda x.N$ produces a function (functions are values) of type $A \supset B$ that takes an argument x of type A giving the value of type B produced by the term N.

Covariable abstraction of a statement: The term $(S).\alpha$ executes the statement S, yielding the value of type A passed to the covariable α.

Coterms *consume* values, they occur in the following forms and are interpreted as follows:

Covariable: The coterm α consumes a value, giving it out to the covariable α.

Left or right projection of a coterm: The coterm $\mathrm{fst}[M]$ consumes a value of type $A \,\&\, B$, projects the first value of type A and passes it to be consumed by the coterm K. Similar for $\mathrm{snd}[N]$.

Case of two coterms: The coterm $[K, L]$ consumes a value of type $A \vee B$ passes it to the relevant component coterm, 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.

Negation of a term: The coterm $\mathrm{not}\langle M \rangle$ consumes a continuation of type $\neg A$ by giving it the value of type A produced by the term M.

Application of a term to a coterm: The coterm $M \mathbin{@} K$ consumes a function of type $A \supset B$ by using the value of type A produced by M as an argument for said function and passing the value of type B that this function produces to the coterm K, which consumes it.

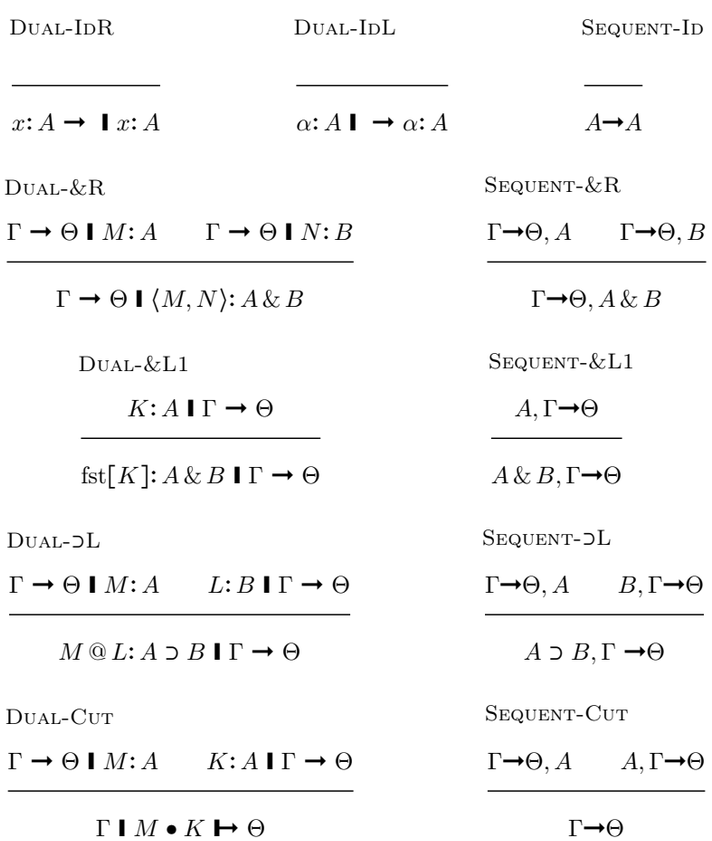
Variable abstraction of a statement: The coterm $x.(S)$ consumes a value of type A, binds this to the variable x and then executes the statement S.

Statements come only in the form $M \bullet K$ and simply pipe together a term and a coterm, feeding the value of type A produced by term M to be consumed by coterm K.

2.3.4 Curry-Howard Correspondence for the Dual Calculus

As has been mentioned, the Dual Calculus 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 Dual Calculus makes this obvious. This subsection will discuss how to interpret the Dual Calculus logically.

Figure 2.3.2: Side by side comparision of some sequent calculus and Dual Calculus inference rules



The types of the Dual Calculus clearly correspond to propositions in classical logic ~~(their interpretations are trivial)~~ 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: the conjunction of the propositions in the antecedent implies the disjunction of the propositions in the succedent. For exmample the right sequent

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

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.

The logical interpretation of terms, proofs of a distinguished proposition in the succedent, is as follows:

perhaps you can somehow combine the computational and logical interpretation sections because there is a lot of repetition

Variable: The term x proves the proposition A by simply propagating the hypothesis x of proposition A.

Pair of terms: The term $\langle M, N \rangle$ proves the proposition $A \,\&\, B$, with a pair of proofs of propositions A and B: M and N.

A left or right injection of a term: The term $\langle M \rangle \text{inl}$ proves the proposition $A \vee B$ by left injection of M, a proof of proposition A. Similar for the term $\langle N \rangle \text{inr}$.

Negation of a coterm: The term $[K]\text{not}$ proves the proposition $\neg A$. The coterm K in the antecedent proves propostion A, this is equivalent to proving $\neg A$ in the succedent. The antecedent is a conjunction of propositions that imply one of the propositions in the succedent,

if we remove one of those propositions, say A from the antecedent then one of the following must be true: one of the existing propositions in the succedent is still true, or the proposition $\neg A$ must be true (otherwise the A is true and the implication from before would hold). $[K]$not simply uses this fact to prove $\neg A$

Function abstraction of a term: The term $\lambda x.N$ proves the implication $A \supset B$ by assuming the hypothesis x of proposition A and proving proposition B with proof N.

Covariable abstraction of a statement: The term $(S).\alpha$ proves the proposition A by assuming the existence of α, a refutation of the proposition $\neg A$, and deriving the contradiction S.

The logical interpretation of coterms is much easier to understand when using the fact that a proof in the antecedent is equivalent to a refutation in the succedent. Coterms are proofs of a distinguished formula in the antecedent, so can be interpreted as follows:

Covariable: The coterm α refutes the proposition A by propagating the hypothesis α of the refuation of A.

Left or right projection of a coterm: The coterm $\mathrm{fst}[K]$ refutes the proposition $A \,\&\, B$, by projecting the coterm K, a refutation of A. Similar for $\mathrm{snd}[L]$.

Case of two coterms: The coterm $[K, L]$ refutes the proposition $A \vee B$ with a pair of refutations, K and L, of the propositions A and B respectively.

Negation of a term: The coterm $\mathrm{not}\langle M \rangle$ refutes the proposition $\neg A$ by using the proof of proposition A given by the term M.

Application of a term to a coterm: The coterm $M \mathbin{@} K$ refutes the implication $A \supset B$ with a pair of a term and a coterm. The term M proves proposition A, while the coterm K refutes

the proposition B, this refutes the propositon $A \supset B$,

Variable abstraction of a statement: The coterm $x.(S)$ refutes the proposition A, by assuming the existence of x, a proof of A, and deriving the contradiction S.

The statement $M \bullet K$ simply represents a contradiction between the term M, a proof of A, and coterm K, a refutation of A. Using these contradictions for proofs, as variable and covariable abstractions do, is a key feature of classical logic. It relies on the law of double negation elimination ($\neg\neg AA \supset$) by assuming $\neg A$ we derive some kind of contradiction, thus demonstrating $\neg\neg A$, which, in classical logic, entails A. This should make the role of statements, and variable and covariable abstractions of them, in the Dual Calculus much clearer.

The relationship to classical logic described in the last subsection gives us the following proposition.

Proposition 2.3.6. *A proposition A is provable in classical logic iff there exists a closed Dual Calculus term M such that* $\rightarrowtail \shortmid M : A$.

All the rest here is just a reproduction of Wadler's paper – instead you should leave these to the implementation where you can actually state and define things in Agda

2.3.5 Dual Translation

Dual Calculus types, terms, coterms, statements, antecedents, and succedents all have duals, with the notable exception of anything that involves implication. This is because implication can be defined in terms of the other connectives, though this will be discussed in more detail later. The dual translation, which maps a a Dual Calculus construct to its dual is defined below. This translation assumes a involutive bijection between variables and covariables.

Definition 2.3.7 (Dual Translation)**.**

Types

$$(X)^\circ \equiv X$$

$$(A \,\&\, B)^\circ \equiv A^\circ \vee B^\circ$$

$$(A \vee B)^\circ \equiv A^\circ \,\&\, B^\circ$$

$$(\neg A)^\circ \equiv \neg A^\circ$$

Terms

$$(x)^\circ \equiv x^\circ$$

$$(\langle M, N\rangle)^\circ \equiv [M^\circ, N^\circ]$$

$$(\langle M\rangle \mathrm{inl})^\circ \equiv \mathrm{fst}[M^\circ]$$

$$(\langle N\rangle \mathrm{inr})^\circ \equiv \mathrm{snd}[N^\circ]$$

$$([K]\mathrm{not})^\circ \equiv \mathrm{not}\langle K^\circ\rangle$$

$$((S).\alpha)^\circ \equiv \alpha^\circ.(S^\circ)$$

Coterms

$$(\alpha)^\circ \equiv \alpha^\circ$$

$$([K, L])^\circ \equiv \langle K^\circ, L^\circ\rangle$$

$$(\mathrm{fst}[K])^\circ \equiv \langle K^\circ\rangle \mathrm{inl}$$

$$(\mathrm{snd}[L])^\circ \equiv \langle L^\circ\rangle \mathrm{inr}$$

$$(\mathrm{not}\langle M\rangle)^\circ \equiv [M^\circ]\mathrm{not}$$

$$(x.(S))^\circ \equiv (S^\circ).x^\circ$$

Statements

$$(M \bullet K)^\circ \equiv K^\circ \bullet M^\circ$$

Antecedents

$$(x_1 : A_1, ..., x_m : A_m)^\circ \equiv x_1^\circ : A_1^\circ, ..., x_m^\circ : A_m^\circ$$

Succedents

$$(\alpha_1 : B_1, ..., \alpha_m : B_m)^\circ \equiv \alpha_1^\circ : B_1^\circ, ..., \alpha_m^\circ : B_m^\circ$$

This dual translation has two important properties which are outlined below.

Proposition 2.3.8. *The dual translation is an involution.*

$$A^{\circ\circ} \equiv A$$

$$M^{\circ\circ} \equiv M \qquad K^{\circ\circ} \equiv K$$

$$S^{\circ\circ} \equiv S$$

$$\Gamma^{\circ\circ} \equiv \Gamma \qquad \Theta^{\circ\circ} \equiv \Theta$$

Proposition 2.3.9. *A dual calculus term, coterm, or statement (that does not include implication) is derivable iff its dual is derivable.*

$$\Gamma \rightarrow \Theta \mid M : A \iff M^{\circ} : A^{\circ} \mid \Theta^{\circ} \rightarrow \Gamma^{\circ}$$

$$K : A \mid \Gamma \rightarrow \Theta \iff \Theta^{\circ} \rightarrow \Gamma^{\circ} \mid K^{\circ} : A^{\circ}$$

$$\Gamma \mid S \mapsto \Theta \iff \Theta^{\circ} \mid S^{\circ} \mapsto \Gamma^{\circ}$$

2.3.6 Reduction Relation

The Dual Calculus has two separate reduction relations, one Call-by-Value, the other Call-by-Name. These relations depend on a subset of Terms and Coterms, known as Values and Covalues, which are defined as follows:

Definition 2.3.10 (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 gives us to ability to define the reduction relations for the Dual Calculus, these definitions are given below. Observe that moving from a Call-by-Value to a Call-by-Name reduction relation, as in the λ-calculus, generalises values to terms. However in the Dual Calculus, this change also specialises coterms to covalues.

Definition 2.3.11 (Call-by-Value Reduction Relation)**.**

$$
\begin{array}{lrcl}
(\beta\&1) & \langle V, W\rangle \bullet \mathrm{fst}[K] & \longrightarrow_v & V \bullet K \\
(\beta\&2) & \langle V, W\rangle \bullet \mathrm{snd}[L] & \longrightarrow_v & W \bullet L \\
(\beta\vee 1) & \langle V\rangle\mathrm{inl} \bullet [K, L] & \longrightarrow_v & V \bullet K \\
(\beta\vee 2) & \langle W\rangle\mathrm{inr} \bullet [K, L] & \longrightarrow_v & W \bullet L \\
(\beta\neg) & [K]\mathrm{not} \bullet \mathrm{not}\langle M\rangle & \longrightarrow_v & M \bullet K \\
(\beta\supset) & \lambda x.N \bullet V@L & \longrightarrow_v & V \bullet x.(N \bullet L) \\
(\beta L) & V \bullet x.(S) & \longrightarrow_v & S\{V/x\} \\
(\beta R) & (S).\alpha \bullet K & \longrightarrow_v & S\{K/\alpha\}
\end{array}
$$

Definition 2.3.12 (Call-by-Name Reduction Relation)**.**

$$
\begin{array}{lrcl}
(\beta\&1) & \langle M, N\rangle \bullet \mathrm{fst}[P] & \longrightarrow_n & M \bullet P \\
(\beta\&2) & \langle M, N\rangle \bullet \mathrm{snd}[Q] & \longrightarrow_n & N \bullet Q \\
(\beta\vee 1) & \langle M\rangle\mathrm{inl} \bullet [P, Q] & \longrightarrow_n & M \bullet P \\
(\beta\vee 2) & \langle N\rangle\mathrm{inr} \bullet [P, Q] & \longrightarrow_n & N \bullet Q \\
(\beta\neg) & [K]\mathrm{not} \bullet \mathrm{not}\langle M\rangle & \longrightarrow_n & M \bullet K \\
(\beta\supset) & \lambda x.N \bullet M@Q & \longrightarrow_n & M \bullet x.(N \bullet Q) \\
(\beta L) & M \bullet x.(S) & \longrightarrow_n & S\{M/x\} \\
(\beta R) & (S).\alpha \bullet P & \longrightarrow_n & S\{P/\alpha\}
\end{array}
$$

To justify the existence of values and covalues, it is useful to consider the (βL) and (βR) rules. Without either of these reductions being restricted to values or covalues then reduction can exhibit some problematic behaviour. Consider the statement $(x \bullet \alpha).\beta \bullet y.(z \bullet \gamma)$, without the restrictions of values and covalues both (βL) and (βR) would apply, making both the below reductions possible.

$$(x \bullet \alpha).\beta \bullet y.(z \bullet \gamma) \longrightarrow x \bullet \alpha$$

$$(x \bullet \alpha).\beta \bullet y.(z \bullet \gamma) \longrightarrow z \bullet \gamma$$

To solve this issue, Call-by-Value will only reduce a cut of a term against a variable abstraction if the term is a value, but does not restrict reduction of a cut of a covariable abstraction against a coterm. Call-by-Name does the opposite.

These reductions allow us to make the titular claim of the original paper: Call-by-Value is dual to Call-by-Name.

Proposition 2.3.13. *For terms, coterms, and statements that do not include implication, Call-by-Value is dual to Call-by-Name.*

$$M \longrightarrow_v N \Leftrightarrow M^\circ \longrightarrow_n N^\circ$$

$$K \longrightarrow_v L \Leftrightarrow K^\circ \longrightarrow_n L^\circ$$

$$S \longrightarrow_v T \Leftrightarrow S^\circ \longrightarrow_n T^\circ$$

Once again we exclude implication as it can be defined in terms of the other connectives. However, since under Call-by-Value a function abstraction must be a value and under Call-by-Name function application must be a covalue, we require separate definitions for the two different strategies.

Proposition 2.3.14. *With Call-by-Value reduction, implication/functions can be defined as*

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

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

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

Proposition 2.3.15. *With Call-by-Name reduction, implication/functions can be defined as*

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

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

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

Using these definitions it is possible to derive the inference and reduction rules for implication from those of the other connectives.

2.3.7 Continuation-Passing Style Transformations

Continuation-Passing Style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation. CPS functions take an extra argument, a continuation, representing the rest of the computation to be carried out, and returns by applying this continuation to the value the computation results in. A CPS transformation maps the terms of a language into this style of programming, the first example of this was Plotkin's λ_v-calculus and corresponding transformation [31]. Reynolds provides a detailed summary of the history of continuations in his 1993 paper [32].

The Dual Calculus has two separate CPS transformations, one Call-by-Value and on Call-by-Name, both map types, (co)values, terms, coterms, and statements (that do not include implication) of the Dual Calculus into CPS translated types and terms of a fragment of the Simply-Typed λ-Calculus. These transformations are defined below.

Definition 2.3.16 (Call-by-Value Continuation-Passing Style Transformation)**.** The Call-by-Value CPS transformation is outlined in Figure 2.3.3

Definition 2.3.17 (Call-by-Name Continuation-Passing Style Transformation)**.** The Call-by-Name CPS Transformation is defined in Figure 2.3.4

The CPS transformation of Values and Covalues can be related to the CPS transformation of Terms and Coterms in the following way.

Proposition 2.3.18. *Let V be a value and P a covalue of the Dual Calculus.*

$$(V)^v \equiv \lambda\gamma.\gamma\,(V)^V$$

$$(P)^n \equiv \lambda z.z\,(P)^N$$

It's interesting to note that the CPS transformations embed the Dual Calculus, which corresponds to a classical logic, into the Simply Typed λ-calculus, which corresponds to an intuitionistic logic. This shows how CPS transformation corresponds to the process of Double Negation Translation, a translation that takes a classical logic proof to an intuitionistic proof with the same meaning. This also gives us the rather interesting property that, despite the inclusion of Peirce's Law and the Law of Double Negation Elimination, classical logic is actually a subsystem of intuitionistic logic.

There are various interesting properties of these transformations as well, those that are relevant to this project are outlined below.

Figure 2.3.3: Call-by-Value CPS Transformation

Types

$$(X)^V \equiv X$$
$$(A \,\&\, B)^V \equiv (A)^V \times (B)^V$$
$$(A \vee B)^V \equiv (A)^V + (B)^V$$
$$(\neg A)^V \equiv (A)^V \to R$$

Values

$$(x)^V \equiv x$$
$$(\langle V, W \rangle)^V \equiv \langle (V)^V, (W)^V \rangle$$
$$(\langle V \rangle \mathrm{inl})^V \equiv \mathtt{inl}(V)^V$$
$$(\langle W \rangle \mathrm{inr})^V \equiv \mathtt{inr}(W)^V$$
$$([K]\mathrm{not})^V \equiv (K)^v$$

Terms

$$(x)^v \equiv \lambda\gamma.\gamma\, x$$
$$(\langle M, N \rangle)^v \equiv \lambda\gamma.(M)^v(\lambda x.(N)^v(\lambda y.\gamma\, \langle x, y \rangle))$$
$$(\langle M \rangle \mathrm{inl})^v \equiv \lambda\gamma.(M)^v(\lambda x.\gamma(\mathtt{inl}\, x))$$
$$(\langle N \rangle \mathrm{inr})^v \equiv \lambda\gamma.(N)^v(\lambda y.\gamma(\mathtt{inr}\, y))$$
$$([K]\mathrm{not})^v \equiv \lambda\gamma.\gamma\,(\lambda z.(K)^v\, z)$$
$$((S).\alpha)^v \equiv \lambda\alpha.(S)^v$$

Coterms

$$(\alpha)^v \equiv \lambda z.\alpha\, z$$
$$([K, L])^v \equiv \lambda z.\mathtt{case}\ z\ \mathtt{of}\ \mathtt{inl}\ x \to (K)^v\, x\,;\ \mathtt{inr}\ y \to (L)^v\, y$$
$$(\mathrm{fst}[K])^v \equiv \lambda z.\mathtt{case}\ z\ \mathtt{of}\ \langle x, \_ \rangle \to (K)^v\, x$$
$$(\mathrm{snd}[L])^v \equiv \lambda z.\mathtt{case}\ z\ \mathtt{of}\ \langle \_, y \rangle \to (L)^v\, y$$
$$(\mathrm{not}\langle M \rangle)^v \equiv \lambda z.(\lambda\gamma.(M)^v\, \gamma)\, z$$
$$(x.(S))^v \equiv \lambda x.(S)^v$$

Statements

$$(M \bullet K)^v \equiv (M)^v\,(K)^v$$

Figure 2.3.4: Call-by-Name CPS Transformation

Types

$$(X)^N \equiv X$$
$$(A \,\&\, B)^N \equiv (A)^N + (B)^N$$
$$(A \vee B)^N \equiv (A)^N \times (B)^N$$
$$(\neg A)^V \equiv (A)^N \to R$$

Covalues

$$(\alpha)^N \equiv \alpha$$
$$([P, Q])^N \equiv \langle (P)^N, (Q)^N \rangle$$
$$(\mathrm{fst}[P])^N \equiv \mathtt{inl}\ (P)^N$$
$$(\mathrm{snd}[Q])^N \equiv \mathtt{inr}\ (Q)^N$$
$$(\mathrm{not}\langle M \rangle)^N \equiv (M)^n$$

Terms

$$(x)^n \equiv \lambda\gamma.\alpha\,\gamma$$
$$(\langle M, N \rangle)^n \equiv \lambda\gamma.\mathtt{case}\ \gamma\ \mathtt{of\ inl}\ \alpha \to (M)^n\,\alpha\,;\ \mathtt{inr}\ \beta \to (N)^n\,\beta$$
$$(\langle M \rangle \mathrm{inl})^v \equiv \lambda\gamma.\mathtt{case}\ \gamma\ \mathtt{of}\ \langle \alpha, \_ \rangle \to (M)^n\,\alpha$$
$$(\langle N \rangle \mathrm{inr})^v \equiv \lambda\gamma.\mathtt{case}\ \gamma\ \mathtt{of}\ \langle \_, \beta \rangle \to (N)^n\,\beta$$
$$([K]\mathrm{not})^v \equiv \lambda\gamma.(\lambda z.(K)^n\,z)\,\gamma$$
$$((S).\alpha)^n \equiv \lambda\alpha.(S)^n$$

Coterms

$$(\alpha)^n \equiv \lambda z.z\,\alpha$$
$$([K, L])^n \equiv \lambda z.(K)^n(\lambda\alpha.(L)^n(\lambda\beta.z\,\langle\alpha, \beta\rangle))$$
$$(\mathrm{fst}[K])^n \equiv \lambda z.(K)^n(\lambda\alpha.z(\mathtt{inl}\,\alpha))$$
$$(\mathrm{snd}[L])^n \equiv \lambda z.(L)^n(\lambda\beta.z(\mathtt{inr}\,\beta))$$
$$(\mathrm{not}\langle M \rangle)^n \equiv \lambda z.z\,(\lambda\gamma.(M)^n\,\gamma)$$
$$(x.(S))^n \equiv \lambda x.(S)^n$$

Statements

$$(M \bullet K)^n \equiv (K)^n\,(M)^n$$

Proposition 2.3.19. *The Call-by-Value and Call-by-Name CPS transformations preserves types.*

$$\Gamma \rightarrowtail \Theta \shortmid V : A \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (V)^V : (A)^V$$

$$\Gamma \rightarrowtail \Theta \shortmid M : A \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (M)^v : (\neg\neg A)^V$$

$$K : A \shortmid \Gamma \rightarrowtail \Theta \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (K)^v : (\neg A)^V$$

$$\Gamma \shortmid S \mapsto \Theta \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (S)^v : R$$

$$P : A \shortmid \Gamma \rightarrowtail \Theta \Leftrightarrow (\neg\Gamma)^N, (\Theta)^N \vdash (P)^N : (A)^N$$

Where the notation $\Gamma, \Theta \vdash M : A$ *means the Simply Typed* λ*-calculus term* M *has type* A*, given the typing environments* Γ *and* Θ*.*

Proposition 2.3.20. *The Call-by-Value and Call-by-Name CPS transformations are dual.*

$$(A)^V \equiv (A^\circ)^N$$

$$(V)^V \equiv (V^\circ)^N$$

$$(M)^v \equiv (M^\circ)^n$$

$$(K)^v \equiv (K^\circ)^n$$

$$(S)^v \equiv (S^\circ)^n$$

Proposition 2.3.21. *The CPS transformations preserve substitution.*

$$(S\{V/x\})^v \equiv (S)^v\{(V)^V/x\}$$

$$(S\{K/\alpha\})^v \equiv (S)^v\{(K)^v/\alpha\}$$

$$(S\{M/x\})^n \equiv (S)^n\{(M)^n/x\}$$

$$(S\{P/\alpha\})^n \equiv (S)^n\{(P)^N/\alpha\}$$

Proposition 2.3.22. *The CPS transformations preserve reductions*

$$M \longrightarrow_v N \Leftrightarrow (M)^v \longrightarrow (N)^v$$

$$K \longrightarrow_v L \Leftrightarrow (K)^v \longrightarrow (L)^v$$

$$S \longrightarrow_v T \Leftrightarrow (S)^v \longrightarrow (T)^v$$

and the same for Call-by-Name

2.4 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 [24], and I used it for all of my proofs.

one can see the line of inspiration from Stella through Ran to this ;) No problem of course, as long as it's not too blatant!

2.4.1 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.

```
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.

```
_+_ : 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 + and returning 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

[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.

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.

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

2.4.2 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 $(x : \mathsf{A}) \to \mathsf{B}\ x$ 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.

```
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 is an exhaustive pattern match as [] cannot give 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.

2.4.3 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
```

wat

This can be interpreted as follows: for a type APTA 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. This is a notion of equality based on proving that two terms are the same semantically, rather than just syntactically, as is the case with definitonal equality.

ehh I'd be careful with that wording since we don't have semantics anywhere

It is also worth discussing universal quantification. In general, if we have a variable x of type A and a dependent type B x, then we can produce a 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:

universal quantification is just dependent functions – the ∀ syntax in agda is just a dependent function type where the type of the argument can be inferred

```
+-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)
```

2.4.4 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.

```
+-identity^r : ∀ (m : ℕ) → m + zero ≡ m
```

We then go about proving the 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.

```
+-identity^r 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 clearly use suc as the function argument for cong, and we can appeal to our inductive hypothesis to produce the proof that the two terms are equal.

```
+-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)
```

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 required 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)). Once again using congruence with suc and appeal to the inductive hypothesis is sufficient to prove this.

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

Now we can get to actually proving 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 this is proved by our +-identity^r lemma.

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

The inductive case 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
```

compress proof

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

Equational reasoning allows us to write a series of equivalent equations, like a human might when producing a proof, 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 defintionally equal to (suc n) + m, thus completing the proof.

2.5 Tools and Design

Visual Studio Code

I used *banacorn's agda-mode* extension for vscode for writing all my code, this provides the ability VSCode
to execute Agda commands and allows for simple unicode input in vscode, making it an effective
\LaTeX
IDE for writing in Agda. The Agda distribution I used also has an ~~experimental~~ Latex back-end allowing easy typesetting of Agda code, which I have used to prepare this report.

The Dual Calculus 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 Dual Calculus 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 Dual Calculus 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 λ-calculus and showed that the Dual Calculus formalisation is capable or representing simple arithmetic correctly.

Chapter 3

Implementation

~~This chapter is made up of 4 sections.~~ The first defines the Syntax of the Dual Calculus, the next two define the semantics of the Dual Calculus, one from an operational standpoint, the other denotational. The final section of this chapter discusses and proves the duality of each the previous sections.

need a repository overview

3.1 Syntax

This section presents the important definitions of my Dual Calculus formalisation, making notice of any interesting issues, and discussing and justifying any design choices I made.

syntax

3.1.1 Intrinsically-typed representation of Syntax

Definitions of a typed language can be *extrinsic* or *intrinsic* [33]. 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 have no meaning without considering

what also explain scope safety here

their typing, it is the typing that meaning is assigned, making ill-typed terms meaningless. Using inductive *families* [14], rather than inductive types, to represent syntax Altenkirch and Reus [2], Bellegarde Paterson Bellegard and Hook [5], and Bird and Patterson [6] all showed how intrinsic typing can be used to achieve *scope-safety*. rather, they show how type- and scope-safety can be enforced using inductive families

Scope-safe terms require that every variable is bound by some binder, or is explicitly tracked in some typing environment or *context*. Using a family indexed by a set allows us to track this scoping information at the type level, meaning that type checking will guarantee that a term is well-scoped. Indexing this family on the type (a type in the language we are defining, not an Agda type) has a similar effect, guaranteeing that a term that type-checks in Agda is well-typed in the defined language. discuss the two at the same time, since type- and scope-safety comes hand-in-hand (can't have intrinsically typed variables without reference to a scope)

An extrinsically typed representation of a language requires either naming variables, in which case maintaining α-equivalence becomes a difficult issue, or using de Bruijn notation [13] which can result into running into difficult to debug errors from having to correctly modify all de Bruijn indices each time the scope changes [20]. As well as this, intrinsically-typed representations of languages tend to require significantly less code [40]. numeric indices / implemented as natural numbers

you don't explain how intrinsic typing solves this

For all the above reasons I decided to use an intrinsically-typed representation of the syntax of the Dual Calculus, following closely the representation of the λ-calculus laid out by Altenkirch and Reus [2]. cite PLFA again, A&R do something slightly different

The types of the Dual Calculus can be represented in Agda with an inductive data type. Here I use the natural numbers as base type X. X

```
data Type : Set where

  `ℕ : Type

  _`×_ : Type → Type → Type
```

you can use spaces to align constructor types nicely (see agda doc)

```
_‘+_ : Type → Type → Type

‘¬_ : Type → Type
```

Recall that 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.

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

return to implication later

```
_⇒ᴺ_ : Type → Type → Type
A ⇒ᴺ B = ‘¬ A ‘+ B
```

For an intrinsically-typed representation of the Dual Calculus, terms, coterms and statements must be indexed by a set representing the scope of the program, just like antecedents and succedents that give the types of free variables and covariables. In Agda we can represent this as a list of Types, each type representing the type a free variable or covariable that can be referred to in the term, coterm or statement, with the most recently bound variable on the right of the list.

Just say
Typing environments are lists 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. These are analogous to the zero and

For example, the third most recently.... `S (`S (`Z)). The constructors are analogous .... for ℕ; indeed, variables can be seen as type- and scope-safe de Bruijn indices.

suc constructors for ℕ. These two constructors can be used to access every single variable in the Context, for example the third most recently bound variable can be constructed with ‘S (‘S (‘Z)).

```
data _∋_ : Context → Type → Set where

  ‘Z : ∀ {Γ A}
       -------
    → Γ , A ∋ A

  ‘S : ∀ {Γ A B}
    → Γ ∋ A
       -------
    → Γ , B ∋ A
```

I think you have a page limit two, which makes these layouts too wasteful – will unfortunately need to put everything on one line (it can still be formatted nicely though)

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 in the sequent it appears in.

We now have all we need to define Dual Calculus 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 independent of their Type and Contexts. These are defined as mutually inductive families below. Note that the constructors of these sequents are essentiallt the typing relation of the Dual Calculus, this is because intrinsic-typing requires that a sequent only has meaning if it is well-typed, this definition makes it impossible to construct a sequent that is not well-typed.

y

double run-on sentence wowee (look up ‘comma splice’)

is there a shorter arrow you could use here?

```
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

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

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

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

if you’re presenting them in parallel, put the ⟨_,_⟩ and [_,_] constructors side-by-side

```
data _⟼_ where

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

one line

3.1.2 Values and Covalues

This subsection provides a definition for values and covalues, a construct we will need throughout the implementation. 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 Dual Calculus term represented by M is a value. The definitions of these families is as expected, with the constructors matching the defintions 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 respectively. These are helpful construct which we will make use of throughout the implementation.

```
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
```

side by side

also comment out the line
\advance\leftskip\mathindent
in agda.sty
so the code doesn't get indented by default

3.1.3 Proof of Validity of Implication

Propositions 2.3.14 and 2.3.15 assert that we can derive the inference rules for implication using the two implication definitions and the existing primitive inference rules. Intrinsic typing makes this an easy proof.

This section is incomplete as I need to make some slight changes to derive the inference rules of implication.

substitution should be discussed in this section (even though it's used in the operational semantics only)

3.2 Operational Semantics

This section defines the operational semantics of the dual calculus, defining its reduction relations, both Call-by-Name and Call-by-Value, as well as type-preserving definitions of renaming and substitution.

3.2.1 Reduction Relations

The Dual Calculus has two reduction relations, one Call-by-Value, the other Call-by-Name. Each reduction relation can take a statement to a statement, a term to a term and a coterm to a coterm, so would have to be implemented as three seperate inductive families in Agda. However, the η and ς rules are not required for any of the properties I intend to prove, in fact, Wadler mentions that when they are omitted the Calculus is strongly normalising. Therefore, for the sake of simplicity I chose to omit these expansion rules from the formalised operational semantics, defining only the congruence and reduction rules. This means that every reduction takes a statement to a statement, thus we require only two inductive families, one for the Call-by-Value reduction relation, and one for the Call-by-Name variant. 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

you don't mention these in the prep so the reader won't know what this is about

relation as an example.

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

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

but worth stating that type preservation and progress are kinda vacuous – you don't even have types, only contexts

The constructors of this inductive family correspond directly to the rules of the reduction relation 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.

```
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.

```
βL : ∀ {Γ Θ A} {V : Γ ⟶ Θ | A} {S : Γ , A ⟼ Θ} (v : Value V)
     ---------------
  → V • (μγ S) ⟶ˢ^V S ^V⟨ ⟨ V , v ⟩ /⟩ˢ

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

Note how both rules include a substitution, substituting a right sequent in the case of βR and a TermValue in the case of βL. These operations are simply shorthand for more general substitution

functions, substitution functions which must be type-preserving; these will be outlined in the following sections. should be discussed in prev section

```
_^V⟨_/⟩^s : ∀ {Γ Θ A}
  → Γ , A ⟼ Θ
  → TermValue Γ Θ A
    --------
  → Γ ⟼ Θ

_^V⟨_/⟩^s {Γ}{Θ} S V =
  sub-statement TermValueKit CotermKit (add (λ Γ A → TermValue Γ Θ A) V id-termvalue) id-coterm S

_[_/]^s : ∀ {Γ Θ A}
  → Γ ⟼ Θ , A
  → A | Γ ⟶ Θ
    -----
  → Γ ⟼ Θ

_[_/]^s {Γ}{Θ} S K =
  sub-statement TermValueKit CotermKit id-termvalue (add (λ Θ A → A | Γ ⟶ Θ) K id-coterm) S
```

I also defined two more inductive families (again, one for each evaluation strategy) to represent multi-step reduction, indexed by two centre sequents. An instance of the inductive family will type-check if one can prove that the first centre sequent will reduce to the second centre sequent after an arbitrary number of reductions. These proofs of reductions are built up through chains of reductions, much like with Agda standard library's equational reasoning. These chains start don't need this much detail

with a begin^{sV} function, then follows a chain of centre sequents, separated by the $\_\ ^{s}\!\longrightarrow^{v}\langle\ \_\ \rangle\ \_$ constructor (call this the step constructor), ending with a $\_\Box^{sV}$. In the gap between the angle brackets of the step constructor is where one provides evidence that the centre sequent preceding the constructor does, in fact, reduce to the sequent following it, this will always be in the form of a reduction rule. A simple example of this relation at work 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 =
  begin sV
    (not[ fst[ K ] ]) • (not⟨ ‘⟨ V , W ⟩ ⟩)
  s⟶V⟨ β¬ ⟩
    ‘⟨ V , W ⟩ • fst[ K ]
  s⟶V⟨ β×₁ v w ⟩
    V • K
  □sV
```

put term and proof step on one line (e.g. like in https://agda.github.io/agda-stdlib/Relation.Binary.PropositionalEquality.html#3214)

3.2.2 Renaming and Substitution

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 [21]. While the motivation for implementing substitution should be clear, the motivation for renaming may be less so and is worth expanding on. When we substitute into a variable/covariable abstraction term/coterm we

since this will be in 3.1, you should forward-reference the reduction relation

(co)variable abstraction (co)term

must recursively substitute into the statement it abstracts on. To avoid variable capture however, we must rename the variables we are mapping from in our substitution, in our case incrementing the de Bruijn indices to account for the new bound variable.

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]. McBride provides a brilliant example of this for the STLC in his 2005 paper [21], which I base my implementation on. McBride defines an abstract ~~inductive~~ 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*, indexed by this inductive family, we can this 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 for the terms of the STLC. Keller demonstrates how we can use Agda record types to represent these kits [19], 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 dual calculus, as a result of two key differences between it and the STLC. 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, we have three different types of sequents and a var can be either a variable *or* a covariable. This means that a kit indexed by an generic T, itself indexed by two Contexts and a Type, would have to support mapping in from variables and covariables, and mapping out to right sequents/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.

bit out-of-place :D

Keller [19] demonstrates ...
(otherwise not immediately clear if I should know who Keller is)

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 an generic T, and 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 an generic C, does the same but with covariables and right sequents. The kit for vars is parameterised by an generic T, indexed by a Context and a Type, this supports mapping into by var and weakening the Context it is indexed by. We can actually use these var kits to support mapping into 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.

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

record CotermSubstKit (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}       → VarSubstKit (λ Θ A → C Γ Θ A)
```

can these not be called VarKit, TermKit and CotermKit?

>>> no, because that's the name of the instances

could you have synonyms for (λ Γ A → T Γ Θ A) and (λ Θ A → C Γ Θ A) e.g. Fix_2 Θ T and Fix_1 Γ T

```
open module K {Γ} = VarSubstKit (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 a type 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 that where the Sorted-Family is instantiated to the datatype `_∋_`.

not needed

don't include comments

```
-Context map-

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

-Renaming map-

_⇝_ : 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. The function is mutually inductive so the type signature for substitution into left and right sequents is also given.

what's going on with the 's here

```
sub-term : ∀ {T A C Γ Θ Γ′ Θ′} → TermSubstKit T → CotermSubstKit C
  → Γ –[ (λ Γ A → T Γ Θ′ A) ]→ Γ′ → Θ –[ (λ Θ A → C Γ′ Θ A) ]→ Θ′
  → Γ ⟶ Θ | A → Γ′ ⟶ Θ′ | A
sub-coterm : ∀ {T A C Γ Θ Γ′ Θ′} → TermSubstKit T → CotermSubstKit C
```

```
  → Γ –[ (λ Γ A → T Γ Θ′ A) ]→ Γ′ → Θ –[ (λ Θ A → C Γ′ Θ A) ]→ Θ′

  → A | Γ ⟶ Θ → A | Γ′ ⟶ Θ′

sub-statement : ∀ {T C Γ Θ Γ′ Θ′} → TermSubstKit T → CotermSubstKit C

  → Γ –[ (λ Γ A → T Γ Θ′ A) ]→ Γ′ → Θ –[ (λ Θ A → C Γ′ Θ A) ]→ Θ′

  → Γ ⟼ Θ → Γ′ ⟼ Θ′
```

what if the instances are called TVKit and CKit? Then the record names can be TermKit and CotermKit

show type signature and explain that it's CBV so needs TermValues

```
_ᵛ⟨_/⟩ˢ {Γ}{Θ} S V =

  sub-statement TermValueKit CotermKit (add (λ Γ A → TermValue Γ Θ A) V id-termvalue) id-coterm S
```

The substitution function takes a TermSubstKit T and a CotermSubstKit C. As well as this, the substitution function will take two arbitrary substitutions, one modifying the antecedent Context and producing instances of the Sorted-Family T, the other modifying the succedent and producing instances of C. This produces a function from one sequent to another, modifying both the Contexts it is indexed by according to the subsitutions given.

or have some sort of consistent (and shorter) naming convention, like
sub-T, sub-C, sub-S
ren-T, ren-C, ren -S
Kit-TV, Kit-C
id-TV, id-C
etc.
I think that would work fine

We will first 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 will define CotermKit, CotermKit is an instance of the CotermSubstKit record, with C instantiated to a left sequent. This kit must provide the ability to map out from C to a left sequent, weaken the antecedent Context and provide an instance of a VarSubstKit that provides mapping into C from a var and weakening the succdedent Context. Since C is already instantiated to a left sequent, mapping out to a left sequent is simply the identity function. We define the VarSubstKit instance with a record constructor, the mapping into C, a left sequent, is provided by the ‘_ constructor, however for the weakening of the succedent Context we must define renaming. The same is true for weakening the antecedent *Context*.

just looking forward, you will probably not have much space and opportunity to discuss it in as much detail as you have, unfortunately. Fortunately you started from top down so you may just give a brief outline of the different components of the definition above rather than a full derivation.

It's nevertheless good that you've written everything down – let's see how much space is left once the rest is figured out.

We can define renaming of sequents by mutual induction, with one function for each of the types of sequents. These type signatures are essentially simpler versions of the type signatures for substitution, note the use of renaming maps rather than general context maps.

```
rename-term : ∀ {Γ Γ′ Θ Θ′ A} → Γ ⇝ Γ′ → Θ ⇝ Θ′ → Γ ⟶ Θ | A → Γ′ ⟶ Θ′ | A

rename-coterm : ∀ {Γ Γ′ Θ Θ′ A} → Γ ⇝ Γ′ → Θ ⇝ Θ′ → A | Γ ⟶ Θ → A | Γ′ ⟶ Θ′

rename-statement : ∀ {Γ Γ′ Θ Θ′} → Γ ⇝ Γ′ → Θ ⇝ Θ′ → Γ ⟼ Θ → Γ′ ⟼ Θ′
```

Most of the cases of this renaming function are defined with simple induction, the interesting cases are for vars and variable/covariable abstraction. In the case for vars we use the `‘_` to construct a sequent from the direct application of the relevant renaming map to the var. For the variable abstraction case we must *lift* the antecedent Context renaming since 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.

```
rename-term ρ ϱ (‘ x) = ‘ (ρ x)

rename-coterm ρ ϱ (‘ α) = ‘ (ϱ α)

rename-term ρ ϱ (μθ S) = μθ (rename-statement ρ (rename-lift ϱ) S)

rename-coterm ρ ϱ (μγ S) = μγ (rename-statement (rename-lift ρ) ϱ S)
```

The function rename-lift is defined recursively on the var that the lifted renaming maps from, the implementation is quite simple, though requires us to define the weakening of a renaming. can be inlined, as you did

```
rename-lift : ∀ {Γ Δ A} → Γ ⇝ Δ → (Γ , A) ⇝ (Δ , A)
rename-lift ρ ‘Z = ‘Z
rename-lift ρ (‘S x) = ‘S (ρ x)
```

The last requirement is to define the weakening of a renaming, in which we extend the Context the renaming is mapping into. The type signature of the function rename-weaken is given below, the function is implemented by simply applying the ‘S constructor the renamed variable.

```
rename-weaken : ∀ {Γ Δ A} → Γ ⇝ Δ → Γ ⇝ (Δ , A)
rename-weaken ρ x = ‘S (ρ x)
```

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

Now we define the TermValueKit, this is very similar to the CotermKit. TermValueKit is an instance of TermSubstKit with T instantiated to TermValue. We map out the TermValue to a right sequent by projecting the right sequent out of the TermValue. Weakening the succedent Context is provided by using TermKit's implementation of weakening on the left sequent in the pair (TermKit is simply the dual of CotermKit). However this must be accompanied by proving that the arbitrary renaming of a Value is still a Value; the type signature of such a proof, and the dual proof for Covalues is given below, though its implementation is omitted.

```
value-rename : ∀ {Γ Γ′ Θ Θ′ A} {V : Γ ⟶ Θ | A} (ρ : Γ ⇝ Γ′) (ϱ : Θ ⇝ Θ′) →
  Value V → Value (rename-term ρ ϱ V)

covalue-rename : ∀ {Γ Γ′ Θ Θ′ A} {P : A | Γ ⟶ Θ} (ρ : Γ ⇝ Γ′) (ϱ : Θ ⇝ Θ′) →
  Covalue P → Covalue (rename-coterm ρ ϱ P)
```

We map into a TermValue from a var by constructing a pair from the application of the ‘_ constructor to the var and the CV-covar constructor. We implement the weakening of the antecedent Context in the same way as we do the weakening of the succedent Context.

```
TermValueKit : TermSubstKit TermValue

TermValueKit = record

  { tm = λ x → proj₁ x

  ; wkΘ = λ x → ⟨ (TermSubstKit.wkΘ TermKit (proj₁ x))

                , value-rename id-var (rename-weaken id-var) (proj₂ x) ⟩

  ; kit = record

    { vr = λ x → ⟨ ‘ x , V-var ⟩

    ; wk = λ x → ⟨ (VarSubstKit.wk (TermSubstKit.kit TermKit) (proj₁ x))

                , value-rename (rename-weaken id-var) id-var (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, its implemetation is simple and is given below.

```
add : ∀{Γ Δ A}(T : Context → Type → Set) → T Δ A → Γ –[ T ]→ Δ → (Γ , A) –[ T ]→ Δ

add T t σ ‘Z = t

add T t σ (‘S v) = σ v
```

id-termvalue and id-coterm simply represent the id termvalue and coterm substitutions respectively, there implementations are trivial so are omitted.

The example of the use of the sub-statement should now make sense so we will now discuss the implementation of substitution. As a reminder, this is the type signature of the mutually inductive substitution function.

```
sub-term : ∀ {T A C Γ Θ Γ′ Θ′} → TermSubstKit T → CotermSubstKit C
```

```
  → Γ –[ (λ Γ A → T Γ Θ′ A) ]→ Γ′ → Θ –[ (λ Θ A → C Γ′ Θ A) ]→ Θ′
  → Γ ⟶ Θ | A → Γ′ ⟶ Θ′ | A
sub-coterm : ∀ {T A C Γ Θ Γ′ Θ′} → TermSubstKit T → CotermSubstKit C
  → Γ –[ (λ Γ A → T Γ Θ′ A) ]→ Γ′ → Θ –[ (λ Θ A → C Γ′ Θ A) ]→ Θ′
  → A | Γ ⟶ Θ → A | Γ′ ⟶ Θ′
sub-statement : ∀ {T C Γ Θ Γ′ Θ′} → TermSubstKit T → CotermSubstKit C
  → Γ –[ (λ Γ A → T Γ Θ′ A) ]→ Γ′ → Θ –[ (λ Θ A → C Γ′ Θ A) ]→ Θ′
  → Γ ⟼ Θ → Γ′ ⟼ Θ′
```

As is the case with renaming, the interesting cases are for variables, covariables, variable abstraction, and covariable abstraction; the other cases are simple induction. For variables we apply the TermSubstKit instance's mapping an of T to a right sequent to the application of antecedent substitution on the variable. The covariable case is entirely dual, using CotermSubstKit and the succedent substitution instead.

```
sub-term k₁ k₂ s t (` x) = TermSubstKit.tm k₁ (s x)
```

```
sub-coterm k₁ k₂ s t (` α) = CotermSubstKit.tm k₂ (t α)
```

The variable abstraction case is similar to the variable abstraction case for renaming, but slightly more complex. As is familiar we must lift the Γ to Γ' substitution to a Γ , A to Γ' , A substitution, this accounts for the newly bound variable. However, unlike renaming, we also must modify the succedent substition 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 instead.

```
sub-term {T}{A}{C}{Γ}{Θ}{ΓI}{ΘI} k₁ k₂ s t (μθ S) = μθ (sub-statement k₁ k₂
  (fmap {λ - → T - ΘI} {λ - → T - (ΘI , A)} (λ x → TermSubstKit.wkΘ k₁ x) s)
  (sub-lift (CotermSubstKit.kit k₂) t) S)

sub-coterm {T}{A}{C}{Γ}{Θ}{ΓI}{ΘI} k₁ k₂ s t (μγ S) = μγ (sub-statement k₁ k₂
  (sub-lift (TermSubstKit.kit k₁) s)
  (fmap {C ΓI} {C (ΓI , A)} (λ x → CotermSubstKit.wkΓ k₂ x) t) S)
```

The function sub-lift is a generalisation of ren-lift, taking a VarSubstKit paramterised by a Sorted-Family T, it is implemented by induction on the variable the lifted substitution maps from. The base case simply maps the 'Z constructor into an instance of T, whereas the inductive case, where the induction variable is instantiated to ('S x), uses the sub-weaken function on the var x. This sub-weaken function simply applies the VarSubstKit's implementation of weakening to the substituted var.

```
sub-lift : ∀ {Γ Δ A T} → VarSubstKit T → Γ –[ T ]→ Δ → (Γ , A) –[ T ]→ (Δ , A)
sub-lift k σ 'Z = VarSubstKit.vr k 'Z
sub-lift k σ ('S x) = sub-weaken k σ x

sub-weaken : ∀ {Γ Δ A T} → VarSubstKit T → Γ –[ T ]→ Δ → Γ –[ T ]→ (Δ , A)
sub-weaken k σ x = VarSubstKit.wk k (σ x)
```

fmap takes a family of functions from $T\ \Gamma\ A$ to $TI\ \Gamma\ A$ and a substitution from Γ to Γ' that produces instances of the Sorted-Family T. This substitution is modified to produce instances of TI by applying f to the result of the substitution applied to a variable.

will need to think about how to present this section because obviously it's quite important in the formalisation but isn't the main point of focus of the project. At the same time, there are some interesting new challenges compared to previous literature, but it might be difficult to appreciate them without detail.

```
fmap : ∀ {T T′ Γ Γ′} (f : ∀ {Γ A} → T Γ A → T′ Γ A) → Γ –[ T ]→ Γ′ → Γ –[ T′ ]→ Γ′
fmap f σ ‘x = f (σ ‘x)
```

With this final definition we have produced a definition of substitution that preserves type and scope-safety, thus completing our definition of the operational semantics of the Dual Calculus.

3.2.3 Operational Semantics of Implication

This section will just demonstrate the reduction rule for implication is derivable from the reduction rules for other connectives, I haven't implemented this yet though it should be simple enough.

3.3 Denotational Semantics

The following section defines a Denotational Semantics of the Dual Calculus, interpreting Dual Calculus 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 transformation.

3.3.1 Continuation-Passing Style Transformation

The CPS transformation defined in the original paper maps from the Dual Calculus to the Simply-Typed λ-calculus, 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 Dual Calculus I decided that we could instead 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, this is because

the Call-by-Name CPS transformation is entirely dual and has no interesting differences to the Call-by-Value variant that can't be noted from the definitions in the original paper.

Again we start with defining the operation for Type and Context, we make extensive use of constructs from the Agda standard library including the ℕ datatype, and implementations of products (×) and sums (⊎) which the corresponding Dual Calculus Type constructors are transformed into. We introduce the arbitrary type R from the original paper by paramaterising the CPSTransformation module by R of type Set. This makes the implementation of the transformation for Types simple. The transformation takes a Context, which is just a list of Types, to a tuple of transformed Types. This requires mapping the empty Context, ∅, to an empty tuple, which turns out to be the unit type. This is easier to understand when you look at it from the logical side of the Curry-Howard correspondence rather than the programming side. If we consider a tuple of one term, this is just the same as a two term tuple with one of said terms being an instance of the empty tuple. The type of this tuple corresponds to a proposition conjoined with whatever we logically interpret the empty tuple as, we clearly want this conjunction to be true if the proposition is true and false if the proposition is false; this behaviour is achieved if we logically interpret the empty tuple as *true*. It's a standard result of the Curry-Howard correspondence that *true* corresponds to the unit type, as such we transform the ∅ to the unit type ⊤.

interpreting contexts as products of types is standard so needs not be elaborated on. You may as well show the code here instead of explaining what happens in probably more words.

```
_^VT : Type → Set
_^Vx : Context → Set
```

ah right, when I said you can omit the implementation and keep the type signatures, I referred less important lemmas and definitions – interpretation of types is quite important!

Now we define the transformation of vars. Say we have a var of type $\Gamma \ni A$, we clearly want to use the transformed var as an Agda term of type A^{VT}. A var is typed-scoped de Bruijn index, pointing a position in a Context, as such it is clear that a transformed var should point to that same position in a transformed Context, where it will locate the transformed Type we want. Therefore

a transformed var must be a function from the transformation of the Context it is indexed by to the transformation of the Type it is indexed by. Transforming the var results in a pointer to certain position in a tuple, which you then pass the transformed Context to to get the desired transformed Type. The transformation then is a simple recursive function on vars, abstracting on the transformed Context, returning the last element in the base case, and recursively transforming the var with a reduced Context in the recursive case.

again, interpreting variables as projections is standard

"As is standard in the set-theoretic model of computational calculi, a typing context is interpreted as the Cartesian product of the interpretations of its elements, and variables are the appropriate projection functions." <code>

$$\_^{VV} : \forall \{\Gamma\ A\} \to (\Gamma \ni A) \to ((\Gamma\ ^{Vx}) \to (A\ ^{VT}))$$

$$\_^{VV}\ \text{`Z} = \lambda\ c \to \text{proj}_2\ c$$

use pattern-matching instead of proj

$$\_^{VV}\ (\text{`S}\ x) = \lambda\ c \to ((x\ ^{VV})\ (\text{proj}_1\ c))$$

Noting that a var must be transformed to a function from a transformed Context to a transformed Type gives us the intuition that transforming of sequents will not be dissimilar. Proposition 2.3.18 gives us some assistance here, as a reminder, here is the Call-by-Value part of that proposition.

but will they be similar??

$$\Gamma \rightarrowtail \Theta \shortmid V : A \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (V)^V : (A)^V$$

$$\Gamma \rightarrowtail \Theta \shortmid M : A \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (M)^v : (\neg\neg A)^V$$

$$K : A \shortmid \Gamma \rightarrowtail \Theta \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (K)^v : (\neg A)^V$$

$$\Gamma \shortmid S \mapsto \Theta \Leftrightarrow (\Gamma)^V, (\neg\Theta)^V \vdash (S)^v : R$$

At the beginning of the section give a high-level summary of how denotational semantics works, e.g.
– interpret types as sets and contexts as products of sets
– interpret terms as functions from denotations of contexts to denotations of types
– prove semantic substitution lemma
– prove soundness of denotational semantics wrt. operational semantics
and instead explain the slight differences in DC, namely the presence of two contexts, negation, and value interpretations.

This essentially gives us the type signature of the transformation of sequents, the only have to bridge is the difference between the STLC typing judgements given above, and the Agda type we desire. Clearly we want the transformation of sequents to be able to produce Agda terms with the types given on the far right of the above rules, which in fact represent continuations.

However, these sequents could include vars within them. This means that we will have to represent the transformation of a sequent as a function from transformed Contexts, two in this case, to a transformed Type. This will allow us to recursively transform the vars within the sequent using the transformation of the antecedent and the transformation of the negation of the succedent, in the case of Call-by-Value. This allows us to derive the following type signature for the transformation of sequents, where $`\neg^{x}$ is a trivial implementation of the negation of a Context.

this explanation is also not really needed

$$\_^{VLV} : \forall \{\Gamma\ \Theta\ A\} \to \mathsf{TermValue}\ \Gamma\ \Theta\ A \to (\Gamma\ ^{Vx} \times (`\neg^{x}\ \Theta)\ ^{Vx}) \to (A\ ^{VT})$$

$$\_^{VL} : \forall \{\Gamma\ \Theta\ A\} \to (\Gamma \longrightarrow \Theta \mid A) \to (\Gamma\ ^{Vx} \times (`\neg^{x}\ \Theta)\ ^{Vx}) \to ((`\neg\ `\neg\ A)\ ^{VT})$$

$$\_^{VR} : \forall \{\Gamma\ \Theta\ A\} \to (A \mid \Gamma \longrightarrow \Theta) \to (\Gamma\ ^{Vx} \times (`\neg^{x}\ \Theta)\ ^{Vx}) \to ((`\neg\ A)\ ^{VT})$$

$$\_^{Vs} : \forall \{\Gamma\ \Theta\} \to (\Gamma \longmapsto \Theta) \to (\Gamma\ ^{Vx} \times (`\neg^{x}\ \Theta)\ ^{Vx}) \to R$$

instead, expand on why you need both terms and values, and what's going on with negations and R

The transformation itself is then implemented in a way that very closely mirrors the definition given in the original paper, the key difference is that we abstract on the the pair of transformed Contexts and then must, in some cases, modify this pair before making a recursive call to the transformation. The cases in which we must do this are the variable and covariable cases, and the variable and covariable abstraction cases. The variable and covariable cases are simple, we are calling the var transformation rather than recursively calling a sequent transformation, the var transformation requires only one transformed Context so we simply project the relevant Context from the pair before passing it to the transformation. 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 $\Gamma \ni A$, it is trivial to prove. We must include this because the covariable α has type $\Theta \ni A$, which the var transformation would take to a function from $\Theta\ ^{Vx}$ to $A\ ^{VT}$, while we require a function from $`\neg^{x}\ \Theta$ to $(`\neg\ A)\ ^{VT}$. We also modify the pair of transformed Contexts in the variable and covariable abstraction cases, this is relatively simple. Clearly a variable abstraction expands the antecedent,

call this lemma something like ¬var (if anything)

this means we add this new variable to the transformed antecedent before making the recursive call. We do the same for covariable abstractions but with the succedent instead. 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

$$\langle\ \text{`}\ x\ ,\ \mathsf{V\text{-}var}\ \rangle\ ^{VLV} = \lambda\ c \to (x\ ^{VV})\ (\mathsf{proj}_1\ c) \qquad \text{pattern-match on c rather than project}$$

$$(\text{`}\ x)\ ^{VL} = \lambda\ c\ k \to k\ ((x\ ^{VV})\ (\mathsf{proj}_1\ c))$$

$$(\text{`}\ \alpha)\ ^{VR} = \lambda\ c\ z \to ((\Gamma \ni A \Rightarrow \neg\Gamma \ni \neg A\ \alpha)\ ^{VV})\ (\mathsf{proj}_2\ c)\ z$$

$$(\mu\theta\ S)\ ^{VL} = \lambda\ c\ \alpha \to (S\ ^{Vs})\ \langle\ (\mathsf{proj}_1\ c)\ ,\ \langle\ (\mathsf{proj}_2\ c)\ ,\ \alpha\ \rangle\ \rangle$$

$$(\mu\gamma\ S)\ ^{VR} = \lambda\ c\ x \to (S\ ^{Vs})\ \langle\ \langle\ \mathsf{proj}_1\ c\ ,\ x\ \rangle\ ,\ (\mathsf{proj}_2\ c)\ \rangle$$

This transformation also doubles up as a proof of proposition 2.3.18, this is because our intrinsically typed representation of syntax required us to define the transformation on well-typed and well-scoped sequents rather than raw terms, coterms and statements, producing well-typed Agda programs with preserved types.

3.3.2 CPS Transformation of Values

Proposition 2.3.18 defines a relation between the the CPS transformation, I now prove this proposition, as well as a dual proposition for covalues. The type signatures of the theorems are given below, note that the proofs are done point-wise on the transformed pair of Contexts.

$$\mathsf{cps\text{-}value} : \forall\ \{\Gamma\ \Theta\ A\}\ (V : \Gamma \longrightarrow \Theta \mid A)\ (v : \mathsf{Value}\ V)\ (c : \Gamma\ ^{Vx} \times (\text{`}\neg^{x}\ \Theta)\ ^{Vx})$$
$$\to (V\ ^{VL})\ c \equiv \lambda\ x \to x\ ((\langle\ V\ ,\ v\ \rangle\ ^{VLV})\ c)$$

```
cps-covalue : ∀ {Γ Θ A} (P : A | Γ ⟶ Θ) (p : Covalue P) (c : Θ ^Nx × (‘¬^x Γ) ^Nx)
  → (P ^NR) c ≡ λ z → z (((⟨ P , p ⟩ ^NRV) c)
```

actually do you need ext for this? You could just do it pointwise like with c

Both are relatively simple inductive proofs. For the variable/covariable and not⟨_⟩ / not[_] the equality holds by definition, as such, refl is a sufficient proof. The other cases make use of ext – shorthand for the Agda standard library's Extensionality – ext states that if two functions applied to the same argument always yield the same result then the two functions are equal. Interestingly, ext cannot be proved within Agda, so we must use the postulate keyword, this allows us to use a theorem without proving it. This does not cause any issues as ext is known to be consistent with the theory underlying Agda. We use ext by providing a proof that $(\forall\ (x : A) \to f\ x \equiv g\ x)$, this will return a proof that $f \equiv g$. Invoking ext allows us to consider the continuation and then use congruence with the definition of the CPS transformation of the relevant constructor and the inductive hypothesis/hypotheses. I present a couple of the cases of the proofs below as examples.

```
cps-value inl⟨ V ⟩ (V-inl v) c = ext (λ k → cong (λ - → - (λ x → k (inj₁ x))) (cps-value V v c))
```

```
cps-covalue ‘[ P , Q ] (CV-sum p q) c = ext (λ z →
  cong₂ (λ -₁ -₂ → -₁ (λ α → -₂ (λ β → z ⟨ α , β ⟩)))) (cps-covalue P p c) (cps-covalue Q q c))
```

I think the proof is not especially interesting or aesthetic so it can be omitted. It may however be worth mentioning (here, or the evaluation) the whole deal with us getting stuck on some proofs and then discovering that this lemma covered all the missing cases, as it was an interesting example of the paper author anticipating something that wasn't seemingly important for the purposes of formalisation (as a kind of opposite of the "gap", where it's the formalisation that brings up various details that were not anticipated by the author)

3.3.3 CPS Transformation of Renamings and Substitutions

Though not defined in the original paper, if we want to consider the CPS transformations of sequents that include a subsitution within them then we have to be able to interpret a substitution, and therefore also a renaming, as a function on Agda terms.

The interpretation of a renamed sequent will clearly be some kind of composition of the interpretation of the original sequent, and the interpretation of the renaming. I believe this is easier to

understand with an example. Consider M of type $\Gamma \rightarrow \Theta \mid A$, 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 is a function from the interpretations of Γ' and Θ' to the interpretation of A. We can create this function from the interpretations of M, ρ, and ϱ by interpreting M as a function from the interpretations Γ and Θ to A and the renamings ρ and ϱ as functions from the interpretation of Γ' or Θ' to the interpretation Γ or Θ respectively. We can then pass the interpretations of Γ' and Θ' to the interpretations of ρ and ϱ and then pass the results of this to the interpretation of M. This is the case because a renaming $\Gamma \rightsquigarrow \Gamma'$ is a list of vars in Γ', so to interpret a renamed term, we must provide an interpretation of all the free variables in Γ'. The type signature of the Call-by-Value interpretation of a renaming is given below.

This whole section can again be shortened, appealing to the known notion of the semantic substitution lemma (substitution is interpreted as composition) and instead explaining how it differs in the case of the dual calculus.

```
ren-int-cbv : ∀ Γ Γ′ → Γ ⇝ Γ′ → (Γ′ ⟦ V x ⟧) → (Γ ⟦ V x ⟧)
```

We can implement this interpretation by induction on Γ. In the base case, with Γ instantiated to $\varnothing$ then the only result of the function can be something of the unit type, this type has only one instance, tt. In the inductive case, with Γ instantiated to Γ , A, we produce a tuple of the recursive interpretation of the contents of Γ and the interpretation of the renamed variable at the head of the renaming.

```
ren-int-cbv ∅ Γ′ ρ γ = tt
ren-int-cbv (Γ , A) Γ′ ρ γ =
  ⟨ ren-int-cbv Γ Γ′ (λ x → ρ (`S x)) γ , ((ρ `Z) ⟦ V V ⟧) γ ⟩
```

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

```
neg-ren-int-cbv : ∀ Θ Θ′ → Θ ⇝ Θ′ → ((`¬ x Θ′) ⟦ V x ⟧) → ((`¬ x Θ) ⟦ V x ⟧)
```

The interpretation of substitutions is implemented in the same way as the interpretation of renamings, the only real difference being that we must supply the other interpreted Context in the pair that makes up the argument of the interpretation of the relevant sequent. The type signatures of the Call-by-Value interpretation of a TermValue and left sequent substitutions are given below.

```
termvalue-sub-int : ∀ Γ Γ′ Θ → Γ –[ (λ Γ A → TermValue Γ Θ A) ]→ Γ′
  → ((‘¬ x Θ) V x) → (Γ′ V x) → (Γ V x)
```

```
coterm-sub-int : ∀ Γ Θ Θ′ → Θ –[ (λ Θ A → A | Γ ⟶ Θ) ]→ Θ′
  → Γ V x → ((‘¬ x Θ′) V x) → ((‘¬ x Θ) V x)
```

3.3.4 The Renaming and Substitution Lemmas

The next two subsections outline my proof that the CPS transformations that I have defined are sound. In the original paper, Wadler asserts that the CPS transformations, as he defines them, preserve reductions. I have taken a slightly different approach to the CPS transformation. Rather then embedding the Dual Calculus into the STLC, I have essentially defined a Denotational Semantics of the Dual Calculus, interpreting its constructs as sets and functions. This means that the notion of preserving reductions from the original paper does not really 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. The type signature of this proposition is given below.

again, looking ahead through the section, it is a loooot more detail than advised, even if it is one of the more complex proofs in the formalisation. You want to strike a balance between showing that the properties are nontrivial, without bombarding the reader with lemma after lemma that they won't really be able to fully appreciate anyway. I think your best bet might be to go through the cases of sub-lemma-coterm and maybe going down one level of lemmas for the variable and abstraction cases. It shouldn't end up being too long (certainly not 6 pages), but even if word count won't end up being a concern, the examiners will not want to read through the intricate details of a complex proof, no matter how important it is.

this explanation should go with the soundness statement in the next section

```
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
```

Before we set about proving this, however, we must prove two important lemmas, essentially stating that the behaviour of the renaming and substitution interpretations is as described in the previous section. To state this explicitly, we wish to prove that the interpretation of a renamed/substituted sequent applied to a pair of interpreted Contexts is equivalent to the interpretation of the original sequent applied to a pair made up of the interpreted renamings/substitutions applied to the interpreted Contexts. I call these the renaming and substitution lemmas, we will tackle the renaming lemma first.

give details of the substitution lemma instead

The renaming lemma is actually four mutually inductive propositions, one for each type of sequent, and one for TermValues, the type signatures are given below.

```
ren-lemma-term : ∀ {Γ Γ′ Θ Θ′ A}
  (M : Γ ⟶ Θ ∣ A) (s : Γ ⇝ Γ′) (t : Θ ⇝ Θ′)
  (γ : Γ′ ^V x) (θ : ‘¬ x Θ′ ^V x) (k : ((‘¬ A) ^V T))
  → (rename-term s t M ^V L) ⟨ γ , θ ⟩ k ≡ (M ^V L) ⟨ ren-int-cbv Γ Γ′ s γ , neg-ren-int-cbv Θ Θ′ t θ ⟩ k

ren-lemma-termvalue : ∀ {Γ Γ′ Θ Θ′ A}
  (V : TermValue Γ Θ A) (s : Γ ⇝ Γ′) (t : Θ ⇝ Θ′) (γ : Γ′ ^V x) (θ : ‘¬ x Θ′ ^V x)
  → (⟨ rename-term s t (proj₁ V) , value-rename s t (proj₂ V) ⟩ ^V L V) ⟨ γ , θ ⟩
    ≡ (V ^V L V) ⟨ ren-int-cbv Γ Γ′ s γ , neg-ren-int-cbv Θ Θ′ t θ ⟩

ren-lemma-coterm : ∀ {Γ Γ′ Θ Θ′ A}
  (K : A ∣ Γ ⟶ Θ) (s : Γ ⇝ Γ′) (t : Θ ⇝ Θ′) (γ : Γ′ ^V x) (θ : ‘¬ x Θ′ ^V x) (k : A ^V T)
  → (rename-coterm s t K ^V R) ⟨ γ , θ ⟩ k ≡ (K ^V R) ⟨ ren-int-cbv Γ Γ′ s γ , neg-ren-int-cbv Θ Θ′ t θ ⟩ k

ren-lemma-statement : ∀ {Γ Γ′ Θ Θ′} (S : Γ ⟼ Θ) (s : Γ ⇝ Γ′) (t : Θ ⇝ Θ′) (γ : Γ′ ^V x) (θ : ‘¬ x Θ′ ^V x)
  → (rename-statement s t S ^V s) ⟨ γ , θ ⟩ ≡ (S ^V s) ⟨ ren-int-cbv Γ Γ′ s γ , neg-ren-int-cbv Θ Θ′ t θ ⟩
```

As is a common theme, the interesting cases are for variables, covariables, variable abstraction, and covariable abstraction; the other cases are relatively simple induction, requiring use of congruence so that we can appeal directly to the inductive hypothesis.

The variable case require us to prove a slightly different proposition to the original: that the interpretation of a renaming applied to a variable applied to an interpreted context is equivalent to the interpretation of that variable applied to the interpreted renaming applied to the same renamed context. The type signature of this is given below.

We can prove this through simple induction. The base case holds definitionally so can be proved with refl. In the inductive case we instantiate x to `S x and then appeal directly to the inductive hypothesis, using the function ren-skip, which removes the last term from a renaming map.

```
ren-lemma-var `Z ρ γ = refl

ren-lemma-var (`S x) ρ γ = ren-lemma-var x (ren-skip ρ) γ
```

We prove a dual proposition for covariables in the same way.

```
ren-lemma-covar : ∀ {Θ ΘI A} (α : Θ ∋ A) (ρ : Θ ⇝ ΘI) (θ : `¬ x ΘI V x) (k : A VT)
  → (Γ∋A⇒¬Γ∋¬A (ρ α) VV) θ k ≡ (Γ∋A⇒¬Γ∋¬A α VV) (neg-ren-int-cbv Θ ΘI ρ θ) k
```

Once these propositions are proved we can prove the variable and covariable cases easily.

```
ren-lemma-term (` x) s t γ θ k = cong k (ren-lemma-var x s γ)


ren-lemma-termvalue ⟨ ` x , V-var ⟩ s t γ θ = ren-lemma-var x s γ


ren-lemma-coterm (` α) s t γ θ k = ren-lemma-covar α t θ k
```

The variable abstraction case is slightly more complex, so I will present it here in equational reasoning form before explaining each of the three gaps we must bridge to prove it.

```
ren-lemma-coterm {Γ}{Γ′}{Θ}{Θ′}{A} (μγ S) s t γ θ k =
  begin
    (rename-coterm s t (μγ S) ᵛᴿ) ⟨ γ , θ ⟩ k
  ≡⟨⟩
    (rename-statement (rename-lift s) t S ᵛˢ) ⟨ ⟨ γ , k ⟩ , θ ⟩
  ≡⟨ ren-lemma-statement S (rename-lift s) t ⟨ γ , k ⟩ θ ⟩
    (S ᵛˢ) ⟨ ⟨ ren-int-cbv Γ (Γ′ , A) (rename-weaken s) ⟨ γ , k ⟩ , k ⟩ , neg-ren-int-cbv Θ Θ′ t θ ⟩
  ≡⟨ cong (λ - → (S ᵛˢ) ⟨ ⟨ - , k ⟩ , neg-ren-int-cbv Θ Θ′ t θ ⟩) (weaken-ren-int-cbv-lemma s γ k) ⟩
    (S ᵛˢ) ⟨ ⟨ ren-int-cbv Γ Γ′ s γ , k ⟩ , neg-ren-int-cbv Θ Θ′ t θ ⟩
  □
```

The first step is made through two definitional equalities, the first is the definition of rename-coterm for variable abstraction. We push the renaming into to the $\mu\gamma$ binder, this changes the function to rename-statement and lifts the antecedent substitution s. We can then use the definition of the CPS transformation of right sequents, this destroys the binder, extending the first interpreted Context with the bound variable, and modifies the transformation to a transformation of centre sequents.

The second step is to appeal to the inductive hypothesis, the effect of this should be obvious, however we also take advantage of a definitional equality in the inductive case of ren-int-cbv, this results in the lifting of the renaming becoming a weakening.

The third step we need to make is to demonstrate an equality between the interpretations of between a renaming and a weakened renaming applied to an arbitrary interpreted Context and that same interpreted Context extended by one interpreted Type. The type signature of the lemma spells this out explicitly.

```
weaken-ren-int-cbv-lemma : ∀ {Γ Γ′ A} (ρ : Γ ⇝ Γ′) γ k
  → ren-int-cbv Γ (Γ′ , A) (rename-weaken ρ) ⟨ γ , k ⟩ ≡ ren-int-cbv Γ Γ′ ρ γ
```

We prove this lemma by induction on Γ, the base case holds definitionally. In the inductive case we instantiate Γ to Γ , B, a simple use of congruence allows us to appeal directly to the inductive hypothesis by removing the last variable from the renaming.

```
weaken-ren-int-cbv-lemma {∅} ρ γ k = refl
weaken-ren-int-cbv-lemma {Γ , B}{Γ′}{A} ρ γ k =
  cong (λ - → ⟨ - , (ρ ‘Z ᵛᵛ) γ ⟩) (weaken-ren-int-cbv-lemma {Γ}{Γ′}{A} (ren-skip ρ) γ k)
```

The proof of the above lemma makes proving the variable abstraction case possible, we prove the covariable case in essentially the same way, requiring a dual lemma for the negated renaming interpretation.

```
weaken-neg-ren-int-cbv-lemma : ∀ {Θ Θ′ A} (ρ : Θ ⇝ Θ′) θ k
  → neg-ren-int-cbv Θ (Θ′ , A) (rename-weaken ρ) ⟨ θ , k ⟩ ≡ neg-ren-int-cbv Θ Θ′ ρ θ
```

This completes the proof of the renaming lemma, we now prove the substitution lemma, it is a very similar proof; just made slightly more complex by nature of the implementation of substitutions being slightly more complex.

```
sub-lemma-term : ∀ {Γ Γ′ Θ Θ′ A}
  (s : Γ –[ (λ Γ A → TermValue Γ Θ′ A) ]→ Γ′) (t : Θ –[ (λ Θ A → A | Γ′ ⟶ Θ) ]→ Θ′)
  (M : Γ ⟶ Θ | A) (γ : Γ′ ᵛˣ) (θ : (‘¬ˣ Θ′) ᵛˣ )
  → ((sub-term TermValueKit CotermKit s t M) ᵛᴸ) ⟨ γ , θ ⟩
    ≡ (M ᵛᴸ) ⟨ termvalue-sub-int Γ Γ′ Θ′ s θ γ , coterm-sub-int Γ′ Θ Θ′ t γ θ ⟩
```

```
sub-lemma-coterm : ∀ {Γ Γ′ Θ Θ′ A}
  (s : Γ –[ (λ Γ A → TermValue Γ Θ′ A) ]→ Γ′) (t : Θ –[ (λ Θ A → A | Γ′ ⟶ Θ) ]→ Θ′)
  (K : A | Γ ⟶ Θ) (γ : Γ′ ᵛˣ) (θ : (‘¬ˣ Θ′) ᵛˣ )
  → ((sub-coterm TermValueKit CotermKit s t K) ᵛᴿ) ⟨ γ , θ ⟩
    ≡ (K ᵛᴿ) ⟨ termvalue-sub-int Γ Γ′ Θ′ s θ γ , coterm-sub-int Γ′ Θ Θ′ t γ θ ⟩

sub-lemma-statement : ∀ {Γ Γ′ Θ Θ′}
  (s : Γ –[ (λ Γ A → TermValue Γ Θ′ A) ]→ Γ′) (t : Θ –[ (λ Θ A → A | Γ′ ⟶ Θ) ]→ Θ′)
  (S : Γ ⟼ Θ) (γ : Γ′ ᵛˣ) (θ : (‘¬ˣ Θ′) ᵛˣ )
  → ((sub-statement TermValueKit CotermKit s t S) ᵛˢ) ⟨ γ , θ ⟩
    ≡ (S ᵛˢ) ⟨ termvalue-sub-int Γ Γ′ Θ′ s θ γ , coterm-sub-int Γ′ Θ Θ′ t γ θ ⟩
```

The cases of interest are, again, variables, covariables, and variable and covariable abstractions.

The variable case once again requires the inductive proof of a slightly different substitution lemma. In this case we must demonstrate an equality between the following: the function resulting from the interpretation of the sequent element of the pair that is produced by applying a TermValue substitution to a variable, applied to an interpreted Context; abstracting on the continuation, and then applying this continuation to the interpretation of the original variable, applied to the interpretation of the substitution, applied to the the interpreted Context. The proof of this is essentially the same as the corresponding proof for the renaming lemma, except in the base case we have to invoke the relationship between the CPS transformation of values and terms.

```
sub-lemma-var : ∀ {Γ Γ′ Θ′ A} (s : Γ –[ (λ Γ A → TermValue Γ Θ′ A) ]→ Γ′)
  (x : Γ ∋ A) (γ : Γ′ ᵛˣ) (θ : (‘¬ˣ Θ′) ᵛˣ )
  → (proj₁ (s x) ᵛᴸ) ⟨ γ , θ ⟩ ≡ (λ k → k ((x ᵛⱽ) (termvalue-sub-int Γ Γ′ Θ′ s θ γ)))
```

```
sub-lemma-var s ‘Z γ θ = cps-value (proj₁ (s ‘Z)) (proj₂ (s ‘Z)) ⟨ γ , θ ⟩

sub-lemma-var {Γ}{ΓI}{ΘI} s (‘S x) γ θ =

  sub-lemma-var (sub-skip (λ Γ A → TermValue Γ ΘI A) s) x γ θ
```

There is a dual lemma for covariables, though this involves none of the complexities introduced by TermValues. Therefore its proof exactly corresponds to the corresponding proof of the renaming lemma.

```
sub-lemma-covar : ∀ {ΓI Θ ΘI A} (t : Θ –[ (λ Θ A → A | ΓI ⟶ Θ) ]→ ΘI)
  (α : Θ ∋ A) (γ : ΓI ᵛˣ) (θ : (‘¬ˣ ΘI) ᵛˣ )
  → (t α ᵛᴿ) ⟨ γ , θ ⟩ ≡ (Γ∋A⇒¬Γ∋¬A α ᵛᵛ) (coterm-sub-int ΓI Θ ΘI t γ θ)
```

The variable abstraction case closely resembles the same case for the renaming lemma. Again I present the proof in equational reasoning form and explain the steps we make.

```
sub-lemma-coterm {Γ}{ΓI}{Θ}{ΘI}{A} s t (μγ S) γ θ = ext (λ x →
  begin
    (sub-statement TermValueKit CotermKit
      (sub-lift (TermSubstKit.kit TermValueKit) s)
      (fmap {λ Θ B → B | ΓI ⟶ Θ} {λ Θ B → B | ΓI , A ⟶ Θ} wkΓᶜ t)
      S ᵛˢ) ⟨ ⟨ γ , x ⟩ , θ ⟩
  ≡⟨
    sub-lemma-statement
      (sub-lift (TermSubstKit.kit TermValueKit) s)
      (fmap {λ Θ B → B | ΓI ⟶ Θ} {λ Θ B → B | ΓI , A ⟶ Θ} wkΓᶜ t)
      S ⟨ γ , x ⟩ θ
```

```
  ⟩
  (S ᵛˢ)
    ⟨ ⟨ termvalue-sub-int Γ (Γ′ , A) Θ′ (sub-weaken (TermSubstKit.kit TermValueKit) s) θ ⟨ γ , x ⟩ , x ⟩
    , coterm-sub-int (Γ′ , A) Θ Θ′
      (fmap {λ Θ B → B | Γ′ ⟶ Θ} {λ Θ B → B | Γ′ , A ⟶ Θ} wkΓᶜ t) ⟨ γ , x ⟩ θ ⟩
≡⟨
  cong (λ - → (S ᵛˢ) ⟨ ⟨ - , x ⟩ , coterm-sub-int (Γ′ , A) Θ Θ′
  (fmap {λ Θ B → B | Γ′ ⟶ Θ} {λ Θ B → B | Γ′ , A ⟶ Θ} wkΓᶜ t) ⟨ γ , x ⟩ θ ⟩)
    (weaken-termvalue-sub-int-lemma s γ θ x)
  ⟩
  (S ᵛˢ)
    ⟨ ⟨ termvalue-sub-int Γ Γ′ Θ′ s θ γ , x ⟩
    , coterm-sub-int (Γ′ , A) Θ Θ′
      (fmap {λ Θ B → B | Γ′ ⟶ Θ} {λ Θ B → B | Γ′ , A ⟶ Θ} wkΓᶜ t)
      ⟨ γ , x ⟩ θ ⟩
≡⟨
  cong (λ - → (S ᵛˢ) ⟨ ⟨ termvalue-sub-int Γ Γ′ Θ′ s θ γ , x ⟩ , - ⟩)
    (fmap-coterm-sub-int-lemma t γ θ x)
  ⟩
  (S ᵛˢ) ⟨ ⟨ termvalue-sub-int Γ Γ′ Θ′ s θ γ , x ⟩ , coterm-sub-int Γ′ Θ Θ′ t γ θ ⟩
□)
```

In this proof we omit the first step that takes advantage of the definitional equalities in the definition of subsitution and the CPS transformation, this is possible as the equalities are definitional

so we need not prove them.

The first step we take in this proof is appealing to the inductive hypothesis, again in the same step we use the definitional equality given by the inductive case of the definition of sub-lift. This leaves us with a formulation that resembles our desired answer but requires some simplification of the pair of interpreted Contexts.

The second step simplifies the interpreted antecedent Context by using a lemma relating the interpretation of a weakened substitution to a simpler formulation. This lemma closely resembles the lemma we had to prove for the renaming lemma, we give its type below.

```
weaken-termvalue-sub-int-lemma : ∀ {Γ Γ′ Θ A} (σ : Γ –[ (λ Γ A → TermValue Γ Θ A) ]→ Γ′) γ θ k
  → termvalue-sub-int Γ (Γ′ , A) Θ (sub-weaken (TermSubstKit.kit TermValueKit) σ) θ ⟨ γ , k ⟩
    ≡ termvalue-sub-int Γ Γ′ Θ σ θ γ
```

We prove this by induction on Γ, the base case holds as a definitional equality. In the inductive case we instantiate Γ to Γ , A, the definition of termvalue-sub-int means we must prove the equality of a pair of elements, we do this by proving the equality of each element individually. We prove the equality of the first elements of the pair by appealing to the inductive hypothesis using the sub-skip function. The equality of the second elements of the pair may look complicated but in fact we just appeal to the renaming lemma and various other lemmas to do with renaming.

```
weaken-termvalue-sub-int-lemma {∅} σ γ θ k = refl
weaken-termvalue-sub-int-lemma {Γ , A} {Γ′} {Θ} σ γ θ k = cong₂ ⟨_,_⟩
  (weaken-termvalue-sub-int-lemma (sub-skip (λ Γ A → TermValue Γ Θ A) σ) γ θ k)
  (trans (ren-lemma-termvalue (σ `Z) (rename-weaken id-var) id-var ⟨ γ , k ⟩ θ)
      (cong₂ (λ _1 _2 → (σ `Z ^VLV) ⟨ _1 , _2 ⟩)
        (trans (weaken-ren-int-cbv-lemma id-var γ k) (id-ren γ)) (id-neg-ren θ)))
```

One of these lemmas is familiar to us, as we used it in proving the variable abstraction case of the renaming lemma. The other two lemmas are new and their type signatures are given below, proving them is a simple inductive proof that also makes use of weaken-ren-int-cbv-lemma.

```
id-ren : ∀ {Γ} (γ : Γ ᵛˣ) → ren-int-cbv Γ Γ id-var γ ≡ γ
```

```
id-neg-ren : ∀ {Θ} (θ : (`¬ˣ Θ) ᵛˣ) → neg-ren-int-cbv Θ Θ id-var θ ≡ θ
```

The third and final step of the proof of the variable abstraction case of the substitution lemma simplifies the interpreted succudent Context by appealing to another new lemma. This lemma relates the interpretation of fmap applied to a substitution to a simple formulation. Its type is given below, however the inductive proof of this lemma is incredibly similar to that of the weaken-termvalue-sub-int-lemma so is omitted for brevity.

```
fmap-coterm-sub-int-lemma : ∀ {Γ Θ Θ′ A} (σ : Θ –[ (λ Θ A → A | Γ ⟶ Θ) ]→ Θ′) γ θ k
  → coterm-sub-int (Γ , A) Θ Θ′
      (fmap {λ Θ B → B | Γ ⟶ Θ} {λ Θ B → B | (Γ , A) ⟶ Θ} wkΓᶜ σ)
      ⟨ γ , k ⟩ θ
      ≡ coterm-sub-int Γ Θ Θ′ σ γ θ
```

This completes the proof of the variable abstraction case, the proof of the covariable abstraction case is entirely dual, requiring proofs of the duals of the lemmas I outlined above. The types of these dual lemmas are given below.

```
weaken-coterm-sub-int-lemma : ∀ {Γ Θ Θ′ A} (σ : Θ –[ (λ Θ A → A | Γ ⟶ Θ) ]→ Θ′) γ θ k
  → coterm-sub-int Γ Θ (Θ′ , A) (sub-weaken (CotermSubstKit.kit CotermKit) σ) γ ⟨ θ , k ⟩
      ≡ coterm-sub-int Γ Θ Θ′ σ γ θ
```

```
fmap-termvalue-sub-int-lemma : ∀ {Γ Γ′ Θ A} (σ : Γ –[(λ Γ A → TermValue Γ Θ A)]→ Γ′) γ θ k
  → termvalue-sub-int Γ Γ′ (Θ , A)
    (fmap {λ Γ B → TermValue Γ Θ B} {λ Γ B → TermValue Γ (Θ , A) B} wkΘ^V σ)
    ⟨ θ , k ⟩ γ
    ≡ termvalue-sub-int Γ Γ′ Θ σ θ γ
```

With that, we have proved the substitution lemma.

3.3.5 Proof of Soundness

With the renaming and substitution lemmas proved, we can, at long last, go about proving the soundness of the Call-by-Value CPS transformation. First we prove a lemma asserting the soundness of the CPS transformation over a single-step reduction. The type of this lemma is almost exactly the same as that of the soundness theorem, it is given below.

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

```
S⟶^V T⇒S^V≡T^V : ∀ {Γ Θ} (S T : Γ ⟼ Θ) (c : (Γ ^ox) ^Nx × ‘¬^x (Θ ^ox) ^Nx)
  → S ^s⟶^V T → (S ^V s) c ≡ (T ^V s) 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-value 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 allows us to use a proof of equality in one direction as a proof of the same equality in the opposite direction.

```
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
```

$$((S\ ^{V}\langle\ \langle\ V\ ,\ v\ \rangle\ /\rangle^{s})\ ^{Vs})\ \langle\ \gamma\ ,\ \theta\ \rangle$$

$\equiv\langle\rangle$

could you have a synonym for `sub-statement TermValueKit CotermKit` to make things a bit shorter?

$$\begin{array}{l}
\quad(\text{sub-statement TermValueKit CotermKit} \\
\qquad(\text{add}\ (\lambda\ \Gamma\ A \to \text{TermValue}\ \Gamma\ \Theta\ A)\ \langle\ V\ ,\ v\ \rangle\ \text{id-termvalue})\ \text{id-coterm}\ S\ ^{Vs}) \\
\qquad\langle\ \gamma\ ,\ \theta\ \rangle \\
\equiv\langle\ \text{sub-lemma-statement}\ (\text{add}\ (\lambda\ \Gamma\ A \to \text{TermValue}\ \Gamma\ \Theta\ A)\ \langle\ V\ ,\ v\ \rangle\ \text{id-termvalue})\ \text{id-coterm}\ S\ \gamma\ \theta\ \rangle \\
\quad(S\ ^{Vs}) \\
\qquad\langle\ \text{termvalue-sub-int}\ (\Gamma\ ,\ A)\ \Gamma\ \Theta\ (\text{add}\ (\lambda\ \Gamma\ A \to \text{TermValue}\ \Gamma\ \Theta\ A)\ \langle\ V\ ,\ v\ \rangle\ \text{id-termvalue})\ \theta\ \gamma \\
\qquad,\ \text{coterm-sub-int}\ \Gamma\ \Theta\ \Theta\ \text{id-coterm}\ \gamma\ \theta\ \rangle \\
\equiv\langle\rangle \\
\quad(S\ ^{Vs}) \\
\qquad\langle\ \langle\ \text{termvalue-sub-int}\ \Gamma\ \Gamma\ \Theta\ \text{id-termvalue}\ \theta\ \gamma\ ,\ (\langle\ V\ ,\ v\ \rangle\ ^{VLV})\ \langle\ \gamma\ ,\ \theta\ \rangle\ \rangle \\
\qquad,\ \text{coterm-sub-int}\ \Gamma\ \Theta\ \Theta\ \text{id-coterm}\ \gamma\ \theta\ \rangle \\
\equiv\langle \\
\quad\text{cong}\ (\lambda\ \text{-} \to (S\ ^{Vs}) \\
\quad\langle\ \langle\ \text{termvalue-sub-int}\ \Gamma\ \Gamma\ \Theta\ \text{id-termvalue}\ \theta\ \gamma\ ,\ (\langle\ V\ ,\ v\ \rangle\ ^{VLV})\ \langle\ \gamma\ ,\ \theta\ \rangle\ \rangle\ ,\ \text{-}\ \rangle) \\
\quad(\text{id-coterm-sub}\ \Gamma\ \Theta\ \gamma\ \theta) \\
\quad\rangle \\
\quad(S\ ^{Vs})\ \langle\ \langle\ \text{termvalue-sub-int}\ \Gamma\ \Gamma\ \Theta\ \text{id-termvalue}\ \theta\ \gamma\ ,\ (\langle\ V\ ,\ v\ \rangle\ ^{VLV})\ \langle\ \gamma\ ,\ \theta\ \rangle\ \rangle\ ,\ \theta\ \rangle \\
\equiv\langle\ \text{cong}\ (\lambda\ \text{-} \to (S\ ^{Vs})\ \langle\ \langle\ \text{-}\ ,\ (\langle\ V\ ,\ v\ \rangle\ ^{VLV})\ \langle\ \gamma\ ,\ \theta\ \rangle\ \rangle\ ,\ \theta\ \rangle)\ (\text{id-termvalue-sub}\ \Gamma\ \Theta\ \gamma\ \theta)\ \rangle \\
\quad(S\ ^{Vs})\ \langle\ \langle\ \gamma\ ,\ (\langle\ V\ ,\ v\ \rangle\ ^{VLV})\ \langle\ \gamma\ ,\ \theta\ \rangle\ \rangle\ ,\ \theta\ \rangle \\
\equiv\langle\ \text{sym}\ (\text{cong}\ (\lambda\ \text{-} \to \text{-}\ (\lambda\ x \to (S\ ^{Vs})\ \langle\ \langle\ \gamma\ ,\ x\ \rangle\ ,\ \theta\ \rangle))\ (\text{cps-value}\ V\ v\ \langle\ \gamma\ ,\ \theta\ \rangle))\ \rangle \\
\quad(V\ ^{VL})\ \langle\ \gamma\ ,\ \theta\ \rangle\ (\lambda\ x \to (S\ ^{Vs})\ \langle\ \langle\ \gamma\ ,\ x\ \rangle\ ,\ \theta\ \rangle)
\end{array}$$

□)

The first step of the proof is a simple definitional equality from the definition of $\_^{V}\langle\_/\rangle^{s}$, reducing the statement to the sub-statement form, about which we have proved multiple lemmas.

The second step appeals to the substitution lemma, the result of this step should be familiar.

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

The next two steps of the proofs use two simple lemmas, that essentially state that the interpretation of identity substitution has no effect. Their types are given below though the proofs are omitted.

```
id-termvalue-sub : ∀ Γ Θ γ θ → termvalue-sub-int Γ Γ Θ id-termvalue θ γ ≡ γ
```

```
id-coterm-sub : ∀ Γ Θ γ θ → coterm-sub-int Γ Θ Θ id-coterm γ θ ≡ θ
```

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

The proof of the right sequent substitution case is a simple dual of the proof given below, the only major difference being that we do not have to appeal to cps-value, as such we omit it.

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

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

This proposition can be proved by structural induction on $S \ {}^{s}\!\twoheadrightarrow^{V} \ T$, there are two possible constructors for which we must prove the proposition. The first is the $\Box^{sV}$ providing evidence that $S \ {}^{s}\!\twoheadrightarrow^{V} \ S$ this equality holds definitionally. Note that a '.' appearing before an argument means that the value of this argument is determined by the patterns given for other arguments. The other case is more complex, in which the constructor $\_ \ {}^{s}\!\rightarrow^{V} \langle \_ \rangle \_$ is used to provide evidence that $S \ {}^{s}\!\twoheadrightarrow^{V} \ T$ by transitivity using as evidence that for some S', $S \ {}^{s}\!\rightarrow^{V} \ S'$ and $S' \ {}^{s}\!\twoheadrightarrow^{V} \ T$. We can prove this case by induction on the evidence that $S' \ {}^{s}\!\twoheadrightarrow^{V} \ T$, as well as appealing to the soundness lemma for $S \ {}^{s}\!\rightarrow^{V} \ S'$.

"Multi-step reduction is the reflexive-transitive closure of single-step reduction, so soundness can be proved by repeated applications of the single-step soundness property above."
No need to explain every bit of the code since it is either pretty clear if the reader is familiar with Agda, or pretty cryptic if they are not, even if you give a detailed explanation of the code. Saying what it's doing at a high-level (repeatedly applying the theorem above) will give a clearer picture of what's going on.

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

This completes our proof of the soundness of the Call-by-Value CPS transformation, not 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.

3.4 Duality

This section discusses the duality of the Dual Calculus as we have defined it so far. We first define the dual translation – demonstrating the duality of the syntax of the calculus – followed by the duality of both the operational and denotational semantics.

3.4.1 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.

We define the dual translation of Type and Context first as they are required for the other translations. They are implemented as two mutually recursive functions, there is nothing particularly interesting about their implementation. other than the duality! Definitely include these, at least the type translation

```
_^oT : Type → Type
_^ox : Context → Context
```

Next we define the bijection between variables and covariables. Since the dual translation of a sequent makes the antecedent of the original the succedent of the dual and vise versa, 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, this is implemented trivially.

```
_^oV : ∀ {Γ A} → (Γ ∋ A) → (Γ ^ox ∋ A ^oT)
```

Now we can define the dual translation of sequents, with separate functions for left, right, and centre sequents. These are all exactly as the original paper would suggest they would look, the only thing to note is that Agda requires the implicit Context arguments to be specified for the variable and covariable abstraction cases as it cannot infer them itself.

```
_^os : ∀ {Γ Θ} → (Γ ⟼ Θ) → (Θ ^ox ⟼ Γ ^ox)
```

and these

and mention here (rather than separately at the end) that the translations serve as constructive proofs of the dual derivability theorem which needs not be proved explicitly thanks to intrinsic typing

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

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

The Dual Translation is an Involution

The first proposition that Wadler lays out in the original paper is that this dual translation is an involution, the mathematical proof of which is very simple. The only issue found was 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, one for each dual translation.

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 will give us an error. This error is not for either of the proof cases but for the type signature of the proof itself.

The error given is as follows: Γ != $(\Gamma\ ^{ox})\ ^{ox}$ of type Context when checking that expression x has type $(\Gamma\ ^{ox})\ ^{ox} \ni (A\ ^{oT})\ ^{oT}$. The issue here is that we are trying to prove the equality of two terms, $(x\ ^{o}V)\ ^{o}V$ and x, that have different types, $(\Gamma\ ^{ox})\ ^{ox} \ni (A\ ^{oT})\ ^{oT}$ and $\Gamma \ni A$ and Agda will not accept this, despite the fact that we have proved that these two types are equal in our previous two proofs. This requires us to include a REWRITE pragma, this marks a proven 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" [37]. To solve this issue, which would reoccur with the the proofs for sequents we use the REWRITE pragma just before the variable proof to mark the involution equalities for Types and Contexts as rewrite rules.

The REWRITE is not a requirement but more of a hack to avoid having to do heterogeneous equalities or substs. Say that in principle this can all be done "properly" but would really complicate matters; and even if REWRITES are generally unsafe, we are using them with established properties so logically there is nothing sinful about this. Also something you can discuss as part of the G A P

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

With this rewrite rule we can prove that the dual translation of sequents is also an involution. This is a simple mutually inductive proof so I include only the type signatures. Note that for left sequents the proof is for a left dual translation followed by a right dual translation, this is because a left dual translation takes a left sequent and produces a right sequent (and vise versa for right sequents).

```
[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
```

A Sequent is derivable iff its dual is derivable

The proof that a sequent is derivable iff its dual is derivable is an excellent example of the power of intrinsic typing. 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. This fact, combined with our completed proof that the dual translation is an involution, makes this proof entirely trivial. I include the full code of the proof only for the purposes of demonstrating how trivial it is.

move the explanation up, but don't include the code as it's indeed pretty trivial

```
Γ⟶Θ|A⇒A°|Θ°⟶Γ° : ∀ {Γ Θ A} → (Γ ⟶ Θ | A) → A °T | Θ °x ⟶ Γ °x
Γ⟶Θ|A⇒A°|Θ°⟶Γ° M = M °L

Γ⟶Θ|A⇐A°|Θ°⟶Γ° : ∀ {Γ Θ A} → (A °T | Θ °x ⟶ Γ °x) → Γ ⟶ Θ | A
Γ⟶Θ|A⇐A°|Θ°⟶Γ° M°L = M°L °R

A|Γ⟶Θ⇒Θ°⟶Γ°|A° : ∀ {Γ Θ A} → (A | Γ ⟶ Θ) → Θ °x ⟶ Γ °x | A °T
A|Γ⟶Θ⇒Θ°⟶Γ°|A° K = K °R

A|Γ⟶Θ⇐Θ°⟶Γ°|A° : ∀ {Γ Θ A} → (Θ °x ⟶ Γ °x | A °T) → A | Γ ⟶ Θ
A|Γ⟶Θ⇐Θ°⟶Γ°|A° K°R = K°R °L

Γ⟼⟶Θ⇒Θ°⟼⟶Γ° : ∀ {Γ Θ} → (Γ ⟼⟶ Θ) → Θ °x ⟼⟶ Γ °x
Γ⟼⟶Θ⇒Θ°⟼⟶Γ° S = S °s

Γ⟼⟶Θ⇐Θ°⟼⟶Γ° : ∀ {Γ Θ} → (Θ °x ⟼⟶ Γ °x) → Γ ⟼⟶ Θ
Γ⟼⟶Θ⇐Θ°⟼⟶Γ° S°s = S°s °s
```

3.4.2 The Duality of Operational Semantics

3.4.3 The Duality of Denotational Semantics

I now outline the proof of one instance of the original paper's titular claim, that Call-by-Value is dual to Call-by-Name, in this case for the CPS transformations.

As is becoming familiar we start with proving the statement for Types and Contexts, the proofs are familiar to us so are omitted.

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 $\Gamma\ ^{Vx}$, $\Theta\ ^{Vx}$, and $A\ ^{VT}$, the other with type indexed by $\Gamma\ ^{Nx\ \circ x}$, $\Theta\ ^{Nx\ \circ x}$, and $A\ ^{NT\ \circ x}$.

This proof also requires that we prove a couple of lemmas and add rewrite rules for the equalities they define. The first lemma states simply 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. Note that 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.

go through things again and make sure that these all have to be rewrites and you can't get away with substs

We now prove the duality of the transformations for vars. This proof is understandable, simply invoking ext to consider an arbitrary transformed Context that we would apply the transformation of a var to.

can also be done pointwise on the contexts, like you do in other proofs

```
x^V≡x^oN : ∀ {Γ A} (x : Γ ∋ A) → x ^VV ≡ ((x ^oV) ^NV)
x^V≡x^oN `Z     = refl
x^V≡x^oN (`S x) = ext (λ c → cong (λ - → - (proj₁ c)) (x^V≡x^oN x))
```

Now follows the proof for sequents. Note that the each of the propositions have been proven 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.

case in point

The variable and covariable cases of this proof simply appeal to duality of the var transformation, and we can use clever pattern matching to make all the other cases simple induction. The only slight exceptions to this are the variable and covariable abstraction cases, which require us to extend the transformed context with newly bound variable/covariable. can probably include the proof here, it's not too bad

Chapter 4

Evaluation

4.1 Work Completed

4.1.1 Project Core

4.1.2 Extensions

4.2 Unit Tests for the Dual Calculus Formalisation

4.3 Simulating other Calculi

4.3.1 Simply-Typed Lambda Calculus

4.3.2 Simply-Typed Lambda Calculus with letcont and throw

4.3.3 Lambda-Mu Calculus

Chapter 5

Conclusion

5.1 Overview of Results

5.2 Future Work

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

[4] 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.

[5] 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.

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

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

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

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

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

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

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

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

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

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

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

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

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

[25] 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.

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

[27] 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.

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

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

[30] 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.

[31] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[32] John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

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

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

[35] 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.

[36] 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.

[37] The Agda Team. Rewriting, 2020.

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

[39] 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.

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

Appendix A

Agda Code

Appendix B

Project Proposal