Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Theorem Proving in Classical Logic

Author:
David Davies

Supervisor:
Dr. Steffen van Bakel

Second Marker:
Dr. Nicolas Wu

June 16, 2021

Abstract

It is well known that functional programming and logic are deeply intertwined. This has led to many systems capable of expressing both propositional and first order logic, that also operate as well-typed programs.

What currently ties popular theorem provers together is their basis in intuitionistic logic, where one cannot prove the law of the excluded middle, '$A \vee \neg A$' – that any proposition is either true or false. In classical logic this notion is provable, and the corresponding programs turn out to be those with control operators.

In this report, we explore and expand upon the research about calculi that correspond with classical logic; and the problems that occur for those relating to first order logic. To see how these calculi behave in practice, we develop and implement functional languages for propositional and first order logic, expressing classical calculi in the setting of a theorem prover, much like Agda and Coq. In the first order language, users are able to define inductive data and record types; importantly, they are able to write computable programs that have a correspondence with classical propositions.

Acknowledgements

I would like to thank Steffen van Bakel, my supervisor, for his support throughout this project and helping find a topic of study based on my interests, for which I am incredibly grateful. His insight and advice have been invaluable.

I would also like to thank my second marker, Nicolas Wu, for introducing me to the world of dependent types, and suggesting useful resources that have aided me greatly during this report.

Finally, I'd like to thank my friends and family for supporting me throughout my time at Imperial College.

Contents

1 Introduction

Proof assistants are gaining in popularity. From helping to teach students how to write mathematical proofs[1] to the large scale projects like checking compiler correctness [13], they are seeing more use in the computing world. Proof assistants are programming languages that correspond with a formal logic, and come in different shapes and forms; Coq [72] has a particular focus on the theorem proving aspect where proofs can be written with intuitive tactics, whereas the language Agda [59] is first and foremost a functional programming language, like Haskell.

Under the hood, theorem provers ensure proof correctness by a strong type system. The mainstream languages are all based on so-called *intuitionistic* type theory [46]. They all capitalise on the astonishing fact that functions in a functional programming language correspond to proofs in *intuitionistic* logic; and the types of these functions correspond to the propositions that are derived by said proofs. Under this correspondence, type-checking a function is the same operation as checking a proof of a proposition [77].

Intuitionism inescapably ties these languages to the fact that they are unable to prove a simple logical notion; that any proposition is either true or false. This is known as the *law of the excluded middle* ((LEM)), and it characterises the logic we call *classical logic* [23]. Intuitionistic logic rejects this notion, and instead is based on the idea that any proof must be *constructive*; so a proof of 'A or B; must be constructed from a proof of either A or B – it is not enough to prove it can't be the case that 'A or B' is not true. This notion of constructive is strongly tied with computability, and it was believed that a proof had a correspondence with a function only if the proof is constructive.

Until the 80s, it was believed that classical logic did not have a computational counterpart. This belief was challenged when Griffin [33] discovered that a control operator, similar to the `call/cc` function in Scheme, corresponded directly with an axiom of classical logic; double negation elimination, which states that a proposition being 'not not true' is the same as it being 'true'. This is another axiom that characterises classical logic. This spawned a new area of research – calculi that correspond with classical logic [61, 20, 70].

In this report, we will explore the current state of research into classical calculi, both for propositional and first order logic. We will then discuss how to use these calculi to form the core of useable proof assistants, and evaluate our own implementations of such languages.

1.1 Report Outline

In Chapter 2 we review both propositional logic and first order logic. We also explore different logical systems within propositional and first order logic, formalising the notions of intuitionistic and classical logic. This is achieved by understanding each logic through their natural deduction inference rules.

In Chapter 3 we will give a quick overview of the simply typed λ-calculus, some simple extensions to it, and an algorithm to find the type of terms in the calculus. This will serve as a strong formal foundation for the less well-known $\lambda\mu$-calculus, as well as prepare the reader for our work later in the report.

Chapter 4 will introduce the $\lambda\mu$-calculus, which is one of the classical calculi hinted at in the introduction. This will describe the idea of evaluation contexts and control operator, and we will explore exactly how $\lambda\mu$ relates to classical logic.

Chapter 5 will expand the correspondence to first order logic by introducing dependent types; types that can depend on terms. This will allow us to express much more interesting logical statements. In particular, as mathematics is a first order language, we will be able to express theorems about maths. This chapter will also explore why dependent types and the classical calculi don't play nicely together, and how we are able to get them to work together soundly.

Chapter 6 serves as the centre of our work in classical propositional logic. We will present a sound and complete principal pairing algorithm for the $\lambda\mu$-calculus, and extend it to allow for

[1] For example, the Xena Project https://wwwf.imperial.ac.uk/~buzzard/xena/

sum and product types, and name polymorphic functions. This algorithm is then able to act as the core of a proof assisstant for propositional logic.

In Chapter 7, we expand on the current work into dependently typed classical calculi, by adding the ability to express coproducts, inductive families and records. Importantly, we will also make sure these new types are able to be expressed safely, avoiding the problems that are outlined in Chapter 5.

Implementations of the work in Chapters 6 and 7 are described in Chapter 8. We will focus on some of the design choices made during development, difficulties we came across and a high level overview of the software.

We assess both the theoretical and practical work in Chapter 9. In particular, we will highlight desirable properties of the calculi that we haven't yet proved. We will also discuss what parts of the implementation were not completed, with respect to the theoretical work.

A brief discussion of the ethical issues of this work will be found in Chapter 10.

Finally, we summarise our work in Chapter 11, and outline how both the theoretical and practical work could be developed and improved upon.

1.2 Contributions

The contributions of this report are as follows:

- Define a sound and complete principal pairing algorithm for the $\lambda\mu$-calculus.
- Extend $\lambda\mu$ (with sum and product types) with functions that allow for name polymorphism, along with an associated sound and complete principal pairing algorithm. We call this calculus $\lambda\mu^N$. The calculus and algorithm double up as the theory needed behind a proof assisstant for classical propositional logic.
- Provide a complete implementation of $\lambda\mu^N$, with Haskell-style syntax.
- Extend the work of [53], to define a classical, dependently typed calculus with dependent functions, pairs and coproducts; $\mathbf{ECC}_{\lambda\mu}$. We further expand this calculus to safely allow for inductive data and record definitions.
- Define a bidirectional algorithm for $\mathbf{ECC}_{\lambda\mu}$, and discuss considerations to make the user-level language easier to work with.
- Provide a partial implementation of $\mathbf{ECC}_{\lambda\mu}$. Due to the time constraints, we focused on implementing the aspects of the bidirectional algorithm that are unique to the classical calculi; in particular the checks for when we allow dependent types and control operators to interact.

2 | Logic

Logic is the study of the structure of human reasoning achieved through examining the relationships between formal statements [23, p. 5]. In this chapter we give an overview of propositional and first order logic, and their respective natrual deduction systems and sequent calculi. We also compare minimal, intuitionistic and classical logic.

2.1 Propositional Logic

Propositional logic is a familiar language for reasoning about statements, called *atoms*, that are assumed to be either true or false.

Definition 2.1: Propositional Logic Syntax [23]

Given a countable set of propositional atoms, P, we define $\mathcal{W}(P)$, the set of all propositions (from these atoms) by:

$$\begin{array}{rcll} A, B & ::= & p & \text{Propositional Atom} \\ & | & (A \to B) & A \textbf{ implies } B \\ & | & (A \wedge B) & A \textbf{ and } B \\ & | & (A \vee B) & A \textbf{ or } B \\ & | & (\neg A) & \textbf{not } A \\ & & & \text{where } p \in P. \end{array}$$

We adopt the usual precedence rules of the connectives, and drop brackets where unambiguous to do so. Bottom is taken to mean 'false', and is considered to be separate from the propositional atoms (although it is sometimes also considered an atomic proposition).

The meaning behind this syntax, called the *semantics*, can be understood through *truth tables*, which show the truth value of a compound term depending on its constituents. We use 1 to represent truth, and 2 to represent falsity. One should observe that these values follow from the meanings given to each symbol;

A	B	$A \to B$	$A \wedge B$	$A \vee B$	$\neg A$
1	1	1	1	1	0
1	0	0	0	1	0
0	1	1	0	1	1
0	0	1	0	0	1

Intuitively, this says that $A \wedge B$ is true when A and B are true, $A \vee B$ is true when at least one of A and B is true, and so on.

An important logical symbol we haven't yet mentioned is *bottom*, $\bot$, which can have a different meaning depending on its context. Some systems might use $\bot$ to only represent contradiction; when we have both A and $\neg A$, we can infer $\bot$. In this case, $\bot$ represents logical absurdity, from which we can derive any proposition.

A system might also allow $\bot$ to be used to encode negation, and thus write $\neg A := A \to \bot$. This means that reasoning about '$\neg$' in these systems is subsumed by reasoning about '$\to$' and '$\bot$'. Note that in both these cases $\bot$ is not treated like the other propositional atoms; it cannot appear on the left of an arrow symbol, nor can we have propositions of the form $A \wedge \bot, \bot \wedge A, A \vee \bot$ or $\bot \vee A$. Instead, $\bot$ is purely meant to represent when we have logical conflict.

We can of course allow $\bot$ to be a propositional atom in its own right, and are thus able to reason about propositions like $\bot \to \bot$. In this case, $\bot$ can be seen as an explicit representation of *falsity*, not just conflict.

When looking at logical systems, we can employ syntactic restrictions on the propositions to form logical *fragments*. This amounts to only using certain connectives. An example of this is 'the

implicative fragment of propositional logic', where we only consider propositions of the form 'p' and '$A \to B$'.

2.2 First Order Logic

First order logic lets us reason statements that refer to known (non-logical) objects. It introduces the ideas of 'for all' and 'there exists'; given by the symbols $\forall$ and $\exists$ respectively. The collection of the objects we talk about is known as the *domain of discourse*, and we say these two new symbols *quantify* over this domain. We write $\forall x.A$ to mean for all objects in the domain of discourse, A is a valid proposition. $\exists x.A$ means there exists an object in the domain such that A is a valid proposition.

The key difference between propositional and first order logic is that, in first order logic, the propositions are defined with respect to variables $x, y, z, \ldots$, which are quantified by the $\forall$ and $\exists$ symbols. Thus, the propositions are able to reason about these objects.

First order logic is the logic of mathematics, and we are able to express many familiar mathematical ideas in first order logic. For example, $\forall x.(even(x) \to \text{isInteger}(x/2))$, states that, for any x, if x is even, then it is divisible by two. What's implicit here, is that x is an integer, and that we even have the function '/', or the constant 2. This is handled by having a known *model* that the logic is able to reason about; a known set of function, propositional and constant symbols. We won't explore this further.

2.3 Natural Deduction

Natural deduction is a system for (syntactic) reasoning about propositions that is meant to closely follow how a human would argue about a proof, whilst maintining formalism. It is defined in terms of judgements and inference rules.

Originally defined by Gentzen [31], natural deduction is a formalised system for manipulation of and reasoning about propositions. they are defined by inference rules over judgements. Judgements[1] are of the form:

$$\Gamma \vdash A$$

Where Γ is a set of propositions, A is a proposition, and $\vdash$ is syntactic entailment (derivability). The judgement is read; "from the assumptions in Γ, we can derive A".

Definition 2.2 (Inference Rule)**.**

Inference rules show how we are allowed to derive judgements from others. They are defined by *premises*, which are known judgements; an *inference line*; and a *conclusion*, which is a judgement said to be *derived* from the premises:

$$\frac{A_1 \qquad \cdots \qquad A_n}{B}$$

Where A_i are the premises, n an integer, and B the conclusion.

Natural deduction systems are defined by a countable/finite set of these rules. When comparing two deduction systems X and Y, we say X is *stronger* than Y (and Y is *weaker* than X) if anything derivable in Y is also derivable in X. If both systems can derive all the same propositions, we say they are *equivalent*. Note that X being neither stronger nor weaker than Y does not necessarily mean the two are equivalent[2].

2.3.1 Logical Subsystems

We will compare different symbolic logics by their canonical inference rules. We focus on just the fragment of propositional logic with implication and bottom. Each of the systems remain

[1] In the original presentation by Gentzen [31], natural deduction does not involve the sequent-style judgements we present. However, the systems are equivalent and are defined by the same introduction/elimination rules - the sequent-style system just makes it clearer which assumptions are open.

[2] The reader can convince themselves of this by considering two simple systems, where X can only derive $\vdash A$, and Y can only derive $\vdash B$ (and $A \neq B$).

separate (and have the same distinctions) after adding the syntax and rules for '$\vee$' and '$\wedge$' (but not (LEM)); we omit them from this discussion for clarity.

The inference rules for the different logics are presented in Figure 2.3, and we will discuss the differences between these logics in the next few paragraphs.

Figure 2.3: A Summary of Natural Deduction Rules for Various Implicative Logics

$$\frac{}{\Gamma, A \vdash A}\ (Ax)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to I) \qquad \text{Minimal}$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot E) \qquad \text{Intuitionistic}$$

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ (RAA) \qquad \text{Classical}$$

We also write IPL for intuitionistic propositional logic, CPL for classical propositional logic, and IPL$_{\to}$ for the implicative fragment of intuitionistic propositional logic.

(Ax) is the 'axiom' rule, which states that if you assume a proposition A, then you can certainly derive A to be true. Implication is characterised by its introduction $(\to I)$ and elimination $(\to E)$. $(\to I)$ captures the meaning of implication well; if, assuming A, we can derive B then this can be rephrased as A implies B. $(\to E)$ explains how we can use an implication; if we know B follows from A, and we also know A to be true, then B must also be true [23, p30].

$(\bot E)$, also called *ex falso quodlibet*, as explained by van Dalen, expresses that from absurdity we can derive anything [23, p30]. (RAA), *reductio ad absurdum*, reifies *proof by contradiction* (PbC): if, assuming $\neg A$, we can derive absurdity, or a contradiction, then it must be the case that A holds [23, p30].

Minimal Logic

Definition 2.4: Minimal Logic [3]

Minimal logic is defined by the rules:

$$\frac{}{\Gamma, A \vdash A}\ (Ax) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to I) \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$$

Importantly, in minimal logic there are no rules specific to $\bot$. In fact, any globally fixed atom could be used in place of $\bot$ (including one that is true, or allowed to appear on the left of arrows) when used to define negation [3, p4].

Intuitionistic Logic

Intuitionistic logic (in the implicative fragment) is found by adding the rule $(\bot E)$ to minimal logic. It is based in constructivism, as it is based on Brouwer's principle that the truth of a proposition is given by providing computable evidence [36]. In intuitionism, then, a proof of $A \vee B$ is constructed from a proof of either A or B. There are no *indirect* proofs; so $\vdash \neg\neg A$ doesn't necessarily entail A. Thus, one more often talks about 'provability' of a proposition than its truth/falsity.

Although we won't explore this further, it is worth noting that intuitionistic logic cannot be understood through truth table semantics, as the tables fundamentally rely on the idea of propositions being true or false.

The constructivist philosophy is also seen in first order logic, where an existential must provide a witness (an explicit 'example' that proves the statement). For example; in proving $\exists x.A$, one must find (and construct) such an x; it is not enough to simply show $\neg\forall x.\neg A$.

Definition 2.5: Intuitionistic Logic [3]

The inference rules for intuitionistic logic are given by adding the following rule to those of minimal logic (as defined in 2.4):

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\,(\bot E)$$

Classical Logic

Classical logic is based on the more traditional view of propositions as being either true or false; so for any A, either A or $\neg A$ is true. The semantics of classical propositional logic can be precisely given by the truth tables discussed earlier in this chapter.

When added to intuitionistic logic, (RAA) gives us classical logic. This is because proof by contradiction is a classical notion; showing that $\neg A$ leads to absurdity is the same as proving A, because we must have either $\neg A$ or A.

Subtly different to (RAA) is the inference rule $(\neg\neg E)$, *double negation elimination*. $(\neg\neg E)$ states that, a derivation of $\neg\neg A$ emits a derivation of A. Again, this follows from the truth table semantics; if we know $\neg\neg A$ is true, then we certainly can't have $\neg A$ be true, thus A is true.

Although (RAA) and $(\neg\neg E)$ may seem similar, they are in fact not equivalent [3]. When added to minimal logic, $(\neg\neg E)$ still gives classical logic; but (RAA) does not. (RAA) added to minimal logic is known as minimal classical logic [3], which is strictly weaker than classical logic, stronger than minimal logic, and neither stronger nor weaker than intuitionistic logic (but the two are not equivalent). Further discussion on intermediate logics (those with proving power between that of intuitionistic and classical logic) can be found in [3].

Definition 2.6: Classical Logic [3]

Classical logic is obtained by extending intuitionistic logic (as in 2.5) with the either of the rules:

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\,(RAA) \qquad \text{or} \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,(\neg\neg E)$$

2.3.2 Conjunction and Disjunction

The logical connectives for conjunction, '$\wedge$', and distinction, '$\vee$', have standard inference rules that can be added to any of the above logics. The logics thus created maintain similar distinctions in the presence of these rules. We will give the intuition behind their inference rules, which follow naturally from their meaning.

Conjunction For $(\wedge I)$, to prove $A \wedge B$, we must have first proved both A and B. Conversely, if we already know $A \wedge B$, then we know that this proposition was constructed from a proof of A and B, so we can ask for these proofs individually, giving us the $(\wedge E_i)$ rules.

Definition 2.7: Conjunction [23, p29]

The inference rules for conjunction are:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge I) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,(\wedge E_1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,(\wedge E_2)$$

Disjunction If we have proved A, then we easily know that $A \vee B$ and $B \vee A$ are true for any proposition B; this idea is encapsulated by the $(\vee I_i)$ rules. The $(\vee E)$ rule is perhaps less obvious, as we have a formula C that seems to have nothing to do with the formula $A \vee B$. We call C the

motive [48] for the elimination of $A \vee B$. The idea is that, if we can prove C given either A or B, and we know at least one of A or B is true, then we can prove C.

Definition 2.8: Disjunction [23, p47]

The inference rules for disjunction are:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee I_1) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash B \vee A}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma \vdash A \to C \qquad \Gamma \vdash B \to C}{\Gamma \vdash C}\ (\vee E)$$

There is a further rule for disjunction that only holds in classical logic; the *law of the excluded middle*, (LEM). It encodes the classical point of view that propositions are either true or false, that is, there is no 'middle' truth value.

Definition 2.9: Law of the Excluded Middle [3, p6]

$$\frac{}{\Gamma \vdash A \vee \neg A}\ (\text{LEM})$$

Adding this rule to intuitionistic logic (with disjunction and conjunction), gives classical logic.

2.3.3 First Order Natural Deduction

Just as with the inference rules for conjunction and disjunction, the inference rules for the quantifiers are standard, and when added to the different logical systems, the resulting systems maintain similar distinctions. It's important that we consider $\forall$ and $\exists$ to both be native to the syntax; in [23], $\exists x.A(x)$ is defined as $\neg\forall x.\neg A(x)$. This is in fact only provable in first order classical logic.

The notation $A(x)$ means A is a proposition with a free variable x; $A[t/x]$ means to substitute every (free) occurrence of x in A by t.

For All The intuition behind $(\forall I)$ is well explained by van Dalen [23, p86][3]; 'if an arbitrary object x has the property A, then every object has the property A'. The inversion of this idea explains $(\forall E)$; if we know that for every object x, we have $A(x)$, we can consider $A(t)$ for an arbitrary object t.

Definition 2.10: For All [23, p93]

$$\frac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x.A(x)}\ (\forall I) \qquad \frac{\Gamma \vdash \forall x.A(x)}{\Gamma \vdash A[t/x]}\ (\forall E)$$

Where x doesn't occur free in any proposition in Γ.

Existence If we can find an object x such that $A(x)$ is true, then we can say $\exists x.A(x)$; this explains the rule $(\exists I)$. $(\exists E)$ is similar to $(\vee E)$ in that it needs a *motive*, B. The right hand premise says that, if we assume $A(x)$, with the free variable x, to be true, then we can derive B. As the left premise has shown we can find such an x, we can thus conclude B.

[3] For consistency with the inference rules presented we use the propositional symbol A; in the original quote, van Dalen uses the symbol φ.

Definition 2.11: Existential [23, p93]

$$\frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x.A(x)}\ (\exists I) \qquad \frac{\Gamma \vdash \exists x.A(x) \qquad \Gamma, A(x) \vdash B}{\Gamma \vdash B}\ (\exists E)$$

2.4 Sequent Calculus

The sequent calculi [57] are an alternative approach for systematic reasoning. Gentzen [31] originally introduced the sequent calculi as a means to prove certain results about natural deduction and proof reductions, but they are a valid logical formalism in their own right. Where natural deduction rules are about the *introduction* and *elimination* of connectives in the conclusions, the sequent calculus rules concern only the introduction of connectives in both the premises and conclusions.

Unlike natural deduction, sequent calculi allow for multiple conclusions. This means we reason about judgments of the form:

$$A_1, \ldots, A_n \vdash B_1, \ldots, B_m$$

Where

- $A_1, \ldots, A_n$ are the *antecedents*; representing the conjuction of assumptions A_1 through A_n - i.e. that all of $A_1, \ldots, A_n$ are assumed to be true.
- $B_1, \ldots, B_m$ are the *succedents* or *conclusions*; representing the disjunction of conclusions - i.e. that at least one of $B_1, \ldots, B_m$ must be true. These can be seen as the 'open cases' of the deduction.

2.4.1 Structural Rules

Sequent Calculi also have explicit structural rules that describe how the structural properties of the sequents themselves relate to their meaning [28]. These involve: *weakening*, where we make an extra assumption (WL) or allow an extra conclusion (WR); *contraction*, where, if we have assumed the same proposition twice, it has the same meaning as assuming it only once (CL), and similar for deriving a proposition twice (CR); and *exchanging*, concerning the ordering of the antecedents and succedents within an individual sequent. The exchange rules are not needed if we consider antecedents and succedents to be finite multisets instead of ordered lists [57]. We take this approach moving forward.

2.4.2 LK and LJ

Definition 2.12 (LK, The Classical Sequent Calculus [31, 28])**.**

The inference rules for the classical sequent calculus can be found in Figure 2.14.

We explain some of the new inference rules.

Top and Bottom

In the sequent calculi, we wish to reason explicitly about both truth, $\top$, and falsity, $\bot$. What we can know for sure about $\top$, is that it is always true, so it is always a valid conclusion – this is represented by the rule ($\top R$). As for the ($\bot L$), we know that if we assume falsity, we have absurdity and we can then certainly derive anything.

Cut

The *Cut* rule lets you prove an intermediate proposition once, and then assume that proposition in the derivation of the main proof [28]. This saves us from having to rewrite a proof for the intermediate proposition A each time we wish to use it in the proof. This rule represents a very common idea in proofs; that of a lemma. For example, in mathematics, we might wish to use the Pythagoras Theorem to prove something about some shapes, and apply the theorem many times – it'd be very annoying to have to write the proof for the theorem each time we wish to use it.

Negation

The derived rules $(\neg R)$ and $(\neg L)$ can be found as follows (by using the definition of $\neg A := A \to \bot$);

$(\neg R)$ can be derived:

$$\dfrac{\dfrac{\Gamma, A \vdash \Delta}{\Gamma, A \vdash \bot, \Delta}\,(WR)}{\Gamma \vdash \neg A, \Delta}\,(\to R)$$

$(\neg L)$ is derived by:

$$\dfrac{\Gamma \vdash A, \Delta \qquad \dfrac{}{\bot \vdash \varnothing}\,(\bot L)}{\Gamma, \neg A \vdash \Delta}\,(\to L)$$

Definition 2.13 (LJ, The Intuitionistic Sequent Calculus [31])**.**

The sequent calculus for intuitionistic logic can be obtained from LK by restricting all rules to only allow one succedent. As argued in [57], this can be relaxed to only restricting $(\to R)$ to single succedents.

Figure 2.14: The Classical Sequent Calculus, LK [28]

Structural Rules

$$\dfrac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,(CL) \qquad \dfrac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\,(CR)$$

$$\dfrac{\Gamma, A \vdash \Delta}{\Gamma, A, B \vdash \Delta}\,(WL) \qquad \dfrac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A, B}\,(WR)$$

Core Rules

$$\dfrac{}{A \vdash A}\,(Ax) \qquad \dfrac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,(Cut)$$

Logical Rules

$$\dfrac{}{\Gamma, \bot \vdash \Delta}\,(\bot L) \qquad \dfrac{}{\Gamma \vdash \top, \Delta}\,(\top R)$$

$$\dfrac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \to B \vdash \Delta, \Delta'}\,(\to L) \qquad \dfrac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\,(\to R)$$

$$\dfrac{\Gamma \vdash A, \Delta \qquad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \wedge B, \Delta, \Delta'}\,(\wedge R) \qquad \dfrac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,(\wedge L_1) \qquad \dfrac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,(\wedge L_2)$$

$$\dfrac{\Gamma, A \vdash \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \vee B \vdash \Delta, \Delta'}\,(\vee L) \qquad \dfrac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,(\vee R_1) \qquad \dfrac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,(\vee R_2)$$

Derived Rules

$$\dfrac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,(\neg L) \qquad \dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,(\neg R)$$

3 | λ-Calculus

In this chapter we give a brief overview of the lambda calculus with simple types, and the definitions we'll need to compare different calculi.

The λ-Calculus is an abstract interpretation of computation, originally defined by Alonzo Church [18]. Being concise, but Turing-Complete [73], it is a great calculus in which to reason about abstract machines. This exposition of the λ-Calculus is based on the lecture notes from the C382 (now COMP60023) Type Systems for Programming Languages course taught at the Department of Computing, Imperial College London [8].

3.1 The Untyped λ-Calculus

Definition 3.1: λ-terms [8]

For a set of valid variables $x, y, z, \ldots$, we define λ-terms by:

$$\begin{aligned} M, N ::= \;& x && \text{(variable)} \\ \mid\; & (\lambda x.M) && \text{(abstraction)} \\ \mid\; & (MN) && \text{(application)} \end{aligned}$$

As usual, we write $\lambda xy.M$ to mean $\lambda x.(\lambda y.M)$, and we drop brackets when unambiguous to do so, with application being left associative.

Variables can be *free* or *bound* in a λ-term [8]. A variable x is *bound* in a term N if x appears in a subterm of $\lambda x.M$, where $\lambda x.M$ is a subterm of N. If a variable x appears in a term not under a binding, it is said to be *free*. Note that a variable can be both free and bound at the same time in a term, for example in $(x)(\lambda x.x)$.

Definition 3.2: Term Substitution [8]

$M[N/x]$, the substitution of a variable x by a term N in a term M, is defined by recursion on the structure of M:

$$\begin{aligned} x[N/x] &= N \\ y[N/x] &= y && y \neq x \\ (\lambda y.M)[N/x] &= \lambda y.(M[N/x]) \\ (LM)[N/x] &= (L[N/x])(M[L/x]) \end{aligned}$$

With x not bound in M.

Variable capture occurs in a substitution $M[N/x]$, when N contains a free variable that is bound in a scope x is in [8]. For ease of reasoning about λ-terms, we adopt *Barendregt's Convention* [11], in which we assume that bound and free variables are always different (and re-labelling of variables is done implicitly when needed).

β-reduction defines the computation of the λ-calculus; substituting a bound variable for an argument.

Definition 3.3: β-reduction [8]

We introduce:

1. The single step β-reduction, $\rightarrow_\beta$, is defined by:

$$(\lambda x.M)N \rightarrow_\beta M[N/x]$$

along with the 'contextual closure' rules,

$$M \to_\beta N \implies \begin{cases} \lambda x.M \to_\beta \lambda x.N \\ LM \to_\beta LN \\ ML \to_\beta NL \end{cases}$$

2. The reflexive, transitive closure of β-reduction is defined as $\to_\beta^*$, satisfying:

$$\begin{aligned} M \to_\beta N &\implies M \to_\beta^* N \\ M &\to_\beta^* M \\ M \to_\beta^* N \;\&\; N \to_\beta^* L &\implies M \to_\beta^* L \end{aligned}$$

3. $=_\beta^*$ is defined by adding symmetry to $\to_\beta^*$, i.e. $M =_\beta^* N \implies N =_\beta^* M$ and $M \to_\beta^* N \implies M =_\beta^* N$. This relation can be seen as 'executing the same function'.

A useful property is that of *confluence*,

Definition 3.4 (Confluence [67, 40])**.**

1. $M \to_\beta^* N$ and $M \to_\beta^* P \implies$ there is a term Q such that $N \to_\beta^* Q$ and $P \to_\beta^* Q$.

2. $M =_\beta^* N$ and $M \to_\beta^* P \implies$ there is a term Q such that $P \to_\beta^* Q$ and $N \to_\beta^* Q$

This is also known as the Church-Rosser Property.

The *values* of the λ-calculus are the terms of the form x or $\lambda x.M$ [65, p127], and we often write V to denote a value.

Confluence expresses that different reductions from the same original term can eventually be 'joined' [8, p6]. We call terms of the form $(\lambda x.M)N$ *redexes* [8, p5]. A term is said to be in *normal form* when it contains no redexes [8, p7].

3.2 The Simply Typed λ-Calculus

The simply typed λ-calculus (STLC), also known as the Curry type system [12], introduces types to the λ-calculus. The typing system precisely follows the syntactic structure of λ-terms, and provides great insight into how terms operate and interact. For all type systems moving forward, we assume there is a countable set of atomic types, which is ranged over by φ.

Definition 3.5: Curry Types [12, 8]

We define the set of Curry types to be:

$$A, B ::= \varphi \mid A \to B$$

$A \to B$ represents functions that take a argument of type A and return a function of type B. $M : A$ means that a term M has type A or, alternatively, M inhabits the type A. M is said to be the *subject*, and A the *predicate* of the *statement* $M : A$.

A *context* is a (partial) mapping of variables to types, of the form $x : A$, that we call *type assignments*, or *statements* [8, p11]. We write $\Gamma, x : A$ to mean $\Gamma \cup \{x : A\}$, with the restriction that if Γ already contains an assignment for x, then that assignment must also be $x : A$.

Derivations of type assigments are presented with judgements of the form, $\Gamma \vdash M : A$, that read; 'from the context Γ, we can derive that M has type A'.

Definition 3.6: Curry Type Assignment [12, 8]

$$\frac{}{\Gamma, x : A \vdash x : A}\,(Ax) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\,(\to I) \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\,(\to E)$$

If we reduce a term, we expect it to have the same type. In particular we would like $(\lambda x.M)N$ and $M[N/x]$ to have the same types. This is property is known as *subject reduction*:

Theorem 3.7 (Subject Reduction). [8, p12] $\Gamma \vdash M : A$ *and* $M \to_\beta^* N \implies \Gamma \vdash N : A$

3.3 Principal Types

For a computer to be able to find the type of a particular term M, we need to encode the type system into an algorithm. As (infinitely) many types can be given to any typeable term, it makes sense to try and find a 'most general' type to give M [8, p14]; that is, a type A subsumes all other types that can be assigned to the given term. If the term contains free variables, we would also like to know the most general types that those variables would need; which gives us a most general context Γ allowing for $\Gamma \vdash M : A$ (where Γ contains exactly the free variables of M). This is known as the 'principal pair' of M; a pair containing the type A and context Γ [8, p14].

To be able to relate different types to each other, we will have to be able to substitute type variables by types, using type substitutions.

Definition 3.8: Type Substitution [8, p14]

A *type substitution*, S, is a partial mapping from type variables to types, which is defined on only finitely many type variables. We write

$$\varphi \mapsto A$$

to represent the substitution that, when applied to φ, returns the type A.
The substitution is defined by recursion on the structure of types;

$$\begin{array}{llcl} (\varphi \mapsto C) & \varphi & = & C \\ (\varphi \mapsto C) & \psi & = & \psi, \text{ if } \psi \neq \varphi \\ (\varphi \mapsto C) & (A \to B) & = & ((\varphi \mapsto C)A) \to ((\varphi \mapsto C)B) \end{array}$$

Substitutions can be applied to contexts by $S\Gamma = \{x : SA \mid x : A \in \Gamma\}$, and a principal pair by $S\langle \Gamma, A\rangle = \langle S\Gamma, SA\rangle$.

If there is a substitution S such that $SA = B$, we say B is an *instance* of A [8, p14].

3.3.1 Unification

Unification of types, as defined by Robinson [66], is a procedure that takes as input two types, and returns a substitution that maps both to their smallest common instance [8, p15].

Definition 3.9: Robinson's Unification Algorithm [66][8, p15]

$$\begin{array}{llll} \mathit{unify} & \varphi & \psi & = \varphi \mapsto \psi \\ \mathit{unify} & \varphi & B & = \begin{cases} \varphi \mapsto B & \text{if } \varphi \text{ doesn't occur in } B \\ \text{error} & \text{otherwise} \end{cases} \\ \mathit{unify} & A & \varphi & = \mathit{unify}\ \varphi\ A \\ \mathit{unify} & A \to B & C \to D & = S_2 \circ S_1 \\ & & & \text{where} \\ & & & \quad S_1 = \mathit{unify}\ A\ B \\ & & & \quad S_2 = \mathit{unify}\ (S_1 C)\ (S_1 D) \end{array}$$

The following proposition shows that Robinson's algorithm does find a common instance that is, in a sense, 'smallest' [8, p15].

Proposition 3.10: Unification Theorem [66, p33][8, p15]

For any types A and B, if there is a unifying substitution S_1 such that $S_1 A = S_1 B$, then there are substitutions S_2 and S_3 such that

$$S_2 = \mathit{unify}\ A\ B$$
$$S_1 = S_3 \circ S_2$$

Definition 3.11 (Most General Unifier, Extend)**.**
For S_2 and S_3 defined as in 3.10,

- S_2 is called the *most general unifier* of A and B [66, p33].
- We say that S_3 *extends* S_2 [8, p16].

These definitions allow us to rephrase 3.10 as: 'any unifying substitution of two types must extend their most general unifier'. Proposition 3.10 will be a very helpful fact to use during our proofs of soundness and completeness for the principal pairing algorithms in Chapter 6.

3.3.2 Principal Type

Definition 3.12: Principal Pair, Principal Type [81]

For a term M, the *principal pair*[a] of M is the pair of a context and type, $\langle \Gamma, A \rangle$ such that:

(i) $\Gamma \vdash M : A$

(ii) For any Γ', A satisfying $\Gamma' \vdash M : A'$, there exists a substitution S such that $(S\ \Gamma) \subseteq \Gamma'$ and $(S\ A) = A'$

A is called the *principal type* of M[b].

[a] This is sometimes also called the principal *typing*, but is distinct from the principal *type*. We call it the principal pair for clarity.

[b] There are various similar but distinct definitions of the principal type of a term; some require the term be closed[12, p71], some require it be defined with respect to an environment[24, p43]. The definition given here subsumes both.

3.4 The Curry-Howard Isomorphism

The Curry-Howard Isomorphism [41] is the observation of the correspondence between the typed λ-calculus and logic. More precisely, any derivation in IPL$_\to$ corresponds to a type assigment of some λ-term. The conclusion of the derivation corresponds with the type, the proof itself corresponds to the λ-term. This is summarised succinctly by the slogans "propositions as types" and "proofs as programs" [77].

Comparing the inference rules side by side, it is very easy to see the correspondence;

Comparing STLC and IPL$_\to$

$\dfrac{}{\Gamma, A \vdash A}\ (Ax)$	$\dfrac{}{\Gamma, x : A \vdash x : A}\ (Ax)$
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to I)$	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ (\to I)$
$\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$	$\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\to E)$

Here it is easy to see that assuming a proposition A amounts to 'labelling' it with the variable x [61]. The key is how implication corresponds with functions; an implication $A \to B$ is a procedure that, when supplied a proof of A, it gives a proof of B. A function $A \to B$ is a procedure that, when supplied a term of type A, returns a term of type B [77].

λ-terms can be seen as an encoding of proofs of IPL$_{\to}$, so we can call the λ-calculus a *proof-term syntax* for IPL$_{\to}$.

3.5 Proof-Term Syntax for Intuitionistic Propositional Logic

Currently our calculus lets us only reason about implication; to be able to reason about conjuctions and disjunctions, we must extend our syntax. We follow the propositional section of intuitionistic type theory outlined by [46, 58].

Definition 3.13: Natural Deduction for IPL[69, p27]

$$\frac{}{\Gamma, A \vdash A}\ (Ax) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot E)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to I) \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge I) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge E_1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee I_1) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash B \vee A}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, \vdash A \to C \qquad \Gamma, \vdash B \to C}{\Gamma \vdash C}\ (\vee E)$$

3.5.1 Inhabiting the Logic with Syntax

This explanation is thanks to Wadler [77].

Conjunction [77, p4] We relate conjunction $A \wedge B$ with the product type $A \times B$. A proof of $A \wedge B$ is formed from a proof of A and B, so a term of type $A \times B$ should consist of a term of type A and a term of type B, that is, a pair of the two terms.

Thus we add *pairs*: $\langle M, N \rangle$, and the ability to select the left or right element in a pair by $\pi_1(M)$ and $\pi_2(M)$, respectively, (which also inhabits the $(\wedge E_i)$ rules) with reduction rules;

$$\pi_1 \langle M, N \rangle \to M$$
$$\pi_2 \langle M, N \rangle \to N$$

Disjunction [77, p4] We relate the proposition $A \vee B$ with the (disjoint) sum type $A + B$, which contains elements of either type A or type B. A proof of $A \vee B$ is formed from either a proof of A or a proof of B, so a term of type $A + B$ should be formed from from a term of type A or a term of type B; and tagging the term to indicate if it was of the left or right type. For $(\vee E)$, we must have a syntax that encodes that we have derived A or B, and in both cases, C can be derived.

Thus we add *injections*: $\text{in}_1(M)$ and $\text{in}_2(M)$, that 'inject' M into the left or right of the sum, respectively. For $(\vee E)$, we express that in both cases of A or B C can be derived, by the syntax $\text{case}(M, N, L)$ (where N handles the left case, L the right) , with reductions;

$$\text{case}(\text{in}_1(M), N, L) \to NM$$
$$\text{case}(\text{in}_2(M), N, L) \to LM$$

Ex Falso From absurdity/falsum, we can derive anything. We should have a marker, ε that notes that A was derived from falsity.

3.5.2 STLC$^{+\times}$ for IPL

Definition 3.14: λ-Calculus for IPL[8, p4, 48]

We define the types of λ-calculus with sums and products by:

$$A, B ::= \bot \mid \varphi \mid A \to B \mid A \times B \mid A + B$$

We define the terms by the grammar:

$$\begin{array}{rcllll} M,N,L & ::= & x & \mid & \lambda x.M & \mid \quad MN \\ & \mid & \langle M,N\rangle & \mid & \pi_i(M) & \\ & \mid & \text{case}(M,N,L) & \mid & \text{in}_i(M) & \\ & \mid & \varepsilon(M) & & & \end{array}$$

Definition 3.15: Type Assignments for the λ-Calculus with sums and products [69, chp4]

$$\frac{}{\Gamma, x : A \vdash x : A}\,(Ax) \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash \varepsilon(M) : A}\,(\bot E)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\,(\to I) \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\,(\to E)$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M,N\rangle : A \times B}\,(\times I)$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1(M) : A}\,(\times E_1) \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2(M) : B}\,(\times E_2)$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{in}_1(M) : A + B}\,(+I_1) \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{in}_2(M) : B + A}\,(+I_2)$$

$$\frac{\Gamma \vdash M : A + B \qquad \Gamma, \vdash N : A \to C \qquad \Gamma, \vdash L : B \to C}{\Gamma \vdash \text{case}(M,N,L) : C}\,(+E)$$

Unit Some logics also have truth, $\top$, as an introducable formula, with a rule;

$$\frac{}{\Gamma \vdash \top}\,(\top I)$$

As we know $\top$ is always true, it has a trivial proof; $\langle\rangle$ (the empty tuple). We represent the type of this proof by either; $\langle\rangle : \top$ or $\langle\rangle : \mathbf{1}$, with a typing rule [75, p436];

$$\frac{}{\Gamma \vdash \langle\rangle : \top}\,(\top I)$$

Computationally, it relates to the 'void' type; as it has only one inhabitant. Of course logically, it might not be interesting to assert truth, but computationally, we can use the trivial proof to construct more interesting terms, much in the same way set theory uses the empty set to construct many more interesting sets.

4 | $\lambda\mu$-Calculus

For a while, it was believed that computation would only correspond to intuitionistic logic, due to its constructive nature, and not classical logic.

More recently, however, computational interpretations of classical logic have been found. The link between classical logic and computation was found in *control operators* [33, 61]. Control operators allow terms to manipulate their evaluation context; for an abstract machine this means control over the execution stack (and program state). Felleisen's $\lambda\mathcal{C}$ [30] calculus was devised to formalise some of the control operators of the Scheme [56] language, like the call/cc construct. Griffin observed [33] that the new term constructor $\mathcal{C}$ could be typed by $((A \to \bot) \to \bot) \to A$ – this was the first time a computational meaning had been given to classical logic; opening a new area of research into calculi with control.

4.1 Evaluation Contexts

The evaluation context of a term amounts to a combination of its arguments and the function being applied to the term (and the function being applied that one, and so on). This can also be seen as the state of the execution stack when evaluating at the term.

In a term $MN_1 \dots N_n$, the *evaluation context* of M is the term with a hole $\bullet N_1 \dots N_n$. In a term $V_1(V_2(\dots V_n(MN_1 \dots N_n))\dots)$, where each V_i is a value, the evaluation context of M is $V_1(V_2(\dots V_n(\bullet N_1 \dots N_n))\dots)$. We can define this formally;

Definition 4.1: Evaluation Context [27]

We define *evaluation contexts* by the grammar:

$$C ::= \bullet \mid CM \mid VC$$

Where $\bullet$ represents a 'hole'; where the context is waiting for a function to be applied. We denote the *insertion* of M into the hole of C by $C\{M\}$.

4.2 $\lambda\mu$-Terms

With the exposition of contexts in mind, we now present the $\lambda\mu$-calculus due to Parigot [61];

Definition 4.2: $\lambda\mu$-Calculus Syntax [61]

With *variables* defined by the Latin symbols, $x, y, \dots$; and *covariables*, or *names*, by the Greek symbols $\alpha, \beta, \dots$; we define $\lambda\mu$-terms by:

$$M ::= x \mid \lambda x.M \mid MN \mid \mu\alpha.[\beta]M$$

We often might discuss also the pseudo terms $\mu\alpha.M$ and $[\beta]M$, but note that all well-formed terms must be as above. We say a term M in $[\alpha]M$ is labelled/named by α.

In the term $\mu\alpha.M$, the name α is *bound* over M. The occurrences of a name α are when it appears in the square brackets, $[\alpha]N$. We say a name α is free in a term M if M contains a subterm of the form $[\alpha]N$, and α is not bound in this subterm.

Definition 4.3: $\lambda\mu$ Values [5, p9]

The *values* of the $\lambda\mu$-calculus are defined by

$$V ::= x \mid \lambda x.M \mid \mu\alpha.[\beta]V$$

The idea behind a term $\mu\alpha.M$ is that it passes, or redirects, its arguments to the terms labelled by α. So if M has a subterms of the form $[\alpha]N$, then, when applied to an argument P, $(\mu\alpha.M)P$ will evaluate by sending P to be an argument of each α-named subterm. In this sense, each subterm $[\alpha]N$ is substituted by $[\alpha]NP$. This idea is formalised by the μ-reduction:

Definition 4.4: $\lambda\mu$-Calculus Reductions [5, 61]

The reduction rules for the $\lambda\mu$-calculus are defined as:

$$\begin{array}{rrcl} \textit{logical } (\beta): & (\lambda x.M)N & \to & (M)[N/x] \\ \textit{structural } (\mu): & (\mu\alpha.M)N & \to & \mu\alpha.M\Big[[\alpha](\bullet N)/[\alpha]\bullet\Big] \end{array}$$

Where $M\Big[[\alpha](\bullet N)/[\alpha]\bullet\Big]$ means to replace every subterm of M with the form $[\alpha]L$ by the term $[\alpha](LN)$. This is called *structural substitution*.
We extend the notions of *free* and *bound* variables to covariables, using μ as the binder for covariables, and $[\alpha]$ for their occurrences.
There are two other rules that can be introduced to the calculus, and they are essentially trivial under the computational interpretation;

$$\begin{array}{rrcl} \textit{renaming } (\mu_n): & [\beta](\mu\alpha.M) & \to & M[\alpha/\beta] \\ \textit{erasing } (\eta_\mu): & \mu\alpha.[\alpha]M & \to & M \quad \textit{if } \alpha \textit{ not free in } M. \end{array}$$

With these reductions, the $\lambda\mu$-calculus is known to be confluent [61, 5]. Notice that terms to the left of a μ term, e.g. N in $N(\mu\alpha.M)$ are not passed to the α-named subterms. The calculus can be extended with another operator, or an extra 'left' reduction rule for μ [25, 62], that can handle the left terms. These calculi do not enjoy conluence, so deterministic evaluation can be gained by choosing a particular evaluation strategy, like call-by-value or call-by-name, and modifying the reduction rules to reflect this strategy.

If we restrict our context syntax introduced in 4.1 to $C := \bullet \mid CM$, we could also write μ-reduction as;

$$C\{\mu\alpha.M\} \to \mu\alpha.M\Big[[\alpha](C\{N\})/[\alpha]N\Big],$$

meaning for every subterm of M of the form $[\alpha]N$, we replace N by itself inserted into the context C; $C\{N\}$. In Section 5.3, we will see that viewing μ as able to manipulate its context allows us to use μ and $[\cdot]$ to understand control operators from other calculi.

Example 4.2.1. Here is a simple term that shows the most basic way the context can be passed to a subterm;

$$(\mu\alpha.[\alpha]M)PQ \to_\mu ()\mu\alpha.[\alpha]MP)Q \to_\mu \mu\alpha.[\alpha]MPQ$$

A term that uses the context twice;

$$\Big(\mu\alpha.[\alpha]\big((\mu\delta.[\alpha]x)y\big)\Big)M_1 \dots M_n \to^* \mu\alpha.[\alpha]\big((\mu\delta.[\alpha](x\ M_1 \dots M_n))y\ M_1 \dots M_n\big)$$

Example 4.2.2. [61] Consider the term;

$$P := \lambda y.\mu\alpha.[\alpha]y\lambda x.\mu\delta.[\alpha]x$$

When applied to $M, N_1, \ldots, N_n$, P reduces as follows;

$$\begin{aligned}
\big(\lambda y.\mu\alpha.[\alpha](y\lambda x.\mu\delta.[\alpha]x)\big)MN_1\ldots N_n &\to \big(\mu\alpha.[\alpha]M(\lambda x.\mu\delta.[\alpha]x)\big)\,N_1\ldots N_n \\
&\to \Big(\mu\alpha.[\alpha]\big(M\ (\lambda x.\mu\delta.[\alpha]x\ N_1)N_1\big)\Big)\,N_2\ldots N_n \\
&\to^* \mu\alpha.[\alpha]\Big(\big(M\ (\lambda x.\mu\delta.[\alpha]x\ N_1\ldots N_n)\big)\,N_1\ldots N_n\Big)
\end{aligned}$$

Where δ is not free in M or any N_i. P has similar behaviour to the call/cc operator of Scheme [61].

When M is being evaluated, its first argument is $Q := \lambda x.\mu\delta.[\alpha](x\ N_1\ldots N_n)$, and it is then given the other arguments $N_1\ldots N_n$. Q as an argument to M effectively gives M control over the context of the original term:

> Noting that δ doesn't occur free in M, thus no subterms of M are named by δ; at any point, M can feed an argument to Q, which causes the current context to be redirected by $\mu\delta$, effectively throwing away the current context.
>
> M is then able to interact with the original context $\bullet N_1\ldots N_n$.

As δ doesn't occur in any subterm, the context it appears in is essentially thrown away. Thus Q is said to *abort* from its current context.

4.3 Type Assignments for the $\lambda\mu$-Calculus

4.3.1 Types and Contexts

Of course, for a logical correspondence, we'll need to be able to assign types to the calculus. We keep the same set of simple types, $A, B ::= \varphi \mid A \to B$.

The key idea behind the typing is in the intuition behind the μ reductions. Consider a term $\mu\alpha.M$ containing a subterm $[\alpha]N$. When applied to an argument P, $\mu\alpha.M$ will reduce by inserting P after N; i.e. $[\alpha]N \to [\alpha]NP$. This means, if $N : A \to B \to C$, that we must have $P : A$, else the subterm NP would be ill-typed. Applying to a second argument Q, through similar reasoning we can see that we must have $Q : B$. As $\mu\alpha.M$ is a term that we want to be well typed when applying to P and Q, its type should match that of N, thus we get $\mu\alpha.M : A \to B \to C$.

More generally, a term $\mu\alpha.M$ has the same type as the subterms labelled by $[\alpha]$. This is well described by de Groote [27][1],

> "…in a $\lambda\mu$-term $\mu\alpha.M$ of type $A \to B$, only the subterms named by α are *really* of type $A \to B$; therefore, when such a term is applied to an argument, this argument must be passed over to the subterms named by α."

4.3.2 Type Assignments

As variables and covariables represent different sorts of objects, we split their typings into *contexts*, Γ, and *co-contexts/conclusions*, Δ.

Our typing judgements are of the form $\Gamma \vdash M : A \mid \Delta$, where $M : A$ is called the *active conclusion*.

Definition 4.5: Type Assignments for $\lambda\mu$-Calculus[61, 70]

The types of the $\lambda\mu$-calculus are defined as;

$$A, B ::= \bot \mid \varphi \mid A \to B \quad (A \neq \bot)$$

With type assignments:

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\,(Ax)$$

$$\frac{\Gamma, x : A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta}\,(\to I) \qquad \frac{\Gamma \vdash M : A \to B \mid \Delta \qquad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}\,(\to E)$$

[1] As the author orignally found referenced in [7, p.34]

$$\frac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash [\alpha]M : \bot \mid \alpha : A, \Delta}\ (name) \qquad \frac{\Gamma \vdash M : \bot \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.M : A \mid \Delta}\ (\mu)$$

This presentation is quite different from Parigot's original presentation [61]. The original system is 'multiplicative' in its contexts and co-contexts, which means that contractions and weakenings of the context are needed. Notably, the (Ax) rule is of the form $x : A \vdash x : A$, which means when typing the closed term $\lambda x.\lambda y.y$, as x will appear in the context when typing $\lambda y.y$, we must allow the weakening rule for the context (to remove x from the context). The presence of the structural rules aren't immediately obvious, as they are left implicit as a comment, rather than explicit rules of the type system [61]. To avoid this complication, we allow the 'shared' (co-)contexts that we have used throughout this report, and allow open assumptions and conclusions in the (Ax) rule (like in STLC).

In Parigot's original type assignments for $\lambda\mu$, $\bot$ was not included in the set of types [61]. Instead, the pseudo-terms of the form $[\alpha]M$, called *commands* [5], had no active type; which in type derivations would be written $\Gamma \vdash [\alpha]M \mid \Delta$. This means we are not able to have a term with a negated type, as there is no $\neg$ symbol for types, nor can we use the encoding $A \to \bot$. By using our definition of types in 4.5, which makes $\bot$ a type that is only allowed to appear on the right of arrows, we are able to represent negation and $(\neg I)$ – but not $(\neg E)$, as will be explained in the next few paragraphs.

To allow for reasoning about negation and absurdity, Parigot proposes to extend the definition of types with $\bot$ [61], not allowing it to appear on the left of arrows; negation is then represented by $A \to \bot$. It is not clear in their work if we can type variables by $\bot$, for example if we allow $x : \bot \vdash x : \bot$ – we proceed with this discussion assuming this is not the case. In this sense, $\bot$ is interpreted as conflict. Then, if we can derive a term to have type $\bot$, we have specific rules when activating and passivating said term – that the covariables with type $\bot$ are not mentioned in the co-context [61];

$$\frac{\Gamma \vdash M : \bot \mid \Delta \qquad \gamma \notin \Delta}{\Gamma \vdash [\gamma]M : \bot \mid \Delta}\ (name_{\bot}) \qquad \frac{\Gamma \vdash M : \bot \mid \Delta \qquad \delta \notin \Delta}{\Gamma \vdash \mu\delta.M : \bot \mid \Delta}\ (\mu_{\bot})$$

We see this as an unsatisfactory solution, as the type system loses its explicitness. If we consider a term M conatining a subterm $[\gamma]N$ such that $N : \bot$ and γ is free in M, this solution would let us type $M : A$, by $\varnothing \vdash M : A \mid \varnothing$; which would make it appear that M has no free names.

Instead of having $\bot$ and these two inference rules, Parigot explains we can equivalently add negation of types and inference rules for $(\neg I)$ and $(\neg E)$) [61];

$$\frac{\Gamma, x : A \vdash M \mid \Delta}{\Gamma \vdash \lambda x.M : \neg A \mid \Delta}\ (\neg I) \qquad \frac{\Gamma \vdash M : \neg A \mid \Delta \qquad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash [\gamma]MN \mid \Delta}\ (\neg E)$$

However, this shares the same problem outlined above, as γ again is not mentioned in the conclusions. We will revisit this problem in Section 4.4.2.

4.4 A Proof-Term Syntax for Classical Propositional Logic

The rule names of the type assignments might hint at a natural deduction correspondence, but the multiple conclusions suggest a sequent calculus. In fact, it corresponds with *classical natural deduction*, devised by Parigot in their development of $\lambda\mu$ [61], which is a 'mixture' [9] between the two.

We reason about judgements of the form $\Gamma \vdash A \mid \Delta$, where Γ is a set of propositions, called open assumptions (or premises), Δ a set of propositions, called open conclusions (or succedents), and A the *active* conclusion. As in the sequent calculus, we assume all the propositions of Γ are true, and that at least one of the propositions in α, Δ is derivable given Γ.

Definition 4.6: Classical Natural Deduction [61]

$$\frac{}{\Gamma, A \vdash A \mid \Delta}\ (Ax) \qquad \frac{\Gamma, A \vdash B \mid \Delta}{\Gamma \vdash A \to B \mid \Delta}\ (\to I) \qquad \frac{\Gamma \vdash A \to B \mid \Delta \qquad \Gamma' \vdash A \mid \Delta'}{\Gamma, \Gamma' \vdash B \mid \Delta}\ (\to E)$$

Note it is easy to define rules for $(\neg I)$ and $(\neg E)$ by taking $\neg A = A \to \bot$.

Definition 4.7: Structural Rules [61]

There are implicit structural rules that one can passivate and activate conclusions at any time.

$$\frac{\Gamma \vdash A \mid \Delta}{\Gamma \vdash \bot \mid A, \Delta}\,(passivate) \qquad \frac{\Gamma \vdash \bot \mid A, \Delta}{\Gamma \vdash A \mid \Delta}\,(activate)$$

We also have the $(WL), (WR), (CL), (CR)$ rules from the sequent calculus, where the (WR) rule must weaken from an active $\bot$.

As this deductive system is essentially IPL$_\to$ with multiple conclusions, we can see that the extra strength for classical reasoning must come from allowing these multiple conclusions and their manipulation (as we know (PC) is not derivable in just IPL$_\to$). The μ and $[\cdot]$ operators correspond with these structural rules; activation and passivation.

4.4.1 Translating Derivations

Example 4.4.1. [61] We show a derivation of Pierce's Law, $((A \to B) \to A) \to A$. This implies that our calculus is at least as strong as minimal classical logic [3, 2].

$$\dfrac{\dfrac{}{(A \to B) \to A \vdash (A \to B) \to A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(passivate)}{A \vdash \bot \mid A}\,(WR)}{A \vdash B \mid A}\,(\to I)}{\vdash A \to B \mid A}\,(\to E)}{\dfrac{\dfrac{(A \to B) \to A \vdash A \mid A}{(A \to B) \to A \vdash A}\,(CR)}{\vdash ((A \to B) \to A) \to A}\,(\to I)}$$

We can just follow the rule labels to see how to inhabit this proposition with a term:

$$\dfrac{\dfrac{}{y : (A \to B) \to A \vdash y : (A \to B) \to A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A \mid \alpha : A, \delta : \bot}}{x : A \vdash [\alpha]x : \bot \mid \alpha : A, \delta : \bot}\,(name)}{x : A \vdash \mu\delta.[\alpha]x : B \mid \alpha : A}\,(\mu)}{\vdash \lambda x.\mu\delta.[\alpha]x : A \to B \mid \alpha : A}\,(\to I)}{(A \to B) \to A \vdash y(\lambda x.\mu\delta.[\alpha]x) : A \mid \alpha : A}\,(\to E)}{\dfrac{\dfrac{(A \to B) \to A \vdash [\alpha]y(\lambda x.\mu\delta.[\alpha]x) : A \mid \alpha : A}{(A \to B) \to A \vdash \mu\alpha.[\alpha]y(\lambda x.\mu\delta.[\alpha]x) : A}\,(\mu)}{\vdash \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\delta.[\alpha]x) : ((A \to B) \to A) \to A}\,(name)}\,(\to I)$$

Note that we needed to expand (CR) into the two rules, $(name)$ then (mu), both on the same covariable. This is closely linked with the (η_μ) reduction. Also see that (WR) corresponded to a (μ) rule for a dummy variable δ that doesn't appear in the subterm, allowing us to reason about an arbitrary extra conclusion.

Remark. The inhabiting term is the same as in example 4.2.2; linking us back to the original discovery of classical computation – callcc (or thereabouts).

Example 4.4.2. [61] We show a derivation of double negation elimination, $\neg\neg A \to A$, which implies that the deduction system is as strong as the full classical logic [2, 3].

$$\dfrac{\dfrac{}{\neg\neg A \vdash \neg\neg A}\,(Ax) \qquad \dfrac{\dfrac{}{A \vdash A}\,(Ax)}{\vdash \neg A, A}\,(\to I)}{\dfrac{\neg\neg A \vdash A}{\vdash \neg\neg A \to A}\,(\to I)}\,(\to E)$$

$$\dfrac{\dfrac{\dfrac{}{\neg\neg A \vdash \neg\neg A}\,(Ax) \qquad \dfrac{\dfrac{\dfrac{}{A \vdash A}\,(Ax)}{A \vdash \bot \mid A}\,(passivate)}{\vdash \neg A \mid A}\,(\to I)}{\neg\neg A \vdash \bot \mid A}\,(\to E)}{\dfrac{\neg\neg A \vdash A}{\vdash \neg\neg A \to A}\,(\to I)}\,(activate)$$

However, if we try to follow the derivation to get the inhabiting term, we hit a snag in the last 3 lines. The issue is that we perform an *activate* without first passivating; instead there is a preceeding $(\to E)$. As (μ) corresponds with (*activate*), (*name*) with (*passivate*), we would need to add an extra (*passivate*) between the (*activate*) and the $(\to E)$; this is because μ terms must always have $[\cdot]$ as its immediated subterm. So we will have to expand to the derivation with:

$$\dfrac{\dfrac{\dfrac{\neg\neg A \vdash \bot \mid A}{\neg\neg A \vdash \bot \mid A, \bot}\,(passivate)}{\neg\neg A \vdash A \mid \bot}\,(activate)}{\vdash \neg\neg A \to A \mid \bot}\,(\to I)$$

But we now have a dangling conclusion of $\bot$. Semantically, it is easy to see it is equivalent (as $\bot$ can't be true) to the previous derivation, however we can't syntactically remove the $\bot$; if we tried to μ-abstract it, we would need to choose another covariable to bind to $\neg\neg A \to A$, leaving us with another dangling conclusion.

If we look at the inhabiting term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : \neg\neg A \vdash y : \neg\neg A}\,(Ax) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A \mid \alpha : A, \delta : \bot}\,(Ax)}{x : A \vdash [\alpha]x : \bot \mid \alpha : A, \delta : \bot}\,(name)}{x : A \vdash \mu\delta[\alpha]x : \bot \mid \alpha : A}\,(\mu)}{\vdash \lambda x.\mu\delta.[\alpha]x : \neg A \mid \alpha : A}\,(\to I)}{\neg\neg A \vdash y(\lambda x.\mu\delta.[\alpha]x) : \bot \mid \alpha : A}\,(\to E)}{\neg\neg A \vdash [\alpha]y(\lambda x.\mu\delta.[\alpha]x) : \bot \mid \alpha : A, \gamma : \bot}\,(name)}{\neg\neg A \vdash \mu\alpha.[\alpha]y(\lambda x.\mu\delta.[\alpha]x) : A \mid \gamma : \bot}\,(\mu)}{\vdash \lambda x.\mu\alpha.[\alpha]y(\lambda x.\mu\delta.[\alpha]x) : \neg\neg A \to A \mid \gamma : \bot}\,(\to I)$$

The derivation requires we know a free covariable γ to have type $\bot$. As explained in [70], this corresponds to the fact that the calculus has no explicit $(\bot E)$ rule; this would correspond with allowing to derive A and then contract with the inactive A (or allowing to activate without first passivating).

This begs the question; if we want a full classical logic, how do we handle this free conclusion?

4.4.2 Towards a Complete Logic

Allowing Free Covariables

If we want to remain in the same $\lambda\mu$-calculus, there are two main options. Parigot, in the original presentation of $\lambda\mu$ [61], suggests that we allow these free covariables (with bottom type), but not mention them in the type system. This would mean γ in the example above wouldn't appear in the co-context. Although this gives a complete logical system, the $(\bot E)$ rule is only implicit, so it loses some obvious correspondence with natural deduction. It also isn't very 'clean' to have free covariables about the place (especially from an implementation perspective); a subterm would need a way to make sure the covariables aren't bound in a superterm.

The Top-Level

An alternative approach is suggested by Ariola et al in [3, 2] is to add a constant coterm *top* that has the typing rule:

$$\dfrac{\Gamma \vdash M : \bot \mid \Delta}{\Gamma \vdash [top]M : \bot \mid \Delta}\,(top)$$

With this rule, we are able to derive the $(\bot E)$ rule;

$$\dfrac{\dfrac{\Gamma \vdash M : \bot \mid \Delta}{\Gamma \vdash [top]M : \bot \mid \Delta}}{\Gamma \vdash \mu\delta.[top]M : A \mid \Delta}$$

Where $\delta \notin fn(M)$. We can in fact construct the term, $\mathcal{A} = \lambda x.\mu\delta.[top]x : \bot \to A$ to act as a constructor for the $(\bot E)$ rule:

$$\dfrac{\Gamma \vdash \mathcal{A} : \bot \to A \mid \Delta \qquad \Gamma \vdash M : \bot \mid \Delta}{\Gamma \vdash \mathcal{A}(M) : A \mid \Delta}\,(\to E)$$

The computational interpretation of $\mathcal{A}$ is with Felleisen's 'abort' operator [30], which allows a term to 'throw' to the top-level [33]; that is, to drop the current context, and operate in the 'empty' context (known as the top-level). Any program's evaluation is started in this implicit top-level context that is guaranteed to have type $\bot$ [70, p135]; and any subterm is able to switch to this context. As explained in [3, p21], this is similar to only considering the terms of the form $\mu\gamma.[\gamma]M$, such that γ is typed with the top-level type. This means our computational interpretation of $\bot$ is the type of the top-level. From a computational perspective, Ariola et al. argue:

> "..., the presence of the continuation *top* makes it possible to distinguish between aborting a computation and throwing to a continuation (as aborting corresponds to throwing to the special top-level continuation)" [3]

Expanding the Calculus

The last option is to determine that the $\lambda\mu$-calculus is indeed too weak, so we must add reductions or more operators to achieve completeness. The symmetric calculus [25] adds an extra reduction rule allowing μ to consume context to the left;

$$N(\mu\alpha.M) \to_{\mu'} \mu\alpha.M\left[{}^{[\alpha](N\bullet)}/_{[\alpha]\bullet}\right]$$

Another calculus is $\bar{\lambda}\mu\tilde{\mu}$ [20], which fully embraces the classical sequent calculus. It uses a stratified definition of syntax, to separate them as explicit 'terms', 'coterms' and 'commands', where commands represent subroutines that don't return a value. There is also a new binder, $\tilde{\mu}$, which is similar the the left rule of the symmetric calculus; it binds values into commands.

In the other direction, the $\nu\lambda\mu$ [70] calculus fully embraces natural deduction. There is an explicit negation type, $\neg$, representing continuations, and it allows general terms to appear in $[\bullet]$ (e.g. $[\lambda x.zMx]y$ is a valid term). Negation is introduced to a term by the ν binder, and is eliminated by applying a continuation; $[M]N$. The calculus also collapses the set of variables and covariables to just the latin letters. (μ) is made to explicity represent (RAA).

Both of these calculi are non-confluent, but, as we suggested in 4.2, this problem can be avoided in practice by choosing a deterministic evaluation strategy, so non-confluece is really a non-issue.

4.5 Sums and Products

Adding sums and products to $\lambda\mu$ is almost as simple as just adding the syntax, typing rules and reductions from 3.15. Thus we can expand the definition of $\lambda\mu$ terms with:

$$M,N ::= \ldots \mid (M,N) \mid \pi_i(M) \mid \mathtt{in}_i(M) \mid \mathtt{case}\ M\ \mathtt{of}\ (N_1|N_2)$$

The only decision to be made is if we allow μ reductions for when projections and case analysis interact with μ-bound terms [34]. If we use de Groote's intuition that, interacting with a term $\mu\alpha.M$ is in fact interacting with the subterms named by α [27], we can expand this notion to projections and case elimination: a projection $\pi_i(\mu\alpha.M)$ is in fact projecting the subterms named by α; a case elimination of $\mu\alpha.M$ is in fact performing case analysis on the subterms named by α.

We refer to these rules as (ζ) rules, thanks to [76].

$$\begin{array}{ll} (\zeta_{\pi_i}) & \pi_i(\mu\alpha.M) \quad \to \mu\alpha.M\Big[[\alpha]\pi_i(\bullet)/[\alpha]\bullet\Big] \\ (\zeta_{+}) & \text{case } \mu\alpha.M \text{ of } (N|L) \quad \to \mu\alpha.M\Big[[\alpha]\text{case } \bullet \text{ of } (N|L)/[\alpha]\bullet\Big] \end{array}$$

The main advantage of these reductions is they allow us to extract the subproofs in each structure.

Example 4.5.1. Consider the following term[a], where $M : A, P : A, Q : B$

$$\mu\alpha.[\alpha](M, \mu\delta.[\alpha](P, Q)) : A \times B$$

This term is in normal form. Without the (ζ) rules, we aren't able to get a 'useful' value out of either projection,

$$\pi_1(\mu\alpha.[\alpha](M, \mu\delta.[\alpha](P, Q))) : A \qquad \pi_2(\mu\alpha.[\alpha](M, \mu\delta.[\alpha](P, Q))) : B$$

as these terms are also in normal form. Although the theorem proving aspect is still sound, from a computational perspective we are unable to access the proofs of A and B. The (ζ) rules allow us to access these proofs:

$$\begin{array}{ll} & \pi_1(\mu\alpha.[\alpha](M, \mu\delta.[\alpha](P, Q))) \\ \to & \mu\alpha.[\alpha]\pi_1(M, \mu\delta.[\alpha]\pi_1(P, Q)) \\ \to & \mu\alpha.[\alpha]M \\ \to & M \end{array} \qquad \begin{array}{ll} & \pi_2(\mu\alpha.[\alpha](M, \mu\delta.[\alpha](P, Q))) \\ \to & \mu\alpha.[\alpha]\pi_2(M, \mu\delta.[\alpha]\pi_2(P, Q)) \\ \to & \mu\alpha.[\alpha]\mu\delta.[\alpha]\pi_2(P, Q)) \\ \to & \mu\alpha.[\alpha]\mu\delta.[\alpha]Q \\ \to & \mu\alpha.[\alpha]Q \\ \to & Q \end{array}$$

[a] The term was found as a simplification of Herbelin's term used to show degeneracy of Σ types in the presence of control [39].

4.6 $\lambda\mu$-Calculus as a Natural Deduction System

In [61], Parigot gives an alternate presentation of the classical natural deduction which has only single conclusions. Summers [70] shows how make the same transformation to the type system for $\lambda\mu$, giving us a type system with single conclusions.

Definition 4.8: Alternative Typing for the $\lambda\mu$-Calculus [61, 70]

$$\frac{}{\Gamma, x : A; \neg\Delta \vdash x : A}\,(Ax)$$

$$\frac{\Gamma, x : A; \neg\Delta \vdash M : B}{\Gamma; \neg\Delta \vdash \lambda x.M : A \to B}\,(\to I) \qquad \frac{\Gamma; \neg\Delta \vdash M : A \to B \qquad \Gamma; \neg\Delta \vdash N : A}{\Gamma; \neg\Delta \vdash MN : B}\,(\to E)$$

$$\frac{\Gamma; \neg\Delta \vdash M : A}{\Gamma; \neg\Delta, \alpha : \neg A \vdash [\alpha]M : \bot}\,(name) \qquad \frac{\Gamma; \neg\Delta, \alpha : \neg A \vdash M : \bot}{\Gamma; \neg\Delta \vdash \mu\alpha.M : A}\,(\mu)$$

To bring the calculus closer to natural deduction, Summers also allows μ-binding and naming to be split, so terms of the form $\mu\alpha.M$ and $[\alpha]M$ are syntactically valid for any term M [70]. This allows the typing rules $(name)$ and (μ) to be used separately in derivations. Importantly, this means that some terms can be explicitly typed by $\bot$. As we will see, $\bot$ will represent conflict much more obviously in this system.

In this reformulation, it is clearer that, if we don't allow μ-binding and $[\cdot]$ to be split, the calculus corresponds with minimal classical logic, rather than classical logic. In particular, there is no obvious correspondence with the $(\bot E)$ rule.

We can see that the (μ) rule is similar to (RAA), and $(name)$ to $(\neg E)$. However, the rules a restricted form of their logical counterparts in which the negated premise is restriced to an

axiom, as explained in [70] with the quasi-derivation (note the false '(Ax)' rule used on the left derivation) comparing the two:

$$\dfrac{\dfrac{}{\Gamma;\neg\Delta,\alpha:\neg A\vdash\alpha:\neg A}\,\text{`}(Ax)\text{'} \qquad \Gamma;\neg\Delta\vdash M:A}{\Gamma;\neg\Delta,\alpha:\neg A\vdash[\alpha]M:\bot}\,(\neg E) \qquad \dfrac{\Gamma;\neg\Delta\vdash M:A}{\Gamma;\neg\Delta,\alpha:\neg A\vdash[\alpha]M:\bot}\,(name)$$

As outlined by Summers [70, p93], from a logical perspective, this restriction of negated propositions to axiomatic assumptions seems arbitrary. To fully inhabit $(\neg E)$, we need to be able to use any (non-trivial) proof within the $[\bullet]$ terms. However, the syntax and thus the $(name)$ rule only allows being able to use the 'special' assumptions (greek letters).

A similar restriction to greek variables is seen in the (μ) rule when compared with (RAA); it means we can only use (RAA) on these 'special' assumptions, and not on non-trivial proofs.

This can lead to more cumbersome derivations when compared to, say, $\text{NK}_{\to}$.

Example 4.6.1. In the logical system corresponding to the type system in Definition 4.8, we can prove double negation elimination $(\neg\neg E)$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg A;\varnothing\vdash\neg\neg A}\,(Ax) \qquad \dfrac{\dfrac{\dfrac{}{A;\varnothing\vdash A}\,(Ax)}{A;\neg A\vdash\bot}\,(name)}{\varnothing;\neg A\vdash\neg A}\,(\to I)}{\neg\neg A;\neg A\vdash\bot}}{\neg\neg A;\varnothing\vdash A}\,(RAA)}{\varnothing;\varnothing\vdash\neg\neg A\to A}\,(\to I)$$

Note that we have to use the 'special' assumption of $\neg A$ in the derivation, to be allowed to use the (RAA) rule on it later on.

However, in a system with $(\neg E)$ and no distinguished set of 'special' assumptions, this proof is a lot more simple:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg A\vdash\neg\neg A}\,(Ax) \qquad \dfrac{}{\neg A\vdash\neg A}\,(Ax)}{\neg\neg A;\neg A\vdash\bot}}{\neg\neg A\vdash A}\,(RAA)}{\vdash\neg\neg A\to A}\,(\to I)$$

5 | Dependent Types

Propositional logic is limited in what it can reason about. As mathematics is a first order system, we need to be able to reason about first order logic if we wish to prove theorems about maths. Luckily, the Curry-Howard correspondence expands to first order logic with *dependent types*. In a dependent type system, types can *depend* on terms and, in fact, types *are* terms.

The most popular proof assisstants, like Coq, Lean and Agda, all have type systems descended from a seminal dependent type system, *Intuitionistic Type Theory*, originally defined by Martin Löf [46].

As the name suggests, these dependent type theories are based on first order intuitionistic logic, and thus we don't have the usual classical tautologies, $A \vee \neg A$ or $\neg\forall x.\neg A \to \exists x.A$. The goal of this report is to explore theorem proving in classical logic, we will want to see how these dependent types interact with the control operators that bring the classical power to intuitionistic calculi. In particular, this will let us see if we are able to define a calculus that can reason about first order classical logic.

In this chapter, we will give an exposition of dependent types. We will then observe the problems that occur when trying to mix dependent types with control operators, and how we can remedy them. Finally, we will introduce inductive families, which are a generalisation of Haskell data types that are able to be indexed by values/terms; these allow us to express and reason about more complex mathematical structures, like vectors.

5.1 Intuitionistic Type Theory

This exposition is based on those in [75, p24] and [47, p10].

To let types depend on terms, we will see that terms will have to appear in types. This means we collapse the syntax of types and terms into just terms;

Definition 5.1: Intuitionistic Type Theory Syntax [75]

M, N, A, B	::=	x	Variable
	|	$(x : A) \to B$	Dependent Function Type
	|	$\lambda x : A.M$	Lambda Abstraction
	|	$\texttt{let}\ x = M\ \texttt{in}\ N$	Let
	|	MN	Function Application
	|	$(x : A) \times B$	Dependent Pair Type
	|	(M, N)	Dependent Pair
	|	$\pi_i(M)$	Pair Projection ($i = 1, 2$)
	|	$A + B$	Sum Type
	|	$\text{in}_i(M)$	Sum Injection ($i = 1, 2$)
	|	$\text{case}(M, N, L)$	Case Analysis
	|	$M = N$	Identity Type
	|	$\texttt{refl}$	Reflexivity
	|	$\texttt{subst}\ M\ N$	Substitution
	|	$\bot$	Empty Type
	|	$\mathbf{1}$	Unit Type
	|	$\langle\rangle$	Element of Unit Type
	|	$\mathcal{U}_i$	Type Universe ($i = 0, 1, \dots$)

As we will see, typing this syntax will let us fully inhabit first order intuitionistic logic.

Universes and Type Families If types are now terms, we need to be able to assign types to types. This 'type of types' is known as a *universe*; a type whose elements are types [75]. If all types are contained within one universe, $\mathcal{U}$, we can prove a type theory version of to Russell's paradox [19]

('does the set of all sets contain itself?'). To avoid this, we have a *hierarchy* of universes, $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$, that is *cumulative*, so $A : \mathcal{U}_i \implies A : \mathcal{U}_{i+1}$. Unless relevant, we usually omit the universe level (the subscript), and leave it implicit.

Thus, types are terms whose type is a universe. For example, our simple types from 4.5 can be typed as $A : \mathcal{U}$. We can also have a *family* of types, $B : A \to \mathcal{U}$, which can be seen as a function that, given an $a : A$, returns a type $B(a)$.

Definition 5.2: Types and Universes [75, p434]

$$\frac{}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ (\mathcal{U}\mathrm{I}) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}\ (\mathcal{U}C)$$

Contexts As our types are now more complex, we have to make sure our contexts are well-formed. This amounts to checking that each type in the context is indeed a type, and not just a term.

Definition 5.3: Contexts [75, p432]

$$\frac{}{\varnothing \vdash \text{valid}} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma, x : A \vdash \text{valid}} \qquad \frac{\Gamma, x : A \vdash \text{valid}}{\Gamma, x : A \vdash x : A}\ (Ax)$$

Constants The only difference with the STLC is we add rules to verify that constant types are indeed types (given a valid context)

Definition 5.4: Constant Types [75, p436]

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i}\ (\mathbf{0}) \qquad \frac{\Gamma \text{ valid}}{\Gamma \vdash \mathbf{1} : \mathcal{U}_i}\ (\mathbf{1}) \qquad \frac{\Gamma \text{ valid}}{\Gamma \vdash \langle\rangle : \mathbf{1}}\ (\mathbf{1}I)$$

Sum Types These sum types are non dependent, and are just the STLC typing rules with extra checks for well-formed types. A dependent version can be given (see [75, p436]), but for reasons we will see later in this chapter, we will stick to the more simple version presented by Herbelin in [38].

Definition 5.5: Sum Types [75, p436]

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash M : A}{\Gamma \vdash \text{in}_1(M) : A + B}\ (+I_1) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash M : B}{\Gamma \vdash \text{in}_2(M) : A + B}\ (+I_2)$$

$$\frac{\Gamma \vdash M : A + B \qquad \Gamma \vdash C : \mathcal{U}_i \qquad \Gamma, x : A \vdash N : C \qquad \Gamma, y : B \vdash L : C}{\Gamma \vdash \text{case}(M, x.N, y.L) : C}\ (+E)$$

Equality We have two kinds of equality in ITT. *Definitional* equality, $M \equiv N$, denotes when M and N are equal under β-equivalence (and α-equivalence); this is a meta-theoretic property that can't be directly reasoned about within the type system. On the other hand, *propositional* equality, $M =_A N$, is a type that identifies two equal terms M and N under the type A. Crucially, this allows us to reason about equality under a type within the type system.

Definition 5.6: Equality [75, p437]

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash \mathtt{refl} : M =_A M}\ (\mathtt{refl}) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash M =_A N}\ (=)$$

$$\frac{\Gamma, x : A \vdash B : \mathcal{U}_i \qquad \Gamma \vdash P : M =_A N \qquad \Gamma \vdash Q : B[M/x]}{\Gamma \vdash \mathtt{subst}\ P\ Q : B[N/x]}\ (\mathrm{subst})$$

Dependent Functions

Definition 5.7: Dependent Function Type Assignments [75, p434]

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : (x : A) \to B}\ (\to I) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash (x : A) \to B : \mathcal{U}_{i \sqcup j}}\ (\Pi)$$

$$\frac{\Gamma \vdash M : (x : A) \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}\ (\to E)$$

A dependent function is a generalised version of the familiar function type [75]. In a dependent function, the codomain depends on the value given to the function; it is a type family $B : A \to \mathcal{U}$. We write these types $(x : A) \to B$, where x is a variable bound over B. Note that in the elimination rule, the argument supplied to M is also substituted in the type, $B[N/x]$.

A itself could be a universe, which means x would be a type, so we get polymorphism (over a given universe) for free. Types $(x : A) \to B$ where x doesn't occur free in B are just the same (and written) as our old function types, $A \to B$.

Dependent functions correspond with $\forall$ quantification in logic. One major advantage of dependent types is that the types make guarantees about compile time safety;

> **Example 5.1.1.** Consider a function $f : x \to (y : \mathbb{N}) \to (y > 0) \to \mathbb{N}$, $f := \lambda xyp.x/y$. Here, x and y are both natural numbers, and p is a proof that y is greater than 0. This means we can be sure that the function won't have a division by zero error at runtime; the function won't run without a certificate that the denominator is non-zero.

Dependent Pairs

Definition 5.8: Dependent Pair Type Assignments [75, p435]

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B[M/x]}{\Gamma \vdash (M, N) : (x : A) \times B}\ (\times I) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash (x : A) \times B : \mathcal{U}_{i \sqcup j}}\ (\Sigma)$$

$$\frac{\Gamma \vdash M : (x : A) \times B}{\Gamma \vdash \pi_1(M) : A}\ (\times E_1) \qquad \frac{\Gamma \vdash M : (x : A) \times B}{\Gamma \vdash \pi_2(M) : B[\pi_1(M)/x]}\ (\times E_2)$$

To get the correspondence with $\exists$ quantification, we generalise pairs $A \times B$ to *dependent pairs*; $(x : A) \times B$. Again, $B : A \to \mathcal{U}$ is a family of types indexed by A, and x is bound over B. Types $(x : A) \times B$ are inhabited by pairs (y, N). We call the left element the *witness*, and the right element the *proof*. In the right elimination rule, we see that the witness is substituted into the type of the proof. $\mathcal{U}_{i \sqcup j}$ means the least upper bound of the two levels $\mathcal{U}_i$ and $\mathcal{U}_j$; its use in the (Σ) rule helps keep the system consistent.

This characterisation of existence is constructive, and is in fact called 'strong' existential quantification; a proof of $(x : A) \times B$ must give a witness that we can extract. This can be compared

with a 'weak' existential, which might come from a proof of $\neg\forall$.

> **Example 5.1.2.** $\exists x.\text{even}(x)$ can be translated to type theory as: $(x : \mathbb{N}) \times \text{even}(x)$. A term inhabiting this type will be a pair (y, M), where $y : \mathbb{N}$ and M will provide a proof that y is even.

5.2 The Problem of Control

Unfortunately, dependent types don't easily expand to classical logic. As Herbelin showed in [39], a naive mix of dependent types and control leads to a 'degeneracy in the domain of discourse'. In English, this means that you can prove any two terms are equal; for example, one can give a proof that $0 = 1$. Specifically, if we add Σ types to $\lambda\mu$ (with the (ζ) rules), we are able to derive the degeneracy.

Such an offending term is given by Miquey in [51];

$$P := \mu\alpha.[\alpha](0, \mu\delta.[\alpha](1, \mathtt{refl})) : \exists x.x = 1$$

If we observe the reductions when asking for a witness x such that $x = 1$,

$$\begin{array}{rcl} \pi_1(\mu\alpha.[\alpha](0, \mu\delta.[\alpha](1, \mathtt{refl}))) & \to & \mu\alpha.[\alpha]\pi_1(0, \mu\delta.[\alpha]\pi_1(1, \mathtt{refl})) \\ & \to & \mu\alpha.[\alpha]0 \\ & \to & 0 \end{array}$$

Then using the typing rule for right projection, (noting that $((x = 1)[\pi_1(P)/x]) \equiv (0 = 1)$) we can derive $0 = 1$,

$$\frac{\Gamma \vdash P : \exists x.x = 1}{\Gamma \vdash \pi_2(P) : 0 = 1}$$

As described by Miquey [51, p8], the issue comes down to P behaving differently in different contexts. In the left projection, P gives the (incorrect) witness 0. In the right projection, the term reaches the $\mu\delta.[\alpha]$, which makes the term throw away the witness 0 (via $\mu\delta$), and then 'backtrack' to the original context (via $[\alpha]$), in which it uses the witness 1 in the proof – P uses a different witness depending on its context.

A key thing to note is that P does contain a valid proof of $\exists x.x = 1$ as a subterm, it's just the backtracking that causes the problem. This suggests that, apart from a collapse of the domain of discourse to a single element, we aren't able to prove things that aren't true for any x.

5.3 Avoiding the Degeneracy

5.3.1 dPA$^\omega$

As the degeneracy is caused by a direct interaction between control and dependent types, a simple solution is to only allow values within the proofs of dependent types and within their elimination (and a call-by-value evaluation strategy) [51].

In [38], Herbelin defines a calculus, dPA$^\omega$, in which this restriction can be relaxed to a subset of terms called *negative elimination free* (NEF). dPA$^\omega$ is a proof system for Peano Arithmetic that allows for classical proofs with dependent types about arithmetic terms with finite types[1].

We present the syntax in 5.9. The calculus and its associated types are split into two (mutually defined) layers; one for *terms*, which represent mathematical terms (like $1 + 2$); and one for *proofs*, representing proofs about the mathematical terms. For completeness, we show the syntax for the inductive, coinductive and recursive proof/term constructors, but they won't be relevant to our discussion.

[1] In this context, finite types means the simple Curry types with the only atomic type being the natural numbers.

Definition 5.9: dPA$^\omega$ Syntax [38]

t, u	::=	x	Arithmetic Variable
	|	0	Zero
	|	$s(t)$	Successor
	|	rec t of $[u_0 \mid (x,y).u_s]$	Recursion Operator
	|	$\lambda x.t$	Arithmetic Function
	|	tu	Arithmetic Function Application
	|	wit(p)	Dependent Left Projection
p, q	::=	a	Proof Variable
	|	(p, q)	Simple Pair
	|	split $(a,b) = p$ in q	Simple Pair Destructor
	|	(t, p)	Dependent Pair
	|	prf(p)	Dependent Right Projection
	|	dest $(x,a) = p$ in q	Non-Dependent Destructor for Dependent Pair
	|	$\mathtt{in}_i(p)$	Sum Type Injection
	|	$\mathtt{case}(p, a_1.q_1, a_2.q_2)$	Sum Elimination
	|	$\lambda x.p$	Term Abstraction
	|	pt	Proof-Term Application
	|	$\lambda a.p$	Dependent Abstraction
	|	pq	Dependent Application
	|	`refl`	Reflexivity
	|	`subst` $p\ q$	Substitution
	|	ind t *of* $[p \mid (x,a).q]$	Inductive Fixpoint
	|	cofix$(t, b, x)\ p$	Coinductive Fixpoint
	|	$\mathtt{catch}_\alpha\ p$	Save Context
	|	`throw` $\alpha\ p$	Switch to Context
	|	`exfalso` p	Context Abort
	|	`let` $a = p$ `in` q	

The control terms are $\mathtt{catch}_\alpha\ p$, `throw` $\alpha\ p$ and `exfalso` p, which can be read as $\mu\alpha.[\alpha]p$, $\mu\delta.[\alpha]p$ and $\mu\delta[top]p$, respectively [38]. Intuitively, $\mathtt{catch}_\alpha\ p$ will 'catch' the current context, and redirect it to all α-named terms – including running p in this context. `throw` $\alpha\ p$ will 'throw' away the current context, and run p in the context that is redirected from α. `exfalso` p is very similar to the $\mathcal{A}$ operator: computationally, it means to 'abort' from the current context, and run p in an empty context; logically, it relates to *ex falso quidliobet*, that we know as $(\bot E)$.

Although quite cumbersome, this stratified syntax makes it apparent when our pairs and functions are allowed to be dependent. As we will see in the typing, a pair of two proofs cannot be dependent, but a pair of a term and a proof can be.

The syntax for 'case' is slightly different, where $\mathtt{case}(p, a_1.q_1, a_2.q_2)$ means a_i is bound over q_i. The typing rule is similar to before, but the two cases being 'functions' is now replaced them by being proofs using the free variable a_i:

$$\frac{\Gamma \vdash p : A \vee B \qquad \Gamma, a_1 : A \vdash q_1 : C \qquad \Gamma, a_2 : B \vdash q_2 : C}{\Gamma \vdash \mathrm{case}(p, a_1.q_1, a_2.q_2) : C}\ (\vee E)$$

For the restrictions on when we allow dependent types, we define the proof *values* and NEF proofs of the calculus.

Definition 5.10: dPA$^\omega$ Proof Values [38]

The values of this dPA$^\omega$ are,

$$V ::= x \mid \mathtt{in}_i(V) \mid (V_1, V_2) \mid (t, V) \mid \lambda a.p \mid \lambda x.p \mid \mathtt{refl}$$

Definition 5.11: NEF Proofs [38]

The NEF proofs are;

$$\begin{array}{rcl} N & ::= & a \\ & | & (N_1, N_2) \mid \text{split } (a,b) = N_1 \text{ in } N_2 \mid (t, N) \mid \text{prf}(N) \mid \text{dest } (x,a) = N_1 \text{ in } N_2 \\ & | & \text{in}_i N \mid \text{case}(p, a.N_1, b.N_2) \\ & | & \lambda x.p \mid \lambda a.p \\ & | & \texttt{refl} \mid \texttt{subst } N_1\ N_2 \\ & | & \text{ind } t \textit{ of } [N_0 | (x,a).N_s] \mid \text{cofix}(t,b,x)\ N \mid \texttt{let } a = N_1 \texttt{ in } N_2 \end{array}$$

NEF proofs represent those proofs which cannot backtrack when evaluated [52, p112]; note that it is more general than values, and certainly all values are NEF. Importantly, this definition says that $\lambda x.p$ and $\lambda a.p$ for any proof p is NEF. This is why applications pq and pt aren't NEF, even for p and q NEF; if $p = \lambda x.p'$ (so p is NEF) and p' isn't NEF, then certainly $p'[t/x]$ isn't NEF (and a similar reasoning for $p = \lambda a.p'$ and pq).

Perhaps we could expand NEF to contain applications $(\lambda x.p)q$ when $p, q \in$ NEF, although this is covered by its immediate reduction to $\texttt{let } x = q \texttt{ in } p$. This suggests it could be worth exploring if a proof p is NEF when $p \to^* q$, and $q \in$ NEF, or that we could consider the set of terms that are 'reducible to NEF'.

5.3.2 Reductions

Reductions in this calculus largely follow a call-by-value strategy [38, 52], and we highlight the main rule that applications reduce to let expressions:

$$(\lambda x.p)q \to \texttt{let } x = q \texttt{ in } p$$

The only exception to the call-by-value is for the coinductive operator, `cofix`, which follows a lazy (call-by-need) reduction. Lazy reduction is needed as coinductive terms can be potentially infinite, so we only want to evaluate the terms we definitely need.

5.3.3 Type Assignments

In dPA$^\omega$, terms have so called *finite types*, which are constructed from the natural numbers and functions on the natural numbers [38],

$$T, U ::= \mathbb{N} \mid T \to U$$

The proofs have the interesting types, called *formulae*[2] [38];

$$\begin{array}{rcl} A, B & ::= & t = u \mid (a : A) \to B \mid A \vee B \mid A \wedge B \mid \bot \mid \top \\ & | & \forall (x : T).A \mid \exists (x : T).A \mid \nu(t, f, x).A \end{array}$$

It might seem odd that we have both a dependent function type $(a : A) \to B$ and a for-all type $\forall (x : T).A$; but a close inspection of the bound variable will explain why: the dependent function $(a : A) \to B$ represents a proof of B depending on a proof of A, and is based on implication; $\forall (x : T).A$ represents a proof about all arithmetic terms of type T, and corresponds instead with $\forall$ of first order logic. This means that in dependent functions (unlike dependent pairs), proofs can depend on both terms and proofs.

The distinction between $A \wedge B$ and $\exists (x : T).A$ is more immediate; it shows that, in a pair, the type of the proof can only depend on a term, not another proof. In particular, we can't existentially quantify over proofs (i.e. $\exists (a : A).B$ is not a valid type).

The typing judgements are of the form $\Gamma \vdash t : T$ or $\Gamma \vdash p : A$; the covariables α are typed as $\alpha : A^{\perp\!\!\!\perp}$, negated assumptions in the context

[2] $\nu(t, f, x).A$ represents the type for coinductive constructs, which, again, we ignore in this discussion.

Pairs

The syntax allows for two kinds of pairs; a pair of proofs (p,q), or a pair of a term and a proof (t,p). If $p : A, q : B$, then (p,q) is typed by the (non-dependent) conjunction $A \wedge B$; the typing rule for introduction is as in 3.15, the elimination rule (using the split constructor) is easy to derive, taking into account Herbelin's definition of $\pi_i := \text{split } (a_1,a_2) = p \text{ in } a_i$.

Definition 5.12: Dependent Pairs [38]

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash p : A[t/x]}{\Gamma \vdash (t,p) : \exists(x:T).A}\ (\exists I) \qquad \frac{\Gamma \vdash p : \exists(x:T).A \qquad \Gamma, x:T, a:A \vdash q : B}{\Gamma \vdash \text{dest } (x,a) = p \text{ in } q : B}\ (\exists E)$$

$$\frac{\Gamma \vdash p : \exists(x:T).A \qquad p \in \text{NEF}}{\Gamma \vdash \text{wit } p : T}\ (\exists^d E_1) \qquad \frac{\Gamma \vdash p : \exists(x:T).A \qquad p \in \text{NEF}}{\Gamma \vdash \text{prf } p : A[\text{wit } p/x]}\ (\exists^d E_2)$$

The pairs (t,p) are the dependent pairs; if $t : T, p : A$, we have $(t,p) : \exists(x:T).A$. This proof has two kinds of elimination rules, $(\exists E)$ and $(\exists^d E_i)$. $(\exists E)$ can be used only when A does not depend on x, so this can be seen as the simple product type, $(t,p) : T \times A$. Here it doesn't matter if $p \notin \text{NEF}$. If, however, x does occur free in A, then the dependent elimination $(\exists^d E_i)$ must be used and, crucially, we can only ask for the witness or proof of p when $p \in \text{NEF}$.

Functions

Definition 5.13: Dependent Functions [38]

$$\frac{\Gamma, a : A \vdash p : B}{\Gamma \vdash \lambda a.p : (a:A) \to B}\ (\to I) \qquad \frac{\Gamma \vdash p : (a:A) \to B \qquad \Gamma \vdash q : A \qquad a \notin fv(B)}{\Gamma \vdash pq : A}\ (\to E)$$

$$\frac{\Gamma \vdash p : (a:A) \to B \qquad \Gamma \vdash q : A \qquad q \in \text{NEF}, a \in fv(B)}{\Gamma \vdash pq : B[q/a]}\ (\to E^d)$$

$$\frac{\Gamma, x : T \vdash p : A}{\Gamma \vdash \lambda x.p : \forall(x:T).A}\ (\forall I) \qquad \frac{\Gamma \vdash p : \forall(x:T).A \qquad \Gamma \vdash t : T}{\Gamma \vdash pt : A[t/x]}\ (\forall E)$$

The change to dependent functions is similar. $(\to I)$ is the same as before, and we have two arrow elimination rules that correspond to whether or not the functions are dependent or not. In $(\to E)$, the function is not dependent, so we don't need $q \in \text{NEF}$. In $(\to E^d)$, the function is dependent, so we do need to ensure $q \in \text{NEF}$.

The $(\forall)$ rules are simpler as the argument of $(\forall)$-typed functions are terms, which we know can't backtrack, so can use the usual rules of ITT.

5.4 Using dPA$^\omega$

In Herbelin's original presentation of dPA$^\omega$ [38], they claim the usual desirable properties of a calculus; subject reduction, normalisation and consistency. Subject reduction is proven, but the proof of consistency relies on the proof of normalisation, which was only sketched. Indeed, this sketch turned out to be 'hard to formalize properly' [52, p113].

Normalisation is tackled by Miquey in [52, p221]. They don't directly prove normalisation of dPA$^\omega$, but instead devise a sequent calculus version of dPA$^\omega$ (building upon their work in [51]) called dLPA$^\omega$, for which they do prove normalisation. They argue that, as the two calculi share the same computational features, 'it is easy to convince ourself that dPA$^\omega$ normalises too'. Unfortunately, a proof of this statement isn't given.

The good news is, this work has recently grown into a more general (polarised) sequent calculus $\mathrm{L}_{\mathrm{dep}}$ [54]. In a presentation [55] and, more recently, a draft paper [53] they add control Extended Calculus of Constructions [45] (ECC) giving a calculus called $\mathrm{ECC}_{\mathrm{K}}$, for which a translation into $\mathrm{L}_{\mathrm{dep}}$ is given. Put simply, $\mathrm{ECC}_{\mathrm{K}}$ looks a lot like a generalised dPA^{ω} mixed with ITT, and this idea will serve as the basis of our work in Chapter 7.

5.5 Inductive Families

In this section we will give a brief overview of inductive families, following a mix of the exposition by Dyber [29] and Brady [14]. To keep consistent with the rest of this project, as well as to help with intuition, we will present inductive family schema via inference rules.

Inductive families are a generalisation of data types by adding *indices* to the types; values that the constructors can depend on. Inductive families can be seen as types with type parameters and value indices.

The go-to example for such types is vectors [59, p24]; lists whose length is in their type.

$$\begin{aligned}&\textbf{data}\ \mathsf{Vec}\ (A : \mathcal{U}) : \mathbb{N} \longrightarrow \mathcal{U}_i\ \textbf{where}\\ &\quad \mathtt{nil} : \mathsf{Vec}\ A\ 0\\ &\quad \mathsf{cons} : (n : \mathbb{N}) \to A \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (n+1)\end{aligned}$$

Importantly, a function that asks for the head of a vector can have the type signature;

$$\mathsf{headV} : (A : \mathcal{U}) \to (n : \mathbb{N}) \to \mathsf{Vec}\ A\ n \to (n > 0) \to A$$

so that the function takes a vector and a *proof* that its length is non-zero, before it is able to return the head of the underlying list.

Definitions

We make a few definitions that we'll need for the inductive families.

Definition 5.14: Telescope [15, 29]

We write the *telescope* $(a :: \sigma)$ as an abbreviation of the (finite) sequence of type assignments $(a_1 : \sigma_1)\cdots(a_n : \sigma_n)$. σ is called a *sequence of types*, of the form $\sigma_1 \cdots \sigma_n$. Crucially, each successive assignment binds its variable over the subsequent sub-telescope. That is, a_i is bound over $(a_{i+1} : \sigma_{i+1})\cdots(a_n : \sigma_n)$.
We sometimes write $\vec{\sigma}$ to mean a sequence of types, and $\vec{x}$ for a sequence of terms, $x_1 \cdots x_n$.

Definition 5.15: Telescope Derivation [15, 79]

Telescopes are typed by the following rules. We write $\cdot$ for an empty telescope and term vector. $m\ \vec{m}$ is taken to mean the sequence formed by adding m to the start of the sequence $\vec{m}$; $(a : A)(b :: B)$ is the telescope formed by adding $(a : A)$ to the start of the telescope $(b :: B)$.

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash m : A \qquad \Gamma \vdash \vec{n} : (b :: B[m/x])}{\Gamma \vdash m\ \vec{n} : (x : A)(xs :: B)}$$

If we write $\vec{m} = m\ \vec{n}$, and $(c :: C) = (x : A)(xs :: B)$, then we can succinctly write the conclusion of the rule above as $\Gamma \vdash \vec{m} : (a :: A)$.

$$\frac{\Gamma \vdash A_1 : \mathcal{U}_i \qquad \cdots \qquad \Gamma, a_1 : A_1, \ldots, a_{n-1} : A_{n-1} \vdash A_n : \mathcal{U}_i}{\Gamma \vdash (a :: A) : \mathcal{U}_i}$$

Note that this means A_i can depend on a_j for $j < i$.

5.5.1 Defining Inductive Families

To generalise Haskell-style data types we need to know; how to form the data type, how to use the constructors, and how to eliminate a data type by case analysis. Brady [14, p23] describes an

inductive family as a disjoint union of constructors. In this sense, they can be seen as a direct generalisation of Haskell's parameterised data types, but allowing them to be indexed by values.

Definition 5.16: Inductive Family Declaration [14]

$$\textbf{data}\ D\ (\sigma :: P) : (\alpha :: I) \longrightarrow \mathcal{U}_i\ \textbf{where}$$
$$\sum_{i=1}^{n} c_i : (\vec{A_i})(\vec{B_i}) \longrightarrow D\ \sigma\ \vec{s_i}$$

Each σ are s-types, so we have $\sigma_i : \mathcal{U}$. $(\vec{A_i})(\vec{B_i}) \longrightarrow D\ \sigma\ \vec{s_i}$ is a shorthand for $A_{i1} \to \cdots \to A_{ik_i} \to B_{i1} \to \cdots \to B_{il_i} \to D\ \sigma\ \vec{s_i}$. These types are allowed to be dependent, and they can also depend on the parameters σ and the indices α. The types A_{ij} do not contain recursive occurences of $D\ \sigma$; the types B_{ij} are allowed to contain recursive occurences of $D\ \sigma$, however these occurences must be *strictly positive*. This means B_{ij} can be written $B_{ij} = (\vec{B'_{ij}}) \to D\ \sigma\ \vec{w_i}$, for some indices $\vec{w_i}$. Strict positivity is used to ensure terminating data type constructors [59, p100].

In general, this means the type of each constructor must be strictly positive wrt $D\ \sigma$. The arguments of type A_{ij} are thus called the *non-recusrive* arguments, and those of type B_{ij} are called *recursive*. The $\vec{s_i}$ are allowed to be fully dependent on all the types, parameters and indices before. This marks one of the main differences between the parameters and the indices; the parameters must be the same when constructing the type from other occurences of $D\ \sigma$, but the indices are allowed to change.

Each constructor c_i has implicit arguments for the parameters σ, i.e.

$$c_i : (\vec{P})(\vec{A_i})(\vec{B_i}) \longrightarrow D\ \sigma\ \vec{s_i}$$

But the parameters must be the same as the data type it is constructing, so, when defining the constructors, the $(\vec{P})$ can be inferred from the P in the first part of the declaration. This means that we can type the application of constructors by:

$$c_i\ \vec{p}\ \vec{x_i}\ \vec{y_i} : D\ \vec{p}\ \vec{s_i}$$

Valid Definition

To ensure an instantiation of a data type is valid, we need to check the parameters and indices are of the correct types, and that the parameters are s-types.

Definition 5.17: Inductive Family Definition Check [29, 14]

$$\textbf{data}\quad \frac{\Gamma \vdash (\sigma :: P) :: \mathcal{U}_i \qquad \Gamma, (\sigma :: P) \vdash (\alpha :: I) :: \vec{\mathcal{U}_i}}{\Gamma \vdash \Big(D\ (\sigma :: P) : (\alpha :: I)\Big) : \mathcal{U}_i}$$

$$\textbf{where}\quad \frac{\Gamma \vdash (a :: A_i) :: \vec{\mathcal{U}_j} \qquad \Gamma, (a :: A_i) \vdash (b :: B_i) :: \vec{\mathcal{U}_j} \qquad \Gamma, (a :: A_i), (b :: B_i) \vdash \vec{s_i} :: I}{\Gamma \vdash c_i : (a :: A_i)(b :: B_i) \longrightarrow D\ \sigma\ \vec{s_i}}$$

Formation and Introduction

Definition 5.18: Inductive Family Formation [29, 14]

Given a data declaration $D\ (\sigma :: P) : (\alpha :: I)$ in the environment, we check instances of the type by:

$$\frac{\Gamma \vdash \vec{p} :: P \qquad \Gamma \vdash \vec{q} :: I[\vec{p}/\sigma]}{\Gamma \vdash D\ \vec{p}\ \vec{q} : \mathcal{U}_i}\ (\text{data})$$

Definition 5.19: Inductive Family Introduction [29, 14]

We are given a data declaration $D\ (\sigma :: P) : (\alpha :: I)$, and constructors $c_i : (a :: A_i)(b :: B_i) \longrightarrow D\ \sigma\ \vec{s_i}$. There are two ways that we could allow the introduction of constructors. The first is to introduce them as constant functions, turning the telescopes into dependent function types;

$$c_i : (p_1 : P) \to \cdots (a_1 : A_{i1}) \to (a_2 : A_{i2}) \to \cdots \to D\ \vec{p}\ \vec{s_i}$$

This has the advantage of allowing partial applications of the constructors; they essentially become built-in functions. This style is typed by:

$$\frac{}{\Gamma \vdash c_i : (p_1 : P) \to \cdots (a_1 : A_{i1}) \to (a_2 : A_{i2}) \to \cdots \to D\ \vec{p}\ \vec{s_i}}\ (c_i)$$

The other option is to only allow the constructors to be used when they have all their arguments:

$$\frac{\Gamma \vdash (\vec{m} :: P) \qquad \Gamma \vdash \vec{n} :: A_i[\vec{m}/p] \qquad \Gamma \vdash \vec{t} :: B_i[\vec{m}/p, \vec{n}/a]}{\Gamma \vdash c_i \vec{m} \vec{n} \vec{t} : (\vec{p} :: P)(a :: A_i)(b :: B_i) \longrightarrow D\ \vec{p}\ \vec{s_i}}\ (c_i)$$

Elimination

An eliminator for an inductive family can be seen as a generalisation of the 'case' construct for sum types, or the 'case * of' construct of Haskell.

Definition 5.20: Inductive Family Eliminator [29, 14, 79]

$$\begin{array}{rl} \text{(target):} & \Gamma \vdash t : D\ p\ q \\ \text{(motive):} & \Gamma, (x : D\ p\ q), (y :: P), (z :: I) \vdash C : \mathcal{U}_i \\ \text{(methods):} & \Gamma \vdash m_i\ p : (a :: A_i)(b :: B_i)(v :: V_i) \longrightarrow C[(c_i\ p\ a\ b)/x, p/y, \vec{s_i}/z] \end{array}$$
$$\frac{}{\Gamma \vdash \texttt{elim}\ t\ \texttt{by}\ (m_1 | \cdots | m_n) : C[t/x, p/y, q/z]}$$

Where;

- $B_{ij} \equiv (\vec{B'_{ij}}) \to D\ p\ q' \equiv B_{ij1} \to \cdots \to B_{ijk} \to D\ p\ q'$, and,
- Each V_{ij} is of the form $(w :: \vec{B'_{ij}}) \to C[(b_j\ w)/x, p/y, q'/z]$

In this sense, each V_{ij} corresponds to b_j. Seeing this as an induction, we get that V_{ij} represents the generalised induction hypothesis of b_j [29, p9].

In m_i; v_i corresponds to the inductive hypothesis constructed from the arguments supplied to b_i. A comparison of the types of each shows their similarity;

$$b_j : \Big((w_1 : B_{ij1}) \to \cdots \to (w_k : B_{ijk}) \to D\ p\ q'\Big)$$
$$v_j : \Big((w_1 : B_{ij1}) \to \cdots \to (w_k : B_{ijk}) \to C[(b_j\ w)/x, p/y, q'/z]\Big)$$

Again, using the notion that this is a generalised sum type, we get the expected reduction; that the constructor c_i is associated with the elimination method m_i. Generalising further, we also allow for recursive elimination. Writing $\vec{m}$ for $(m_1 | \cdots | m_n)$, we get the reduction [14][3];

$$\texttt{elim}\ (c_i\ \vec{p}\ \vec{a}\ \vec{b})\ \texttt{by}\ \vec{m} \longrightarrow m_i\ \vec{a}\ \vec{b}\ \Big(\vec{w}.(\texttt{elim}\ b_1\ \vec{w}\ \texttt{by}\ \vec{m})\Big) \cdots \Big(\vec{w}.(\texttt{elim}\ b_n\ \vec{w}\ \texttt{by}\ \vec{m})\Big)$$

Note that we have abstractions of $\vec{w}$ in $\vec{w}.(\texttt{elim}\ b_j\ \vec{w}\ \texttt{by}\ \vec{m})$; m_i is able to supply these with arguments to cause a recursive elimination.

[3] The reduction presented in [14] is only for the case that each $b_i : D\ p\ q'$, rather than a function. What we give is a simple generalisation.

6 | Simple Types with Name Polymorphism in $\lambda\mu$

In this chapter we present a type system for the $\lambda\mu$-calculus with sum and product types, and a principal typing algorithm. This is used as the core of a Haskell-style language with context control with name polymorphism.

6.1 Typing Algorithms for $\lambda\mu$-calculus

In this section we give a principal pairing algorithm for the $\lambda\mu$-calculus, and give the intuition behind how it was found.

6.1.1 Towards a Principal Pairing Algorithm

(μ) rule

We recall the μ rule:

$$\frac{\Gamma \vdash M : \bot \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.M : A \mid \Delta}\ (\mu)$$

The behaviour of μ with conclusions is analagous to that of λ with the context.
The algorithm will take in $\mu\alpha.M$. We will walk through how the algorithm was derived.

1. To form the principal pair for this term, we will need to know the principal pair of the subterm M, so we should recurse on it.

2. Assuming it succeeds on M, the typing rule states that we expect M to have type $\bot$, so we will need to check this.

3. We will have to manage two cases for the returned conclusions:

   (a) If α is not free in M, it won't appear in the open conclusions. From a bottom up perspective reading of the rules, this is still permitted in the typing; as the (Ax) rule allows open open conclusions. So we want this to be a valid term, but, as their is no type A given, we will have to generate a fresh type for the term.

   (b) If α is in the conclusions, we should simply remove it, and set the type to that of α.

Collecting these points, we get the function:

$$pp\ \mu\alpha.M \quad = \quad \begin{cases} \langle \Gamma, \Delta \setminus (\alpha : A), A \rangle & \text{if } (\alpha : A) \in \Delta \quad \text{\# by (3b)} \\ \langle \Gamma, \Delta, \varphi \rangle & \text{otherwise} \quad \text{\# by (3a)} \end{cases}$$

where

$\langle \Gamma, \Delta, P \rangle$	$= pp\ M$	# by (1)
if $P \neq \bot$	**error**	# by (2)
φ	is fresh	# by (3a)

(*name*) rule

Following the intuition of [70], we compare the rules for application and naming. This will help us figure out how to form the principal pair of a named term.

$$\frac{\Gamma;\neg\Delta \vdash M : A \to B \qquad \Gamma;\neg\Delta \vdash N : A}{\Gamma;\neg\Delta \vdash MN : B}\,(\to E) \qquad \frac{\Gamma;\neg\Delta \vdash N : A}{\Gamma;\neg\Delta,\neg(\alpha : A) \vdash [\alpha]N : \bot}\,(name)$$

Note that (*name*) focuses on a term N. We can see the similarity in the 'shape' of the syntax of the terms $[\alpha]N$ and MN. The typing of MN comes naturally from the idea of M being a function with type $A \to B$, and N an argument with that required type A. This application produces a value with type B.

We can try to, informally, see how this corresponds to $[\alpha]N$. Notice that, in the conclusions, we write $\neg(\alpha : A)$. We can use the logical interpretation to see that the rule says; if we assume a proof of $\neg A$, and we 'apply' it to a proof of A, we will get falsum. If we now write $\neg A$ as $A \to \bot$, so we can interpret this 'application' like so (forgiving the abuse of notation):

$$\frac{\alpha : A \to \bot \vdash [\alpha] : A \to \bot \qquad \Gamma;\neg\Delta \vdash N : A}{\Gamma;\neg\Delta,\alpha : A \to \bot \vdash [\alpha]N : \bot}$$

which looks very much like $(\to E)$[1].

A case to watch out for is if α is already in the co-context. Well, then the type A must be the same for N and α. This suggests an algorithm would need to use a unification procedure to ensure the types are consitent between N and α.

We collect our observations:

1. There will only be one context and co-context to reason about (coming from $pp\ M$), so we won't need to worry about unifying multiple (co-)contexts.

2. The returned type will always be $\bot$.

3. We will have to extend the co-context by $(\alpha : A)$, where A is the type of N.

4. If α is already in the co-context, we will have to unify its type with that of N.

Thus we can derive the algorithmic case for $[\alpha]N$ terms:

$$\begin{aligned} pp\ [\alpha]N \quad = \quad & \left\{ \begin{array}{ll} \langle \Gamma, (\Delta, \alpha : A), \bot \rangle & \text{if } \alpha \notin \Delta \\ S\langle \Gamma, \Delta, \bot \rangle & \text{if } \alpha \in \Delta \end{array} \right\} \text{\# by (2) \& (3)} \\ & \text{where} \\ & \quad \langle \Gamma, \Delta, A \rangle \qquad\qquad = pp\ N \\ & \quad \text{if } (\alpha : B) \in \Delta \\ & \qquad S = unify\ A\ B \qquad\qquad \text{\#by (4)} \\ & \quad \text{\# removed the extra unifications, by (1)} \end{aligned}$$

6.1.2 Principal Pairing Definitions

Definition 6.1: Principal Pairing Algorithm for $\lambda\mu$

We give pp, the principal pairing algorithm for $\lambda\mu^a$. We assume there is an environment allowing for unique fresh types to be generated, and to be able to handle errors. The cases for x, $\lambda x.M$ and MN are almost identical to the usual principal pairing algorithm for the

[1] In fact, in [70, p94-98], this idea is explored in more detail, by allowing any term to appear in the square brackets (along with some other big changes to the calculus). Application for $[M]N$ is interpreted as applying the *continuation* M to the argument N, and not returning any value (i.e. the return type is $\bot$, in line with our interpretation).

λ-calculus [8]; we just add the co-context in the returned tuple.

$$
\begin{array}{lcl}
pp\ x & = & \langle \{x : \varphi\}, \varnothing, \varphi \rangle \\
pp\ \lambda x.M & = & \begin{cases} \langle \Gamma \setminus (x : A), \Delta, A \to B \rangle & \text{if } (x : A) \in \Gamma \\ \langle \Gamma, \Delta, \varphi \to B \rangle & \text{otherwise} \end{cases} \\
\multicolumn{3}{l}{\text{where}} \\
\multicolumn{3}{l}{\quad \langle \Gamma, \Delta, B \rangle = pp\ M} \\
\multicolumn{3}{l}{\quad \varphi \quad \text{is fresh}} \\
\\
pp\ MN & = & S_3 \circ S_2 \circ S_1 \langle \Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2, \varphi \rangle \\
\multicolumn{3}{l}{\text{where}} \\
\multicolumn{3}{l}{\quad \langle \Gamma_1, \Delta_1, A_1 \rangle = pp\ M} \\
\multicolumn{3}{l}{\quad \langle \Gamma_2, \Delta_2, A_2 \rangle = pp\ N} \\
\multicolumn{3}{l}{\quad \varphi \quad \text{is fresh}} \\
\multicolumn{3}{l}{\quad S_1 = unify\ A_1\ (A_2 \to \varphi)} \\
\multicolumn{3}{l}{\quad S_2 = unifyCtxt\ (S_1\ \Gamma_1)\ (S_1\ \Gamma_2)} \\
\multicolumn{3}{l}{\quad S_3 = unifyConcs\ (S_2 \circ S_1\ \Delta_1)\ (S_2 \circ S_1\ \Delta_2)} \\
\\
pp\ \mu\alpha.M & = & \begin{cases} \langle \Gamma, \Delta \setminus (\alpha : A), A \rangle & \text{if } (\alpha : A) \in \Delta \\ \langle \Gamma, \Delta, \varphi \rangle & \text{otherwise} \end{cases} \\
\multicolumn{3}{l}{\text{where}} \\
\multicolumn{3}{l}{\quad \langle \Gamma, \Delta, P \rangle = pp\ M} \\
\multicolumn{3}{l}{\quad \text{if } P \neq \bot \quad \textbf{error}} \\
\multicolumn{3}{l}{\quad \varphi \quad \text{is fresh}} \\
\\
pp\ [\alpha]M & = & \begin{cases} \langle \Gamma, (\Delta, \alpha : A), \bot \rangle & \text{if } \alpha \notin \Delta \\ S\langle \Gamma, \Delta, \bot \rangle & \text{if } \alpha \in \Delta \end{cases} \\
\multicolumn{3}{l}{\text{where}} \\
\multicolumn{3}{l}{\quad \langle \Gamma, \Delta, A \rangle = pp\ M} \\
\multicolumn{3}{l}{\quad \text{if } (\alpha : B) \in \Delta} \\
\multicolumn{3}{l}{\qquad S = unify\ A\ B}
\end{array}
$$

[a] Although this algorithm in fact returns a triple instead of a pair, we use the pairing name for consistency with the literature.

This algorithm satisfies soundness and completeness, meaning it is safe to use as the basis for a propositional proof assisstant.

Proposition 6.2: Completeness for $pp_{\lambda\mu} \Longrightarrow$ Proof in A.1

If we have $\Gamma \vdash M : B \mid \Delta$ then,

- The algorithm succeeds on M.
- If $pp\ M = \langle \Pi, \Sigma, A \rangle$, then there exists a substitution S such that:

$$S\,\Sigma \subseteq \Gamma, S\,\Pi \subseteq \Delta \text{ and } S\,A = B$$

Proposition 6.3: Soundness for $pp_{\lambda\mu} \Longrightarrow$ Proof in A.1

For any term M, if $pp\ M = \langle \Gamma, \Delta, A \rangle$, then $\Gamma \vdash M : A \mid \Delta$

6.2 A Theorem Prover for Classical Propositional Logic

We expand upon the last section to define a core calculus of a theorem prover for classical propositional logic. We derive its type system by combining that of the $\lambda\mu$ and the Λ^N [8] calculi, to obtain a system with context control and name polymorphism; $\lambda\mu^N$ ($\lambda\mu$-calculus with names). We lift the restriction that definition terms can't contain names, so that they are now able to call functions that have already been defined. The type system doesn't allow recursive definitions. As the typing is sequential (with respect to the order the functions are defined), the calculus doesn't permit mutual recursion.

Although mutual recursion can be a desirable feature of a programming language, disallowing it makes it easier to ensure correctness with regards to type checking as proof verification. Consider if we had a haskell-like program:

```
proof_of_false :: False
proof_of_false = another_proof_of_false

another_proof_of_false :: False
another_proof_of_false = proof_of_false
```

If we allowed these mutual definitions, we would have to check for termination in both of them (else we would have 2 proofs of false). In this example, it's obvious that either function wouldn't terminate, but it is a well known result that generalised termination checking is undecidable [74].

6.2.1 Syntax

We add products and sums to $\lambda\mu$, along with their reductions (including the (ζ) reductions), and then add the ability for terms to reference function names.

Definition 6.4: $\lambda\mu^N$-calculus

The set of types is defined by;

$$A, B ::= \bot \mid \top \mid \varphi \mid A \to B \mid A + B \mid A \times B$$

The set of terms and definitions is defined by the grammar;

$$\begin{array}{rcl} M, N & ::= & x \mid \mathrm{n} \mid \lambda x.M \mid MN \\ & \mid & \mu\alpha.[\beta]M \mid \mu\alpha.[top]M \\ & \mid & \pi_i(M) \mid (M, N) \\ & \mid & \mathrm{in}_i(M) \mid case(M, N_1, N_2) \\ & \mid & \langle\rangle \\ \mathrm{n} & ::= & \textit{`string of characters'} \\ \mathrm{Defs} & ::= & (\mathrm{n} = M); \mathrm{Defs} \mid \epsilon \\ \mathrm{Program} & ::= & \langle \mathrm{Defs}; M \rangle \end{array}$$

We extend reductions with the rule that, if $\mathrm{n} = M$ in the definitions, then $\mathrm{n} \to M$.

6.2.2 Type Assignments for $\lambda\mu^N$

Definition 6.5: Type Assignments for the $\lambda\mu$-calculus with names

We define an *environment* $\mathcal{E}$, as a (partial) mapping of function names to types; $\mathrm{n} : A$.
We use the judgements,

$$\mathcal{E}; \Gamma \vdash M : A \mid \Delta \qquad \mathcal{E} \vdash \mathrm{Defs} \qquad \mathcal{E}; \Gamma \vdash \langle \mathrm{Defs}, M \rangle : A \mid \Delta$$

for terms, definition lists and programs (respectively), with type assignments;

Logical Rules

$$\frac{}{\mathcal{E};\Gamma, x : A \vdash x : A \mid \Delta}\ (Ax) \qquad \frac{}{\mathcal{E}, \mathrm{n} : A; \Gamma \vdash \mathrm{n} : S\,A \mid \Delta}\ (\mathrm{n}) \qquad \frac{}{\mathcal{E};\Gamma \vdash \langle\rangle : \top \mid \Delta}\ (\top)$$

$$\frac{\mathcal{E};\Gamma, x : A \vdash M : B \mid \Delta}{\mathcal{E};\Gamma \vdash \lambda x.M : A \to B \mid \Delta}\ (\to I) \qquad \frac{\mathcal{E};\Gamma \vdash M : A \to B \mid \Delta \qquad \mathcal{E};\Gamma \vdash N : A \mid \Delta}{\mathcal{E};\Gamma \vdash MN : B \mid \Delta}\ (\to E)$$

$$\frac{\mathcal{E};\Gamma \vdash M : A \qquad \mathcal{E};\Gamma \vdash N : B}{\mathcal{E};\Gamma \vdash \langle M, N\rangle : A \times B \mid \Delta}\ (\times I) \qquad \frac{\mathcal{E};\Gamma \vdash M : A \times B \mid \Delta}{\mathcal{E};\Gamma \vdash \pi_1(M) : A \mid \Delta}\ (\times E_1) \qquad \frac{\mathcal{E};\Gamma \vdash M : A \times B \mid \Delta}{\mathcal{E};\Gamma \vdash \pi_2(M) : B \mid \Delta}\ (\times E_2)$$

$$\frac{\mathcal{E};\Gamma \vdash M : A + B \qquad \mathcal{E};\Gamma, \vdash N : A \to C \mid \Delta \qquad \mathcal{E};\Gamma, \vdash L : B \to C \mid \Delta}{\Gamma \vdash \mathrm{case}(M, N, L) : C \mid \Delta}\ (+E)$$

$$\frac{\mathcal{E};\Gamma \vdash M : A \mid \Delta}{\mathcal{E};\Gamma \vdash \mathrm{in}_1(M) : A + B \mid \Delta}\ (+I_1) \qquad \frac{\mathcal{E};\Gamma \vdash M : A \mid \Delta}{\mathcal{E};\Gamma \vdash \mathrm{in}_2(M) : B + A \mid \Delta}\ (+I_2)$$

Structural Rules

$$\frac{\mathcal{E};\Gamma \vdash M : \bot \mid \alpha : A, \Delta}{\mathcal{E};\Gamma \vdash \mu\alpha.M : A \mid \Delta}\ (\mu) \qquad \frac{\mathcal{E};\Gamma \vdash M : A \mid \Delta}{\mathcal{E};\Gamma \vdash [\alpha]M : \bot \mid \alpha : A, \Delta}\ (name) \qquad \frac{\mathcal{E};\Gamma \vdash M : \bot \mid \Delta}{\mathcal{E};\Gamma \vdash [top]M : \bot \mid \Delta}\ (top)$$

Enviroment Rules

$$\frac{\mathcal{E};\varnothing \vdash M : A \mid \varnothing \qquad \mathcal{E}, \mathrm{n} : A \vdash \mathrm{Defs}}{\mathcal{E}, \mathrm{n} : A \vdash (\mathrm{n} = M); \mathrm{Defs}}\ (\mathrm{Defs}) \qquad \frac{}{\mathcal{E} \vdash \epsilon}\ (\epsilon)$$

$$\frac{\mathcal{E} \vdash \mathrm{Defs} \qquad \mathcal{E}, \Gamma \vdash M : A \mid \Delta}{\mathcal{E};\Gamma \vdash \langle \mathrm{Defs}; M\rangle : A \mid \Delta}\ (\mathrm{Program})$$

Note in the (n) rule, we have a substitution S of the principal type of n; this is to allow this name polymorphism.

We give the principal pairing algorithm in Definition 6.8. The entrypoint for the algorithm is to supply $pp_{\lambda\mu^N}$ with a list of function definitions (in the order they were defined in) and the current term that needs to be typed. As is usual, we assume there is some environment allowing for unique, fresh types to be generated when needed, and that it can deal with errors from the algorithm or unification.

We extend unification for sums and products by unifying them in the same way we unify arrow types (replacing $\square$ for $+$ and $\times$);

$$\begin{aligned} \mathit{unify}\ (A \square B)\ (C \square D) = \quad & S_2 \circ S_1 \\ & \text{where} \\ & \quad S_1 = \mathit{unify}\ A\ B \\ & \quad S_2 = \mathit{unify}\ (S_1 C)\ (S_1 D) \end{aligned}$$

We also allow unification of variables with constant types (e.g. $\bot$). As the definition of unification is identical for each of the binary type connectives, we can see that *unify* will satisfy the same properties as before, in particular; Proposition 3.10.

Again, we have soundness and completeness of the algorithm:

Proposition 6.6: Completeness for $pp_{\lambda\mu^N} \implies$ Proof in A.1

If we have $\mathcal{E}; \Gamma \vdash \langle \text{Defs}, M \rangle : B \mid \Delta$ then,

- The algorithm succeeds on M and,
- If $pp_N\ \mathcal{E}\ \langle \text{Defs}, M \rangle = \langle \Pi, \Sigma, A \rangle$, then there exists a substitution S such that:

$$S\,\Sigma \subseteq \Gamma, S\,\Pi \subseteq \Delta, \text{ and } S\,A = B$$

Proposition 6.7: Soundness for $pp_{\lambda\mu^N} \implies$ Proof in A.1

For any term M, if $pp_N\ \mathcal{E}\ \langle \text{Defs}, M \rangle = \langle \Gamma, \Delta, A \rangle$, then $\mathcal{E}; \Gamma \vdash \langle \text{Defs}, M \rangle : A \mid \Delta$

This means the algorithm will also be logically sound, so is certainly safe to use for a theorem prover. As for completeness, $\lambda\mu$ is complete only in what can be proved in natural deduction, it is not complete with respect to the proofs themselves, although this is because $\lambda\mu$ is based on free deduction, rather than the usual natural deduction. As a result, we have an algorithm that is effectively able to check a proof of propositional logic, and in turn we have a calculus that is able to prove any tautology in classical logic.

Definition 6.8: Principal Pairing for $\lambda\mu^N$

$$
\begin{array}{lcl}
pp\ \mathcal{E}\ x & = & \langle \{x:\varphi\}, \varnothing, \varphi \rangle \\
pp\ \mathcal{E}\ \mathrm{n} & = & \langle \varnothing, \varnothing, \mathit{FreshInstance}(\mathcal{E}\,\mathrm{n}) \rangle \\
pp\ \mathcal{E}\ \lambda x.M & = & \begin{cases} \langle \Gamma \setminus (x:A), \Delta, A \to B \rangle & \text{if } (x:A) \in \Gamma \\ \langle \Gamma, \Delta, \varphi \to B \rangle & \text{otherwise} \end{cases} \\
 & & \quad \text{where} \quad \langle \Gamma, \Delta, B \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad \varphi \quad \text{is fresh} \\
pp\ \mathcal{E}\ MN & = & S_3 \circ S_2 \circ S_1 \langle \Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2, \varphi \rangle \\
 & & \quad \text{where} \quad \langle \Gamma_1, \Delta_1, A_1 \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad \langle \Gamma_2, \Delta_2, A_2 \rangle = pp\ \mathcal{E}\ N \\
 & & \qquad\qquad \varphi \quad \text{is fresh} \\
 & & \qquad\qquad S_1 = \mathit{unify}\ A_1\ (A_2 \to \varphi) \\
 & & \qquad\qquad S_2 = \mathit{unifyCtxt}\ (S_1\ \Gamma_1)\ (S_1\ \Gamma_2) \\
 & & \qquad\qquad S_3 = \mathit{unifyConcs}\ (S_2 \circ S_1\ \Delta_1)\ (S_2 \circ S_1\ \Delta_2) \\
pp\ \mathcal{E}\ \mu\alpha.M & = & \begin{cases} \langle \Gamma, \Delta \setminus (\alpha:A), A \rangle & \text{if } (\alpha:A) \in \Delta \\ \langle \Gamma, \Delta, \varphi \rangle & \text{otherwise} \end{cases} \\
 & & \quad \text{where} \quad \langle \Gamma, \Delta, P \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad \text{if } P \neq \bot \quad \textbf{error} \\
 & & \qquad\qquad \varphi \quad \text{is fresh} \\
pp\ \mathcal{E}\ [\alpha]M & = & \begin{cases} \langle \Gamma, (\Delta, \alpha:A), \bot \rangle & \text{if } \alpha \notin \Delta \\ S\langle \Gamma, \Delta, \bot \rangle & \text{if } \alpha \in \Delta \end{cases} \\
 & & \quad \text{where} \quad \langle \Gamma, \Delta, A \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad \text{if } (\alpha:B) \in \Delta \\
 & & \qquad\qquad\quad S = \mathit{unify}\ A\ B \\
pp\ \mathcal{E}\ [\mathit{top}]M & = & S\langle \Gamma, \Delta, \bot \rangle \\
 & & \quad \text{where} \quad \langle \Gamma, \Delta, A \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad S = \mathit{unify}\ A\ \bot \\
pp\ \mathcal{E}\ \mathrm{in}_i(M) & = & \begin{cases} \langle \Gamma, \Delta, A + \varphi \rangle & \text{if } i = 1 \\ \langle \Gamma, \Delta, \varphi + A \rangle & \text{if } i = 2 \end{cases} \\
 & & \quad \text{where} \quad \langle \Gamma, \Delta, A \rangle = pp\ M \\
 & & \qquad\qquad \varphi \quad \text{is fresh} \\
pp\ \mathcal{E}\ \mathrm{case}(M,N,L) & = & S_5 \circ S_4 \circ S \langle \Gamma_1 \cup \Gamma_2 \cup \Gamma_3, \Delta_1 \cup \Delta_2 \cup \Delta_3, \varphi \rangle \\
 & & \quad \text{where} \quad \langle \Gamma_1, \Delta_1, A \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad \langle \Gamma_2, \Delta_2, B \rangle = pp\ \mathcal{E}\ N \\
 & & \qquad\qquad \langle \Gamma_3, \Delta_3, C \rangle = pp\ \mathcal{E}\ L \\
 & & \qquad\qquad S_1 = \mathit{unify}\ A\ (\varphi_1 + \varphi_2) \\
 & & \qquad\qquad S_2 = \mathit{unify}\ (S_1\ B)\ (S_1(\varphi_1 \to \varphi)) \\
 & & \qquad\qquad S_3 = \mathit{unify}\ (S_2 \circ S_1\ C)\ (S_2 \circ S_1(\varphi_2 \to \varphi)) \\
 & & \qquad\qquad S = S_3 \circ S_2 \circ S_1 \\
 & & \qquad\qquad S_4 = \mathit{unifyCtxts}\ (S\,\Gamma_1)\ (S\,\Gamma_2)\ (S\,\Gamma_3) \\
 & & \qquad\qquad S_5 = \mathit{unifyCtxts}\ (S_4 \circ S\,\Delta_1)\ (S_4 \circ S\,\Delta_2)\ (S_4 \circ S\,\Delta_3) \\
 & & \qquad\qquad \varphi, \varphi_i \quad \text{are fresh} \\
pp\ \mathcal{E}\ \pi_i(M) & = & \langle \Gamma, \Delta, S\,\varphi_i \rangle \\
 & & \quad \text{where} \quad \langle \Gamma, \Delta, A \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad S = \mathit{unify}\ A\ (\varphi_1 \times \varphi_2) \\
 & & \qquad\qquad \varphi_i \quad \text{is fresh} \\
pp\ \mathcal{E}\ (M,N) & = & S_2 \circ S_1 \langle \Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2, A \times B \rangle \\
 & & \quad \text{where} \quad \langle \Gamma_1, \Delta_1, A \rangle = pp\ \mathcal{E}\ M \\
 & & \qquad\qquad \langle \Gamma_2, \Delta_2, B \rangle = pp\ \mathcal{E}\ N \\
 & & \qquad\qquad S_1 = \mathit{unifyCtxts}\ \Gamma_1\ \Gamma_2 \\
 & & \qquad\qquad S_2 = \mathit{unifyConcs}\ (S_1\ \Delta_1)\ (S_1\ \Delta_2)
\end{array}
$$

$$
\begin{array}{lcl}
pp\ \mathcal{E}\ \langle \epsilon; M \rangle & = & pp\ \mathcal{E}\ M \\
pp\ \mathcal{E}\ \langle (\mathrm{n} = M); \mathrm{Defs};\ N \rangle & = & pp\ (\mathcal{E}, \mathrm{n}:A)\ \langle \mathrm{Defs};\ N \rangle \\
 & & \quad \text{where } \langle \varnothing, \varnothing, A \rangle = pp\ \mathcal{E}\ M
\end{array}
$$

7 | ECC$_{\lambda\mu}$: Dependently Typed $\lambda\mu$-calculus

In this chapter we present a type system that adds control to the Extended Calculus of Constructions [45], using work based on [38, 55]. It is based on the calculus **ECC$_K$** from the draft paper [53], but we use μ and $[\cdot]$ as our control operators, and we extend the calculus to allow for coproducts, inductive records and inductive families. We bring the notion NEF along with these extensions, so that we are able to safely have dependent eliminations for both structures. We also present a bidirectional typing algorithm to be used as the core for a theorem prover based on such a calculus.

7.1 Some formalisms

Following [53], we collapse the syntax of dPA$^\omega$, and collapse arithmetical terms and proofs into a single set of terms. We make every (arithmetical) term NEF. We prove a couple of technical lemmas (that the author didn't seem to be able to find) about this collapsed syntax (that should translate easily to dPA$^\omega$):

Lemma 7.1: $\Rightarrow$ Proofs in B.1

- (NEF-Substitution Closure) If $m, n \in$ NEF then $m[n/x] \in$ NEF
- (NEF-Reduction Closure) If $m \in$ NEF, and $m \to n$, then $n \in$ NEF

As a corollary, we get the same result for $m \to^* n$. Note that we do not have this result for $m =_\beta n$; consider an example, $p, q \in$ NEF, $x \notin fv(p)$; then $(\lambda x.p)q \notin$ NEF, reduces to $\texttt{let}\ a = p\ \texttt{in}\ q \in$ NEF. Thus they are β-equal, but not both in NEF. This fact will be very useful when looking at a user-level language.

We claim that this fact of NEF-reduction closure is essential to the idea of NEF terms; that they can't contain a subterm that will backtrack, and that they won't reduce to a term that will backtrack. When expanding the calculus, we will use this as a goal for when to define new syntax to be NEF; specifically, when a new term construct has a reduction into previously defined terms, we will make the new construct NEF only when all of its single-step reductions are NEF.

This same idea will also help us determine when we can allow for dependent elimination. If we can reduce new syntax to terms for which we already know we when we allow dependent elimination, we can infer the requirements back to the new syntax.

7.2 Type System

The calculus and type assignments are based on a combination of dPA$^\omega$[38] and ECC [45], and is heavily inspired by **ECC$_K$**[53]. The syntax of dPA$^\omega$ is collapsed into a single set of terms, while maintaining the idea of when a term is NEF. This does require knowing when a *type* is NEF; we follow the idea of [53] where types are NEF if all their subterms are NEF. The author admits that this does need further investigation. The syntax is obtained by generalising [38], guided by [53]

7.2.1 Syntax

Definition 7.2: Syntax

m, n, A, B	::=	x	Variable
	|	$(x : A) \to B$	Dependent Function Type
	|	$\lambda x : A.m$	Lambda Abstraction
	|	$\texttt{let}\ x = m\ \texttt{in}\ n$	Let
	|	mn	Function Application
	|	$(x : A) \times B$	Dependent Pair Type
	|	(m, n)	Dependent Pair
	|	$\pi_i(m)$	Pair Projection ($i = 1, 2$)
	|	$A + B$	Coproduct Type
	|	$\texttt{in}_i(m)$	Injection ($i = 1, 2$)
	|	$\texttt{case}\ m \triangleright z.A\ \texttt{of}\ (x.n_1 \mid y.n_2)$	(Dependent) Coproduct Eliminator
	|	$m = n$	Identity Type
	|	$\texttt{refl}$	Reflexivity
	|	$\texttt{subst}\ m\ n$	Substitution
	|	$\mu\alpha.[\beta]m$	Context Control
	|	$\mu\alpha.[top]n$	Context Control
	|	$\bot$	Empty Type
	|	$\mathbf{1}$	Unit Type
	|	$\langle\rangle$	Element of Unit Type
	|	$\mathcal{U}_i$	Type Universe ($i = 1, 2, \dots$)

We usually use $A, B, \dots$ to denote types specifically, and $t, u, \dots, m, n, \dots$ for general terms (that may or may not be types). x, y, z are reserved for variables. We will often write about terms of the form $\mu\alpha.[\beta]m$, but, unless otherwise stated, the discussions will also apply to terms of the form $\mu\alpha.[top]m$

The normal forms of the calulus are easy to derive by inspection of the reduction rules (seen in Section 7.2.2), and avoiding redexes:

Definition 7.3: $\mathbf{ECC}_{\lambda\mu}$ Normal Forms

n, N	::=	$x \mid \bot \mid \mathbf{1} \mid \langle\rangle \mid \mathcal{U}_i$	
	|	$(x : N_1) \to N_2 \mid (x : N_1) \times N_2 \mid N_1 + N_2 \mid n_1 = n_2$	
	|	$\lambda x : A.n \mid (n_1, n_2) \mid \texttt{in}_i(n) \mid \texttt{refl}$	
	|	$\mu\alpha[\beta]n$	$(n \neq \mu\delta.n')$
	|	$\texttt{case}\ n \triangleright z.N\ \texttt{of}\ (x_1.n_1 \mid x_2.n_2)$	$(n \neq \texttt{in}_i(n'), \mu\alpha.n')$
	|	$\pi_i(n)$	$(n \neq (n_1, n_2), \mu\alpha.n')$
	|	$x n_1 \cdots n_k$	$(n_i \neq \mu\alpha.n')$
	|	$\texttt{subst}\ n_1\ n_2$	$(n_1 \neq \texttt{refl})$

The values of the calculus are standard, and can be seen as a generalisation of those in [38]. We use the intuition given in [68], that constructors are values only when their arguments are values, and that data types are always values. As we can see dependent function types and dependent pair types as data types, we can view them as always being values. Importantly, eliminators are not values, which is why we don't have applications, projections, subst or case analysis in the values. The definition we present is obtained by generalising [38], using [68] and guided by [53].

Definition 7.4: $\mathbf{ECC}_{\lambda\mu}$ Values

v, V	::=	$x \mid \bot \mid \mathbf{1} \mid \langle\rangle \mid \mathcal{U}_i$
	|	$(x : A) \to B \mid (x : A) \times B \mid A + B \mid m = n$
	|	$\lambda x : A.m \mid (v_1, v_2) \mid \texttt{in}_i(v) \mid \texttt{refl}$

With a definition of values, we have the call-by-value evaluation contexts, which we obtain by generalising those of [38], guided by those of [68, 6].

Definition 7.5: $\mathbf{ECC}_{\lambda\mu}$ CBV Evaluation Contexts

$$\begin{array}{rcl} \kappa & ::= & \bullet \mid \kappa m \mid v\kappa \mid \mu\alpha.[\beta]\kappa \\ & \mid & \mathtt{in}_i(\kappa) \mid (\kappa, m) \mid (V, \kappa) \\ & \mid & \mathtt{case}\ \kappa \triangleright z.A\ \mathtt{of}\ (x_1.n_1 | x_2.n_2) \mid \pi_i(\kappa) \mid \mathtt{subst}\ \kappa\ m \\ & \mid & \mathtt{let}\ x = \kappa\ \mathtt{in}\ m \end{array}$$

We obtain the definitions of NEF terms by generalising those of [38], guided by the work in [53]:

Definition 7.6: $\mathbf{ECC}_{\lambda\mu}$ Negative Elimination Free Terms

$$\begin{array}{rcl} m_{\text{NEF}}, n_{\text{NEF}}, A_{\text{NEF}}, B_{\text{NEF}} & ::= & x \mid \bot \mid \mathbf{1} \mid \langle\rangle \mid \mathcal{U}_i \\ & \mid & (x : A_{\text{NEF}}) \to B_{\text{NEF}} \mid \lambda x : A.m \mid \mathtt{let}\ x = m_{\text{NEF}}\ \mathtt{in}\ n_{\text{NEF}} \\ & \mid & (x : A_{\text{NEF}}) \times B_{\text{NEF}} \mid (m_{\text{NEF}}, n_{\text{NEF}}) \mid \pi_i(m_{\text{NEF}}) \\ & \mid & A_{\text{NEF}} + B_{\text{NEF}} \mid \mathtt{in}_i(m_{\text{NEF}}) \mid \mathtt{case}\ m_{\text{NEF}} \triangleright z.A\ \mathtt{of}\ (x.n_{1\text{NEF}} | y.n_{2\text{NEF}}) \\ & \mid & m_{\text{NEF}} = n_{\text{NEF}} \mid \mathtt{refl} \mid \mathtt{subst}\ m_{\text{NEF}}\ n_{\text{NEF}} \end{array}$$

Note that, as in dPA^ω, m need not be NEF for $\lambda x : A.m$ to be NEF.

7.2.2 Reductions

As the calculus is built as an extension to dPA^ω, we retain its call-by-value reduction strategy [38]. We also allow the left and right μ-reduction, as the evaluation is deterministic under the call-by-value strategy. The full set of reductions can be found in B.2. The reductions to highlight are:

$$\begin{array}{rcl} (\lambda x.m)n & \to_\beta & \mathtt{let}\ x = n\ \mathtt{in}\ m, (m \neq \mu\alpha.m') \\ \mathtt{let}\ x = v\ \mathtt{in}\ m & \to & m[v/x] \\ v(\mu\alpha.m) & \to_{\mu'} & \mu\alpha.m[[\alpha]v \bullet / [\alpha]\bullet] \\ (\mu\alpha.m)n & \to & \mu\alpha.m[[\alpha] \bullet n / [\alpha]\bullet] \end{array}$$

Crucially, we prioritise (left) μ' reduction over β-reduction; this is to preserve the CBV nature of the calculus. Without this prioritisation, we lose the CBV nature of reductions; if $v = \lambda x.n$, then we would have a *critical pair*, $(\lambda x.n)(\mu\alpha.m)$. This is similar to the issues seen in the $\lambda\mu\tilde{\mu}$-calculus [20], in which they argue that CBV is gained by prioritising the right reduction[1]. The key idea is that the argument $\mu\alpha.m$ is evaluated before being given to $\lambda x.n$. Using our CBV evaluation contexts, we can succinctly write the μ reductions as

$$\kappa\{\mu\alpha.m\} \to \mu\alpha.m[[\alpha]\kappa\{n\}/[\alpha]n]$$

expressing how the μ and $[\cdot]$ operators control the context.

7.2.3 Type Assignments

The type assignments are similar to those in [53], and are essentially a combination ECC and a collapsed dPA^ω. The main difference with ECC is the addition of control and that dependent eliminations can only be used for NEF terms. The difference with $\mathbf{ECC}_\mathbf{K}$[53] is the addition of coproducts. Note that NEF terms are allowed to have types that are not NEF, and we still allow such terms to be used in dependent elimination.

[1] In their calculus, this is similar to prioritising the μ reduction over the $\tilde{\mu}$; $\tilde{\mu}$ reduction is similar to the left reduction of our calculus.

Definition 7.7: Dependent Calc [38, 53]

Valid Contexts

$$\frac{}{\varnothing \vdash \cdot \mid \varnothing}\ (\cdot) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta}{\Gamma, x : A \vdash x : A \mid \Delta}\ (Ax) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta}{\Gamma \vdash \cdot \mid \alpha : A\ \Delta}\ (\alpha x)$$

Function Introduction/Formation

$$\frac{\Gamma, x : A \vdash m : B \mid \Delta}{\Gamma \vdash \lambda x.m : (x : A) \to B \mid \Delta}\ (\to I) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta \qquad \Gamma, x : A \vdash B : \mathcal{U}_j \mid \Delta}{\Gamma \vdash (x : A) \to B : \mathcal{U}_{i \sqcup j} \mid \Delta}\ (\Pi)$$

Pair Introduction/Formation

$$\frac{\Gamma \vdash m : A \mid \Delta \qquad \Gamma \vdash n : B[m/x] \mid \Delta}{\Gamma \vdash (m, n) : (x : A) \times B \mid \Delta}\ (\times I) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta \qquad \Gamma, x : A \vdash B : \mathcal{U}_j \mid \Delta}{\Gamma \vdash (x : A) \times B : \mathcal{U}_{i \sqcup j} \mid \Delta}\ (\Sigma)$$

Coproduct Introduction/Formation

$$\frac{\Gamma \vdash m : A_i \mid \Delta \qquad \Gamma \vdash A_1 : \mathcal{U}_j \mid \Delta \qquad \Gamma \vdash A_2 : \mathcal{U}_j \mid \Delta}{\Gamma \vdash \mathrm{in}_i(m) : A_1 + A_2 \mid \Delta}\ (+I_i) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash B : \mathcal{U}_i \mid \Delta}{\Gamma \vdash A + B : \mathcal{U}_i \mid \Delta}\ (+F)$$

Non-Dependent Elimination

$$\frac{\Gamma \vdash m : A \to B \mid \Delta \qquad \Gamma \vdash n : A \mid \Delta}{\Gamma \vdash mn : B \mid \Delta}\ (\to E) \qquad \frac{\Gamma \vdash m : A \mid \Delta \qquad \Gamma, x : A \vdash n : B \mid \Delta \qquad x \notin fv(B)}{\Gamma \vdash \mathtt{let}\ x = m\ \mathtt{in}\ n : B \mid \Delta}\ (\mathrm{let})$$

$$\frac{\Gamma \vdash m : (x : A) \times B \mid \Delta \qquad x \notin fv(B)}{\Gamma \vdash \pi_1(m) : A \mid \Delta}\ (\times E_1) \qquad \frac{\Gamma \vdash m : (x : A) \times B \mid \Delta \qquad x \notin fv(B)}{\Gamma \vdash \pi_2(m) : B \mid \Delta}\ (\times E_2)$$

$$\frac{\begin{array}{c} z \notin fv(C) \\ \Gamma \vdash m : A + B \mid \Delta \qquad \Gamma \vdash C : \mathcal{U}_i \mid \Delta \qquad \Gamma, x : A \vdash n_1 : C \mid \Delta \qquad \Gamma, y : B \vdash n_2 : C \mid \Delta \end{array}}{\Gamma \vdash \mathrm{case}\ m \triangleright z.C\ \mathrm{of}\ (x.n_1 | y.n_2) : C \mid \Delta}\ (+E)$$

Dependent Elimination

$$\frac{\Gamma \vdash m : (x : A) \to B \mid \Delta \qquad \Gamma \vdash_{\mathrm{NEF}} n : A \mid \Delta}{\Gamma \vdash mn : B[n/x] \mid \Delta}\ (\to E^d) \qquad \frac{\Gamma \vdash_{\mathrm{NEF}} m : A \mid \Delta \qquad \Gamma, x : A \vdash n : B \mid \Delta}{\Gamma \vdash \mathtt{let}\ x = m\ \mathtt{in}\ n : B[m/x] \mid \Delta}\ (\mathrm{let}^d)$$

$$\frac{\Gamma \vdash_{\mathrm{NEF}} m : (x : A) \times B \mid \Delta}{\Gamma \vdash \pi_1(m) : A \mid \Delta}\ (\times E_1^d) \qquad \frac{\Gamma \vdash_{\mathrm{NEF}} m : (x : A) \times B \mid \Delta}{\Gamma \vdash \pi_2(m) : B[\pi_1(m)/x] \mid \Delta}\ (\times E_2^d)$$

$$\frac{\begin{array}{c} \Gamma, z : A + B \vdash C : \mathcal{U}_i \mid \Delta \\ \Gamma \vdash_{\mathrm{NEF}} m : A + B \mid \Delta \qquad \Gamma, x : A \vdash n_1 : C[\mathrm{in}_1(x)/z] \mid \Delta \qquad \Gamma, y : B \vdash n_2 : C[\mathrm{in}_2(y)/z] \mid \Delta \end{array}}{\Gamma \vdash \mathrm{case}\ m \triangleright z.C\ \mathrm{of}\ (x.n_1 | y.n_2) : C[m/z] \mid \Delta}\ (+E^d)$$

NEF

$$\frac{\Gamma \vdash m : A \mid \Delta \qquad m \in \mathrm{NEF}}{\Gamma \vdash_{\mathrm{NEF}} m : A \mid \Delta}\ (\mathrm{NEF}I) \qquad \frac{\Gamma \vdash_{\mathrm{NEF}} m : A \mid \Delta}{\Gamma \vdash m : A \mid \Delta}\ (\mathrm{NEF}E)$$

Control

$$\frac{\Gamma \vdash m : \mathbf{0} \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.m : A \mid \Delta}\ (\mu) \qquad \frac{\Gamma \vdash m : A \mid \Delta}{\Gamma \vdash [\alpha]m : \mathbf{0} \mid \alpha : A, \Delta}\ (name) \qquad \frac{\Gamma \vdash m : \mathbf{0} \mid \Delta}{\Gamma \vdash [top]m : \mathbf{0} \mid \Delta}\ (top)$$

Types

$$\frac{}{\Gamma \vdash \mathbf{1} : \mathcal{U}_i \mid \Delta}\ (\mathbf{1}) \qquad \frac{}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i \mid \Delta}\ (\mathbf{0}) \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathbf{1} \mid \Delta}\ (\text{unit}) \qquad \frac{}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1} \mid \Delta}\ (\mathcal{U}_i)$$

Propositions

$$\frac{}{\Gamma \vdash \mathbb{P} : \mathcal{U}_0 \mid \Delta}\ (\mathbb{P}) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta \qquad \Gamma, x : A \vdash B : \mathbb{P} \mid \Delta}{\Gamma \vdash (x : A) \to B : \mathbb{P} \mid \Delta}\ (\Pi_{\mathbb{P}})$$

Equality

$$\frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash m : A \mid \Delta}{\Gamma \vdash \mathtt{refl} : m =_A m \mid \Delta}\ (\mathtt{refl}) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash a : A \mid \Delta \qquad \Gamma \vdash b : A \mid \Delta}{\Gamma \vdash a =_A b : \mathcal{U}_i \mid \Delta}\ (=)$$

$$\frac{\Gamma, x : A \vdash B : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash n : B[P/x] \mid \Delta \qquad \Gamma \vdash m : P = Q \mid \Delta}{\Gamma \vdash \mathtt{subst}\ m\ n : B[Q/x] \mid \Delta}\ (\text{subst})$$

Propositions

The new type, $\mathbb{P}$ represents the impredicative set of logical propositions [45]. An impredicative definition is one that is self-referencing; in our case, it means that the set of propositions $\mathbb{P}$ is defined in terms of itself [45]. What separates this set from the usual types is the rule $(\Pi_{\mathbb{P}})$, which characterises the type $\mathbb{P}$, where functions whose codomain is a proposition are themselves propositions.

Context Control

dPA$^\omega$ and $\mathbf{ECC_K}$ use catch and throw control operators (dPA$^\omega$ also has exfalso), rather than μ and $[\cdot]$. We can recover $\mu\alpha.[\beta]m$ by encoding them as $\mathtt{catch}_\alpha\ (\mathtt{throw}\ \beta\ m)$, effectively reversing the intuition Herbelin gives in [38, p4] explaining $\mathtt{catch}_\alpha\ m$ is roughly equivalent to $\mu\alpha.[\alpha]m$, and $\mathtt{throw}\ \alpha\ m$ to $\mu\delta.[\alpha]m$ (with δ not free in m)[2]. Thus our encoding seems sound, as

$$\mathtt{catch}_\alpha\ (\mathtt{throw}\ \beta\ m) \mapsto \mu\alpha.[\alpha](\mu\delta.[\beta]m) \to \mu\alpha.[\beta]m$$

We are also able to recover $\mathtt{exfalso}\ m$ by $\mu\delta.[top]m$ (with $\delta \notin fv(m)$).

Dependent Coproducts

In Herbelin's original formulation of dPA$^\omega$, case terms were defined via pattern matched methods. Although this doesn't let us consider terms of the form $\mathtt{case}\ m\ \mathtt{of}\ (p|q)$, it allows us to determine when such a term is NEF.

If we did consider terms $\mathtt{case}\ m\ \mathtt{of}\ (p|q)$, the naïve condition for a case construct to be NEF would be: $\mathtt{case}\ m_{\text{NEF}}\ \mathtt{of}\ (p_{\text{NEF}}|q_{\text{NEF}}) \in$ NEF. If, however, $m = \mathtt{in}_1(n)$, the term would reduce to pn. If p is of the form $\lambda x.p'$, which is NEF, then $pn \to p'[n/x]$, which we don't know to be NEF. It follows that we must know when the subterm p' is NEF, in order to determine when the original is NEF. If we force the syntax to always have the form $\mathtt{case}\ m\ \mathtt{of}\ (x.p|y.q)$, then we are always able to determine whether or not the term is NEF.

7.2.4 Properties

We maintain the closure of NEF terms under substitution and reductions from dPA$^\omega$.

[2] This does require we allow the symmetric μ reduction, or we consider dPA$^\omega$ with the control operators only allowed to capture the right context

Lemma 7.8: NEF Substitution/Reduction Closure $\Longrightarrow$ Proof in B.1

(i) $m, n \in \text{NEF} \implies m[n/x] \in \text{NEF}$

(ii) $m \in \text{NEF}$ and $m \to^* n \implies n \in \text{NEF}$

A term substitution lemma is usually used to prove subject reduction. However, a naive substitution lemma is not provable; consider $m = \pi_1(x, \mathtt{refl}) : x =_A x$, for some $x : A$. If $n : A$ and $n \notin \text{NEF}$, then $m[n/x] = \pi_1(n, \mathtt{refl})$ is *not* typeable. This problem is avoided by the fact that our reductions are CBV, so we know we will only be substituting values; and all values are NEF, thus the substitutions in the dependent eliminations will still be safe.

Lemma 7.9: Term Substitution $\Longrightarrow$ Proof in B.1

If there is a type C such that $\Gamma, x : C \vdash m : A \mid \Delta$ and $\Gamma \vdash_{\text{NEF}} n : C \mid \Delta$, then

$$\Gamma[n/x] \vdash m[n/x] : A[n/x] \mid \Delta[n/x]$$

Proposition 7.10: Subject Reduction $\Longrightarrow$ Proof in B.1

If $\Gamma \vdash m : A \mid \Delta$ and $m \to n$, then $\Gamma \vdash n : A \mid \Delta$.

We claim consistency of the calculus by sketching a proof of translations to and from $\mathbf{ECC}_{\mathbf{K}}$.

Claim 7.11: $\mathbf{ECC}_{\lambda\mu}$ Consistency

There is no term t such that $\vdash t : \bot$

Proof. (Sketch) We encode $\mathbf{ECC}_{\lambda\mu}$ into $\mathbf{ECC}_{\mathbf{K}}$ by:

$$\begin{array}{rcl}
\llbracket A + B \rrbracket & = & (b : \mathbb{B}) \times (\mathtt{if}\ b\ \mathtt{then}\ A\ \mathtt{else}\ B) \\
\llbracket \mathtt{in}_1(m) \rrbracket & = & (\mathtt{true}, m) \\
\llbracket \mathtt{in}_2(m) \rrbracket & = & (\mathtt{false}, m) \\
\llbracket \mu\alpha.[\beta]m \rrbracket & = & \mathtt{catch}_\alpha\ \mathtt{throw}\ \beta\ m \\
\llbracket m \rrbracket & = & \llbracket\cdot\rrbracket \text{ applied recursively to the subterms of } m
\end{array}$$

With the reverse translation given by:

$$\begin{array}{rcll}
\llbracket \mathbb{B} \rrbracket & = & \mathbf{1} + \mathbf{1} & \\
\llbracket \mathtt{true} \rrbracket & = & \mathtt{in}_1(\langle\rangle) & \\
\llbracket \mathtt{false} \rrbracket & = & \mathtt{in}_2(\langle\rangle) & \\
\llbracket \mathtt{if}\ b\ \mathtt{then}\ m\ \mathtt{else}\ n \rrbracket & = & \mathtt{case}\ \llbracket b \rrbracket \triangleright z.\llbracket \mathbb{B} \rrbracket\ \mathtt{of}\ (\llbracket m \rrbracket | \llbracket n \rrbracket) & \\
\llbracket \mathtt{catch}_\alpha\ m \rrbracket & = & \mu\alpha.[\alpha]m & \\
\llbracket \mathtt{throw}\ \alpha\ m \rrbracket & = & \mu\delta.[\alpha]m & \delta \notin fn(m) \\
\llbracket \mathtt{exfalso}\ m \rrbracket & = & \mu\delta.[top]m & \delta \notin fn(m) \\
\llbracket m \rrbracket & = & \llbracket\cdot\rrbracket \text{ applied recursively to the subterms of } m &
\end{array}$$

We then use the fact that $\mathbf{ECC}_{\mathbf{K}}$ is consistent [53]. □

Perhaps important to note is that well-typed normal forms are values.

Lemma 7.12: Proof in $\Longrightarrow$ B.1

If $n \in \text{NF}$ and there is a type A such that $\vdash n : A$, then n is a value.

7.3 Dependent Algebraic Data Types

In this section, we generalise the results for coproducts and dependent pairs to inductive families and inductive records.

As the author found towards the end of this project, work has been done by Lepigre in [43] combining data and records with $\lambda\mu$. Their system allows for simple, non-parameterised data and record types. Where we restrict to NEF terms in type assignments, they restrict to values. Lepigre explains that, from the user's perspective, the value restriction isn't an issue, as one can derive similar rules for non-values by reducing them to normal values before checking their types. Of course this then requires an additional termination check to be made on the term. As it turns out, most of the restrictions to values can simply be replaced by the more flexible restriction to NEF terms; this can be seen by just comparing the two type systems.

7.3.1 Inductive Families

The intuition behind extending the calculus to inductive families is in how they are a generalisation of coproduct types.

We have inductive family definitions:

$$\textbf{data}\ D\ (\sigma :: P) : (\alpha :: I) \longrightarrow \mathcal{U}_i\ \textbf{where}$$
$$\sum_{i=1}^{n} c_i : (\vec{A_i})(\vec{B_i}) \longrightarrow D\ \sigma\ \vec{s_i}$$

With standard syntax given by:

$$\begin{array}{rcl} m, n, A, B & ::= & \ldots \mid D_\sigma(\alpha) \mid c_i \\ & \mid & \mathtt{elim}\ m \triangleright z.C\ \mathtt{by}\ (x_1.n_1 \mid \cdots \mid x_k.n_k) \mid \mathtt{case}\ m \triangleright z.C\ \mathtt{of}\ (x_1.n_1 \mid \cdots \mid x_k.n_k) \end{array}$$

Where $x_i.n_i$ means x_i is a sequence of variables, bound over n_i. The `elim` construct is for inductive eliminations, that will recurse on the data type. case is for non-recursive elimination, just like in Haskell. Logically, case relates to not using the inductive hypotheses given by the constructors.

To allow compatibility with the classical operators, we follow the ideas we set out in 7.1; namely that the set of NEF terms should be closed under reduction, and to use their reductions to infer when we can allow dependent elimination.

First, we need to know when the new terms are NEF. For an instance $D_p(q)$ to be NEF, we should have no subterm able to backtrack, so we need that p and q are sequences of NEF terms; this is similar to when other types are NEF.

To allow constructors to form NEF terms, we allow their applications to NEF terms to be NEF. There are two ways we can argue why this is the case. The first is that our constructors are a generalisation of the successor function, and we use the definition in 7.1 to say that all the arithmetical terms of dPA$^\omega$ are NEF. Similarly, the constructors are a generalisation of $\mathtt{in}_i(\cdot)$ in dPA$^\omega$, and we have $\mathtt{in}_i(m) \in$ NEF only when $m \in$ NEF. The other perspective is to view them through their reductions. The arguments of the constructor are only accessible by the inductive eliminator (`elim`). We consider the simpler version of the rule (that doesn't recurse on the inductive hypotheses), `case`;

$$\mathtt{case}\ (c_i\ \vec{p}\ \vec{a}\ \vec{b})\ \mathtt{of}\ (x_1.m_1 \mid \cdots \mid x_k.m_k) \longrightarrow \mathtt{let}\ x_i = \vec{b}\ \vec{a}\ \vec{p}\ \mathtt{in}\ m_i$$

Where $\mathtt{let}\ \vec{x} = \vec{m}\ \mathtt{in}\ n$ is taken to mean $\mathtt{let}\ x_1 = m_1\ \mathtt{in} \cdots \mathtt{let}\ x_k = m_k\ \mathtt{in}\ n$. We know the right hand side of this equation is NEF when $\vec{a}, \vec{b}, \vec{p}$ and $m_i \in$ NEF, so we carry this requirement over to the left hand side. Thus, we get $\mathtt{case}\ (c_i\ \vec{p}\ \vec{a}\ \vec{b})\ \mathtt{of}\ (x_1.m_1 \mid \cdots \mid x_k.m_k) \in$ NEF when $\vec{a}, \vec{b}, \vec{p}, m_i \in$ NEF. This implies (by generalising coproducts), that we can consider $(c_i\ \vec{p}\ \vec{a}\ \vec{b})$ to be NEF.

Therefore, taking $\vec{m}_{\text{NEF}}$ to mean each $m_i \in$ NEF, we extend NEF terms;

$$\begin{array}{rcll} m_{\text{NEF}}, n_{\text{NEF}}, A_{\text{NEF}}, B_{\text{NEF}} & ::= & \ldots \mid c_i\ n_{\text{NEF}1} \ldots n_{\text{NEF}p} & (p = 0, 1, \ldots) \\ & \mid & \mathtt{elim}\ m_{\text{NEF}}\ \mathtt{by}\ (x_1.n_{\text{NEF}1} \mid \cdots \mid x_k.n_{\text{NEF}k}) & \\ & \mid & D_{\vec{m}_{\text{NEF}}}(\vec{n}_{\text{NEF}}) & \end{array}$$

We use the rules for validating an inductive definition from 5.17, and the rules for formation and introduction from 5.19.

We inspect the reduction rule for when to allow dependent elimination;

$$\begin{aligned}&\texttt{elim}\ (c_i\ \vec{p}\ \vec{a}\ \vec{b})\ \texttt{by}\ (x_1.m_1|\cdots|x_k.m_k)\\&\longrightarrow \texttt{let}\ x_i = \vec{p}\ \vec{a}\ \vec{b}\ \left(\vec{w}.(\texttt{elim}\ b_1\ \vec{w}\ \texttt{by}\ \vec{m})\right)\cdots\left(\vec{w}.(\texttt{elim}\ b_n\ \vec{w}\ \texttt{by}\ \vec{m})\right)\ \texttt{in}\ m_i\end{aligned}$$

We see that `elim` reduces to a let expression, so we use the rule for when we allow dependent let binding. In particular, we must have $\vec{a},\vec{b} \in \text{NEF}$, and also each of $\vec{w}.(\texttt{elim}\ b_j\ \vec{w}\ \texttt{by}\ \vec{m}) \in \text{NEF}$. This suggests that a general target t of elimination must be NEF.

As the recursive terms are under a binder we might be able to immediately determine them NEF. However, we must take care in the case that $\vec{w}$ is empty; as we then directly have the operand $(\texttt{elim}\ b_j\ \texttt{by}\ \vec{m})$, so we must have it be NEF. In this case, by the definition of when `elim` is NEF, we must have all the methods $m_i \in \text{NEF}$.

So we have multiple cases for when we allow dependent elimination: if the data type has no recursive occurrences in its constructors, we only need the arguments to the constructor, a and b, and the target term t to be NEF; if the data type has a constructor with a recursive argument, then we need $a, b, t\text{NEF}$ and also each $m_i\text{NEF}$; if the constructors only contain arguments that compute to a recursive argument, then we get the same as if it has no recursive arguments.

By enforcing pattern matching of the methods, by making them of the form $x_i.m_i$, it is possible to determine when the methods are NEF, and we are then able to determine the dependent elimination typing rule:

Definition 7.13: Inductive Family Pattern Matched Dependent Elimination

$$\frac{\begin{array}{l}\Gamma \vdash_{\text{NEF}} t : D\ p\ q \mid \Delta\\ \Gamma,(x : D\ p\ q),(y :: P),(z :: I) \vdash C : \mathcal{U}_i \mid \Delta\\ \Gamma,\vec{x_i} :: (p :: P)(a :: A_i)(b :: B_i)(v :: V_i) \vdash_{\text{NEF}} m_i : C((c_i\ p\ a\ b),p,\vec{s_i}) \mid \Delta\end{array}}{\Gamma \vdash \texttt{elim}\ t\ \texttt{by}\ (\vec{x_1}.m_1|\cdots|\vec{x_n}.m_n) : C(t,p,q) \mid \Delta}\ (\text{elim}^d)$$

Where we write $C(t,p,q)$ to mean $C[t/x,p/y,q/z]$.

In the simpler `case` construct, the reductions is of the form:

$$\texttt{case}\ (c_i\ \vec{p}\ \vec{a}\ \vec{b})\ \texttt{of}\ ((x_1.m_1|\cdots|x_k.m_k)) \longrightarrow \texttt{let}\ x_i = \vec{p}\ \vec{a}\ \vec{b}\ \texttt{in}\ m_i,$$

which suggests we only need the target t to be NEF;

Definition 7.14: Inductive Family Dependent Case Analysis

$$\frac{\begin{array}{l}\Gamma \vdash_{\text{NEF}} t : D\ p\ q \mid \Delta\\ \Gamma,(x : D\ p\ q),(y :: P),(z :: I) \vdash C : \mathcal{U}_i \mid \Delta\\ \Gamma,\vec{x_i} : (p :: P)(a :: A_i)(b :: B_i) \vdash m_i : C((c_i\ p\ a\ b),p,\vec{s_i}) \mid \Delta\end{array}}{\Gamma \vdash \texttt{case}\ t\ \texttt{of}\ (\vec{x_1}.m_1|\cdots|\vec{x_n}.m_n) : C(t,p,q) \mid \Delta}\ (\text{case}^d)$$

Where we write $C(t,p,q)$ to mean $C[t/x,p/y,q/z]$.

In the non-dependent elimination, we can use the usual typing;

Definition 7.15: Inductive Family Non-Dependent Elimination [14]

$$\frac{\Gamma \vdash t : D\ p\ q \mid \Delta \qquad \Gamma \vdash C : \mathcal{U}_i \mid \Delta \qquad \Gamma,\vec{x_i} :: (p :: P)(a :: A_i)(b :: B_i)(v :: V_i) \vdash m_i : C \mid \Delta}{\Gamma \vdash \texttt{elim}\ t\ \texttt{by}\ (\vec{x_1}.m_1|\cdots|\vec{x_n}.m_n) : C \mid \Delta}\ (\text{elim})$$

7.3.2 Inductive Records

Inductive records allow for projections to depend on one another. There is less legwork for us to derive these, as they can be encoded via dependent pairs [59]. A simple way to help ensure termination is for projections to only be able to depend on those defined previously.

The key idea behind record types is they are 'dual' to data types. Where data types are defined by their constructors, we define record types by their *destructors* or *projections*. We can define simple pairs by the record definition:

$$\begin{aligned}&\textbf{record}\ (\times)\ (A : \mathcal{U})(B : A \to \mathcal{U}) : \mathcal{U}\ \textbf{where}\\ &\quad \pi_1 : A\\ &\quad \pi_2 : B\ \pi_1\end{aligned}$$

Note that, as every projection is applied to a term of type $A \times B$, they each have an implicit first argument of type $A \times B$; so π_1 could actually be seen to have the type $A \times B \to A$. We then define the constructor for pairs (m, n) by how they are projected; $\pi_1(m, n) := m, \pi_2(m, n) := n$.

When a record allows recursion, we can generalise them to codata. A canonical example for codata is a stream; a list with infinite length.

```
codata Stream (A : Set) : Set where
    head : A
    tail : Stream A
```

We can use a stream to create an infinite list of integers, mimicking those in Haskell;

```
[_..] : Nat -> Stream Nat
head [n..] = n
tail [n..] = [suc n..]
```

Note that we define the stream by its projections, as opposed to how data is defined by its constructors. We can understand this by the idea that we define data by how it is constructed, and codata by how it behaves under its projections.

Thus we generalise products to records. A recursive record R with parameters $(\sigma :: P)$ is defined by;

$$\textbf{record}\ R\ (\sigma :: P) : (\alpha :: I) \longrightarrow \mathcal{U}_i\ \ \textbf{where}\ \ \bigtimes_{i=1}^{k} p_i : A_i$$

where A_i is strictly positive wrt R, and $fv(A_i) \subseteq \sigma$. We let $x_1, \ldots x_{i-1}$ be the free variables in A_i. We extend the syntax similarly to how we did for data types. Of course, a projection might need extra arguments; for example, we might have a record $R\ (A : \mathcal{U})$ with a projection $p_R : R \to \mathbb{N} \to A$. Thus we should to allow the syntax to bind these variables over the methods; $\texttt{build}(\vec{x}_1.m_1|\cdots|\vec{x}_k.m_k)$.;

$$m, n, A, B ::= \ldots \mid R\ \vec{p}\ \vec{q} \mid p_i(\vec{m}) \mid \texttt{build}(\vec{x}_1.n_1|\ldots|\vec{x}_k.n_k)$$

We define the NEF terms by generalising those of product types. A pair (m, n) is NEF when $m, n \in$ NEF; this can be generalised to $\texttt{build}(\vec{x}_1.m_1|\cdots|\vec{x}_k.m_k) \in$ NEF for $m_i \in$ NEF.

$$\begin{aligned}m_{\text{NEF}}, n_{\text{NEF}}, A_{\text{NEF}}, B_{\text{NEF}} \quad ::= \quad & \ldots \mid p_i(m_{\text{NEF}})\\ \mid \quad & \texttt{build}(\vec{x}_1.n_{\text{NEF}1}|\ldots|\vec{x}_n.n_{\text{NEF}k})\end{aligned}$$

For a record instance $R\ \vec{p}$ to be NEF, we impose the same restriction as for data types; that each $A_i \in$ NEF and $\vec{p}_i \in$ NEF.

Definition 7.16: Record Declaration

$$\textbf{record}\qquad \frac{\Gamma \vdash (\sigma :: P) :: \mathcal{U}_i \mid \Delta \qquad \Gamma, (\sigma :: P) \vdash (\alpha :: I) :: \mathcal{U}_i \mid \Delta}{\Gamma \vdash R\ (\sigma :: P)\ (\alpha :: I) : \mathcal{U}_i \mid \Delta}$$

$$\textbf{where}\qquad \frac{\Gamma, p_1 : R\ \sigma\ \alpha \to A_1, \ldots, p_{i-1} : R\ \sigma\ \alpha \to A_{i-1} \vdash A_i : \mathcal{U}_j \mid \Delta}{\Gamma \vdash p_i : R\ \sigma\ \alpha \longrightarrow A_i \mid \Delta}$$

The formation of record instances is very similar to that for data instances. The `build` construct is typed as a generalisation of the pairing construct $(\cdot,\cdot)$; where the type of each successive term is dependent on the previous terms.

Definition 7.17: Record Formation/Introduction

Given a valid record declaration $R\ (\sigma :: P) : (\alpha :: I) \to \mathcal{U}_i$ in the environment, we check instances of the type and introductions, where p and q are vectors of terms, by;

$$\frac{\Gamma \vdash p :: P \mid \Delta \qquad \Gamma \vdash q :: I[p/\sigma] \mid \Delta}{\Gamma \vdash R\ p\ q : \mathcal{U}_i \mid \Delta}\ (\text{record})$$

$$\frac{\begin{array}{c}\Gamma \vdash R\ p\ q : \mathcal{U}_i \mid \Delta \\ \Gamma \vdash y_1.n_1 : A_1[p/\sigma, q/\alpha] \mid \Delta \qquad \cdots \qquad \Gamma \vdash y_k.n_k : A_k[p/\sigma, q/\alpha][n_1/x_1, \ldots, n_{k-1}/x_{k-1}] \mid \Delta\end{array}}{\Gamma \vdash \texttt{build}\ R\ p\ q\ \texttt{with}\ (y_1.n_1 | \ldots | y_k.n_k) : R\ p\ q \mid \Delta}\ (\text{build}_R)$$

The typing for $y_i.n_i$ means that A_i is of the form $A_{i1} \to A_{in}$, where the length of the variable vector y_i is $n-1$. In the derivation then, it is typed by:

$$\frac{\Gamma, y_{i1} : A_{i1}, \ldots, y_{i(n-1)} : A_{i(n-1)} \vdash n_i : A_{in} \mid \Delta}{\Gamma \vdash y_i.n_i : A_i \mid \Delta}$$

The projections are a generalisation of the pair projections.

Definition 7.18: Record Projection

For t and u term vectors;

$$\frac{\Gamma \vdash m : R\ t\ u \mid \Delta \qquad \Gamma \vdash R\ t\ u : \mathcal{U}_i \mid \Delta \qquad x_j \notin fv(A_i) \text{ for } j = 1, \ldots, (i-1)}{\Gamma \vdash p_i(m) : A_i[t/\sigma, u/\alpha] \mid \Delta}\ (p_i)$$

$$\frac{\Gamma \vdash_{\text{NEF}} m : R\ t\ u \mid \Delta \qquad \Gamma \vdash R\ t\ u : \mathcal{U}_i \mid \Delta}{\Gamma \vdash p_i(m) : A_i[t/\sigma, u/\alpha][p_1 m/x_1, \ldots, p_{i-1} m/x_{i-1}] \mid \Delta}\ (p_i^d)$$

7.3.3 Reductions for (Co)Inductive Types

For the inductive data types, the reductions are similar those seen in Chapter 5, although presented in a call-by-value fashion. The main difference is that the methods are pattern matched on the parameters and arguments to the constructor; this allows us to identify when the resulting term is NEF, and preserve the whether or not the target is NEF after reduction. The alternative would be allowing the methods to be functions that accept the parameters and arguments, but then the case analysis would reduce to a function application, which could not be NEF. This means we would lose the closure of NEF terms under reduction; a property we very much want to keep.

$$\begin{array}{rcl}\texttt{elim}\ (c_i\ \vec{p}\ \vec{a}\ \vec{b})\ \texttt{by}\ \vec{x}.\vec{m} & \longrightarrow & \texttt{let}\ \vec{x} = \vec{a}\ \vec{b}\ \left(\vec{w}.(\texttt{elim}\ b_1\ \vec{w}\ \texttt{by}\ \vec{m})\right) \cdots \left(\vec{w}.(\texttt{elim}\ b_n\ \vec{w}\ \texttt{by}\ \vec{m})\right) \\ & & \texttt{in}\ m_i \\ \texttt{case}\ (c_i\ \vec{p}\ \vec{a}\ \vec{b})\ \texttt{of}\ (\vec{x}.\vec{m}) & \longrightarrow & \texttt{let}\ \vec{x} = \vec{p}\ \vec{a}\ \vec{b}\ \texttt{in}\ m_i\end{array}$$

Where writing $\vec{x} = \vec{m}\ \vec{n}$ means the sequence of variables $\vec{x}$ is of the same length as the sequence $\vec{m}\ \vec{n}$, and each x_i is bound to the corresponding term on the right hand side.

Lazy Evaluation

A call-by-value strategy will not work for coinductive records; they represent (potentially) infinite objects, so we cannot evaluate each of its components in advance [38]. Thus, viewing coinductive records as a generalisation of the corecursive fixpoint `cofix` of [38], we need to use a lazy evaluation strategy, so that we only evaluate as much of the infinite object we need. Following [38], this also requires the introduction of specific contexts for lazy evaluation, that we label $\mathcal{L}$.

Definition 7.19: Lazy Evaluation Contexts

$$\mathcal{L} ::= \bullet \mid \mathcal{L}\{\kappa\} \mid \texttt{let}\ x = \texttt{build}(\vec{x_1}.n_1 | \cdots | \vec{x_k}.n_k)\ \texttt{in}\ \mathcal{L}$$

We generalise the lazy reduction rules of the `cofix` operator of dPA$^\omega$[38];

$$\begin{array}{rcl}
\kappa\{\texttt{build}(y_1.n_1|\cdots|y_k.n_k)\} & \longrightarrow & \texttt{let}\ x = \texttt{build}(y_1.n_1|\cdots|y_k.n_k)\ \texttt{in}\ \kappa\{x\} \\
\kappa\{\texttt{let}\ x = \texttt{build}(y_1.n_1|\cdots|y_k.n_k)\ \texttt{in}\ m\} & \longrightarrow & \texttt{let}\ x = \texttt{build}(y_1.n_1|\cdots|y_k.n_k)\ \texttt{in}\ \kappa\{m\} \\
\texttt{let}\ x = \texttt{build}(y_1.n_1|\cdots|y_k.n_k)\ \texttt{in}\ \mathcal{L}\{p_i(x\ \vec{m})\} & \longrightarrow & \texttt{let}\ y_i = \vec{m}, x = n_i\ \texttt{in}\ \mathcal{L}\{x\} \\
\texttt{let}\ x = \texttt{build}(y_1.n_1|\cdots|y_k.n_k)\ \texttt{in}\ \mu\alpha.[\beta]m & \longrightarrow & \mu\alpha.[\beta]\texttt{let}\ x = \texttt{build}(y_1.n_1|\cdots|y_k.n_k)\ \texttt{in}\ m
\end{array}$$

The way in which these rules achieve lazy evaluation is well explained by Miquey [52, p112]: The first two rules highlight that, when we reach a coinductive structure in a call-by-value context, we delay its computation by abstracting it, or keeping it abstracted, under a let expression variable; the third rule precisely corresponds to when the coinductive structure is linked to x, whose value is needed, so a single evaluation step is performed. The last reduction describes how control operators interact with coinductive structures.

7.4 Typing Algorithm

7.4.1 Weak Head Normal Form

We make extensive use of terms that are in *weak head normal form* (WHNF). We obtain the definitions of WHNF by combining those of [59] (for dependent types) and [10] (for μ-terms).

As described in[3], the idea of WHNF is that the term has been evaluated to the outermost data constructor or lambda abstraction. The fundamental idea is the 'head' of the term isn't reducible. For weak head normal forms not immediately in the litreature, we use the notion from Peyton [63, p198]; that in a term of the form $f\,e_1 \ldots e_n$, with $n \geq 0$, we must have: either f is a variable or data object; or f is a lambda abstraction and $n = 0$; or f is a built in function such that $f\,e_1 \ldots e_m$ is not a redex for $m \leq n$.

For our syntax, we consider $(\cdot,\cdot)$, $\texttt{in}_i(\cdot)$, $(\times)$, and $(\rightarrow)$ to be data constructors, so these are already in WHNF. π_i and `case` are eliminators, so we need to interpret them as built-in functions. As the idea of WHNF is we want the head of the term to dictate its behaviour, we need to make sure the eliminators don't reduce. For terms of the form $\pi_i(m)$, which reduces (for $m \rightarrow (n_1, n_2)$) to n_i. Thus, we want to make sure m doesn't reduce to a pair, so we need it to be in WHNF and that it isn't a pair. Similarly, for $\texttt{case}\ m \triangleright C\ \texttt{of}\ (n_1|n_2) \rightarrow n_i$, we need m to be WHNF but not of the form $\texttt{in}_i(n)$.

For equality, a term `refl` is already in normal form. For `subst` $m\ n$, we have a reduction (when $m = \texttt{refl}$) to n; so is only in WHNF when this is not the case; we can ensure this by having m be WHNF and not `refl`.

To determine when a μ term is in WHNF, we use the work in [10]: for a term $\mu\alpha.[\beta]m$ to be in normal form, we need to disallow reductions on the head. If $\alpha = \beta$, and $\alpha \notin fn(m)$, then we can apply the (μ_η) rule; so we certainly need either $(\alpha \neq \beta$, or $\alpha \in fn(H)$ in order for the term to be in WHNF. The other case to consider is the renaming reduction; when $m = \mu\gamma.[\delta]m')$, we are

[3] https://stackoverflow.com/questions/6872898/what-is-weak-head-normal-form

able to reduce the term by $\mu\alpha.[\beta]\mu\gamma.[\delta]m' \to \mu\alpha.[\delta]m'[\beta/\gamma]$. Thus, we must have the subterm $m \neq \mu\gamma.[\delta]m'$. For the subterm itself, m, we need it to be in WHNF.

The WHNF definition also needs to respect the various μ reductions. For example, a term $\pi_i(\mu\alpha.M)$ is (head) reducible, so this is not in WHNF.

The constants are trivially in WHNF, so we can give a full definition;

Definition 7.20: Weak Head Normal Form Terms [10, 59]

$$
\begin{array}{rcll}
H & ::= & x \mid \lambda x.t \mid (m,n) \mid \mathtt{in}_i(m) \mid \mathtt{refl} & \\
& \mid & Hm_1 \dots m_n & (H \neq \lambda x.m \text{ and } H \neq \mu\alpha.m) \\
& \mid & \pi_i(H) & (H \neq (m,n), (\mu\alpha.m)) \\
& \mid & \mathtt{case}\ H \triangleright z.C\ \mathtt{of}\ (x_1.n_1 | x_2.n_2) & (H \neq \mathtt{in}_i(m), (\mu\alpha.m)) \\
& \mid & \mu\alpha.[\beta]H & (\alpha \neq \beta, \text{ or } \alpha \in fn(H); H \neq \mu\gamma.[\delta]m) \\
& \mid & \mathtt{subst}\ H\ m & (H \neq \mathtt{refl}, (\mu\alpha.n)) \\
& \mid & (x : A) \to B \mid (x : A) \times B \mid A + B & \\
& \mid & \langle\rangle \mid \mathbf{1} \mid \mathbf{0} \mid \mathcal{U}_i &
\end{array}
$$

7.4.2 Bidirectional Algorithm

We have two types of judgements in this system [59]:

$$
\begin{array}{ll}
\Gamma \vdash m \Rightarrow A \mid \Delta \triangleright t & \text{(Type Inference)} \\
\Gamma \vdash m \Leftarrow A \mid \Delta \triangleright t & \text{(Type Checking)}
\end{array}
$$

Which we read: infer the type A for m, with output t; and check the type A against the term m, with output t. During the typing algorithm, the terms are sometimes (partially) evaluated to check their weak-head normal form or if they're NEF; this evaluated term is given by the output t.

The intuition behind the notation is that: for type inference, the type is inferred *from* the term, so the arrow goes from the term to the type; for type checking, the type is known, and checked *against* the term.

The bidirectional algorithm itself is largely standard, and is mostly guided by [59], with similarities to those in [47, 79, 64]. We highlight the new rules; the full presentation (including subtyping rules) can be seen in B.3. The rules for eliminations are split into their dependent and non-dependent versions, with the appropriate NEF checks, otherwise they are much the same as those in [59] and [64].

Soundness of the algorithm (wrt the type system) comes from the nature of bidirectional algorithms; it is directly derived from the type system. We certainly do not have completeness with this algorithm; in general, we can only type terms when we are given the initial type to check, and the term is in weak head normal form. Although this seems restrictive, this is in fact the usual case for type checking; as we define a function by declaring its type and then giving an inhabiting term.

Control

To type a (μ) term, we need to add the assignment $\alpha : A$ to the co-context. This means we have to know the type A before the typing, so the (μ) rule must be a type *checking* rule, so that it can be given the type A as input.

$$
\frac{\Gamma \vdash m \Leftarrow \mathbf{0} \mid \alpha : A, \Delta \triangleright t}{\Gamma \vdash \mu\alpha.m \Leftarrow A \mid \Delta \triangleright \mu\alpha.t}\ (\mu)
$$

As for the (*name*) rule, it's in fact quite similar to the ($\to E$) rule[4]. As we must already know the type of α (else it is not bound in the term) and *top*, we only need to check that m has a matches that type.

$$
\frac{\Gamma \vdash m \Leftarrow A \mid \Delta \triangleright t}{\Gamma \vdash [\alpha]m \Rightarrow \mathbf{0} \mid \alpha : A, \Delta \triangleright [\alpha]t}\ (name) \qquad \frac{\Gamma \vdash m \Leftarrow \mathbf{0} \mid \Delta \triangleright t}{\Gamma \vdash [top]m \Rightarrow \mathbf{0} \mid \Delta \triangleright [top]t}\ (top)
$$

[4] This intuition came from how the $\nu\lambda\mu$ calculus views a term $[\alpha]M$ as a 'continuation application', so can be seen as a modified form of application [70]

7.4.3 NEF Rules

These rules are written with $\square$, which can be replaced with either $\Rightarrow$ or $\Leftarrow$; meaning we can check a term is NEF whilst checking it against, or inferring, a type. The simplest way to implement this rule is to just check if the given term, m, is NEF.

$$\frac{m \in \text{NEF} \qquad \Gamma \vdash m \square A \mid \Delta \triangleright t}{\Gamma \vdash_{\text{NEF}} m \square A \mid \Delta \triangleright t}\ (\text{NEF}I) \qquad \frac{\Gamma \vdash_{\text{NEF}} m \square A \mid \Delta \triangleright t}{\Gamma \vdash m \square A \mid \Delta \triangleright t}\ (\text{NEF}E)$$

RNEF

From the user's perspective, however, this can lead to a very restrictive language. A simple example is to consider the term $M := m\ (n\ p)$, with $m : (x : A) \to B$. For M to be typeable, we must have $(n\ p) \in$ NEF, but an application can't be NEF. If we know n is of the form $x.n'$, and that $n', p \in$ NEF then $n\ p \to n'[p/x] \in$ NEF. In this case, $M \to m\ n'[p/x]$, which is typeable in our system. This motivates the need to consider the class of terms RNEF; terms that *reduce* to NEF terms.

Definition 7.21: RNEF

We say $m \in$ RNEF (NEF-reducible) when there exists a term $n \in$ NEF such that $m \to^* n$.

We could now consider the rule;

$$\frac{m \to^* n \in \text{NEF} \qquad \Gamma \vdash n \square A \mid \Delta \triangleright t}{\Gamma \vdash_{\text{NEF}} m \square A \mid \Delta \triangleright t}\ (\text{NEF}I)$$

This idea is explored by Lepigre in [43], where they have a value restriction for their dependent typings; they suggest allowing the user to write a non-value term, and then have an elaborator that attempts to normalise the term to a value. This of course needs some susbsystem that checks if a term is normalising [43] (which must be a conservative check, as this is undecideable in general [73]).

Performing an evaluation that the user doesn't see could lead to a disconnect between what the user intends to mean with a term, and what type is actually derived. We give an example to explain this, based on a modification of Herbelin's term used to show the degeneracy of dependent pairs in [39].

> **Example 7.4.1.** Consider $P := \mu\alpha.[\alpha]((0,0), \mu\delta.[\alpha]((1,1),(\mathtt{refl},\mathtt{refl})))$. We could have $\pi_1(P) \to (0,0) : \mathbb{N} \times \mathbb{N}$, but $\pi_2(P) \to (\mathtt{refl},\mathtt{refl})$ cannot be typed, as $P \notin$ NEF, and P is in normal form, so there isn't a P' such that $P' \in$ NEF. In the naive system, this would be typed by $\pi_2(P) \to (\mathtt{refl},\mathtt{refl}) : (0 = 1) \times (0 = 1)$
>
> If we now consider $\pi_i(\pi_2(P))$, observe that $\pi_i(\pi_2(P)) \to \pi_i(\mathtt{refl},\mathtt{refl})$.
>
> Of course this term is now typeable, and would have the type $(1 = 1) \times (1 = 1)$; this mightn't be immediately obvious to the user.

To avoid allowing terms like this to be RNEF, we might have to consider a reduction $\to_{\not\mu}$, which stands for the usual reduction relation $\to$, without the μ-reduction (structural reduction). Then, we would check $m \in$ RNEF if there is some $n \in$ NEF such that $m \to_{\not\mu} n$.

NEF and RNEF Functions

We obviously want users to be able to define functions. If we use the simple NEF rule when type checking, we get a similar problem as above when using a function as an argument. A trivial example is $\mathtt{id}\ x = x$. If we naively use the definition of NEF, then we could consider $\mathtt{id}\ x \in$ RNEF. But it should be obvious that when applied to a term m, we only have $\mathtt{id}\ m \in$ RNEF when $m \in$ RNEF.

Generalising, consider a user-defined function $f : (x : A) \to B$ (where $A : \mathcal{U}, B : A \to \mathcal{U}$). We now consider the term $P := m\ (f\ a)$, where $m : (y : B\ x) \to C$, $C : (B\ x) \to \mathcal{U}$. If y occurs free in C, then we must have $(f\ a) \in$ RNEF. How can we check this?

The most simple solution is to simply evaluate $f\ a$ to RNEF using the definition of f. During type-checking, f is effectively *inlined* every time it appears in the term being checked – this makes type-checking a slow operation.

Instead of a termination checking algorithm, we have briefly explored an alternative way to check RNEF. We consider our usual type system and algorithm, as well as a 'naive' type system and algorithm that are exactly the same, except the the NEF rules, which always say a term is NEF:

$$\frac{\Gamma \vdash m : A \mid \Delta}{\Gamma \vdash_{\text{NEF}} m : A \mid \Delta}\ (\text{NEF}I')$$

The idea is to first type-check a term m in the naive system, and if m passes the check, normalise m to n, and then perform the type check on the n (if n is NEF, else fail). This relies on the (unproven) property that well-typed terms are normalising in the naive system. This would be a reasonable assumption to make, as any well-typed term in (non-classical) **ECC** is known to be normalising [45, chp4, p100], although this would certainly be complicated by the control operators and, if included, the coinductive structures.

8 | Implementation

At the beginning of this project, implementation of even just a propositional theorem prover seemed daunting; how do we represent variables, or check names are fresh? What's a monad? Fortunately, there are many fantastic learning resources focused specifically on the *implementation* of (dependently) typed functional languages, not just the theory they're based on. In particular, the author found the lecture series 'Designing Dependently-Typed Programming Languages' [78] (with the code in [79]) incredibly useful – and we recommend the interested reader check it out.

For a high level overview of our implementations, we describe a typical REPL cycle given by Figure 8.1.

Parse to AST The user-level syntax is parsed to our AST representation. This desugars some of the syntax, for example, a lambda expression `\ x y -> ...` is converted to a representation `Lam x (Lam y ...)`. This intermediate stage keeps some syntactic sugar, in particular for negation `¬A`, and if and only if; `A <-> B`.

To Core Syntax The AST is then converted to the core term representation, where variable names are converted strings to the `VName` type, which, using the `unbound` library, lets us bind names appropriately.

Typecheck The type-checking algorithm for each language is used here. It works by sequentially type-checking the declarations, and then adding them to the context if checking succeeds, or throwing an error if checking fails.

Update Context The context that the user can reference in the interactive commands is updated with the definitions in the current file.

Prepare Results Depending on what command the user gave, the output of the type-checking will be prepared by pretty printing. If the user type-checked a file, then the REPL will report back any holes left by the user. If the user asked to type check a term, then its (pretty-printed) type will be returned. When an error is thrown for any command, the REPL is able to print this nicely back to the user. Errors can occur during parsing and typechecking, and this is made apparent to the user.

We also describe the two typechecking cycles, given in Figures 8.2 and 8.3:

Cycle for Propositional Prover For a declaration `f : A, f = m`, the `Principal Type` algorithm will infer the type of `m`, that we will call `B`. `Instantiate to Signature` will then attempt to apply substitutions to `B` to instantiate it to `A`. If successful, then `f : A` is added to the context, and the algorithm will type the next declaration. Once all declarations are type-checked, or if an error is thrown, the typechecking exits.

Cycle for Dependent Prover For a declaration `f : A, f = m`, `Type Check` will execute bi-directional type checking of `m` against `A`. When a term needs to be evaluated to WHNF, the algorithm uses `WHNF`, which uses the evaluator on the head of the term until it is in WHNF, and then returns it back to the type-checker. If the algorithm needs to check if a term is RNEF, then it checks this through the `NEF` interface, which uses the same evaluator to try and normalise the term. If the declaration successfully type checks, then it is added to the context, and the algorithm repeats on the next declaration. Once all declarations are type-checked, or if an error is thrown, the typechecking exits.

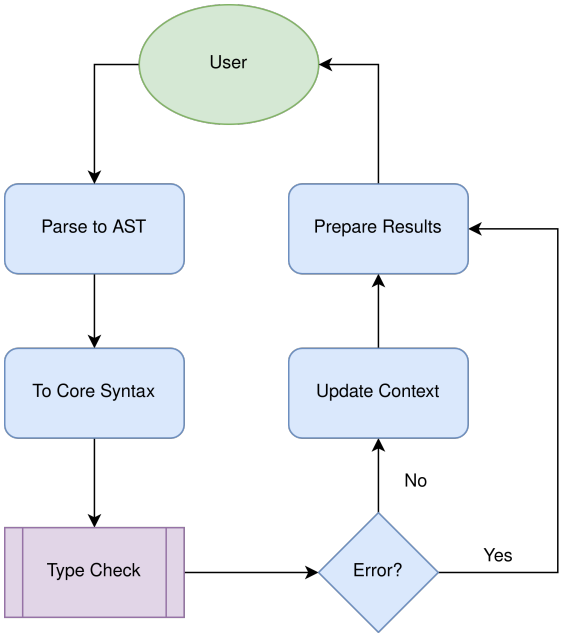

Figure 8.1: The main cycle

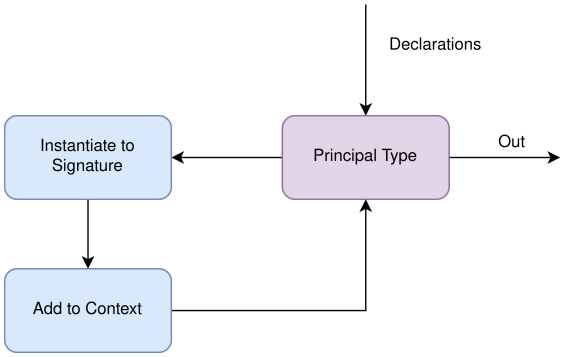

Figure 8.2: Type checking loop for the propositional prover

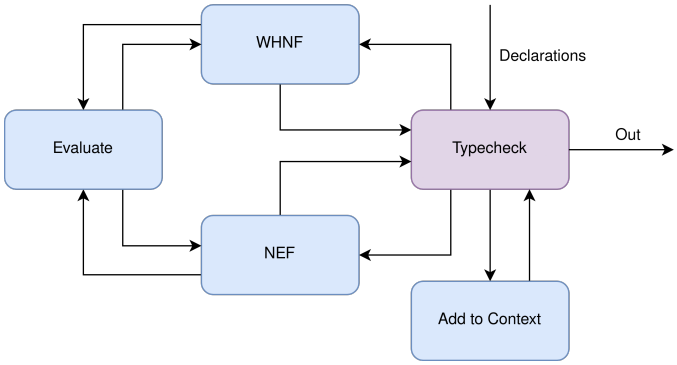

Figure 8.3: Type checking loop for the dependently typed prover

8.1 Variables and Representing Terms

Throughout the theoretcal discussion, we have been using Barendregt's convention for free and bound variable names. This has made our discussions much simpler, as we don't have to constantly worry about α-equivalence. When implementing variables in a language, however, this is a difficult convention to enforce on the user; and α-equivalence for a naive implementation can be very slow.

Luckily, there are ways to represent terms that avoid these pitfalls, both allowing the user not to worry about variable capture, and having speedy α-equivalence.

8.1.1 De Bruijn Indices

De Bruijn Indices [26] are a way to represent terms that allows for computationally efficient checking for α-equivalence. Instead of using unique names to bind variables, we instead have occurences of variables reference what they are bound by; they are now indices (numbers) referring to how many binding 'levels' their binder is. 0 would represent the closest binder (λ), 1 would represent the next, and so on. For example, the term $\lambda xyz.xz(\lambda w.z)$ in De Bruijn form is given by $\lambda(\lambda(\lambda\ 2\ 0\ (\lambda\ 1)))$. As the variables names don't appear in the terms, we call this a *nameless* representation.

The most immediate problem, one might notice, is that these nameless terms aren't very readable. Systems using De Bruijn indices will usually have a nameful user language, and then use the nameless terms under the hood; shielding the user from the nasty numbers.

8.1.2 Unbound – Locally Nameless

We use the library unbound-generics [42], which is based on the work of [80]. It uses a *locally nameless* term representation; which marries the nameful and nameless representations. Simply put, bound variables are represented by de Bruijn indices, and free variables by atoms (usually strings) [80]. This means our variables and binders have a nameful interface, so the user language can stay closer to the core language, and it makes the implementation much more intuitive.

The indices are a pair of natural numbers, represented by $i@j$; representing the j^{th} variable in the i^{th} binder (this is to allow multiple variables to be bound in a single binder). For example; a term $\lambda xyz.\lambda w.z$ can be represented by $\texttt{Bind}(x,y,z).\texttt{Bind}(w).1@2$.

Checking alpha equivalence only needs comparing the structure and the indices, we don't need to worry about the actual variable names. For example, $\texttt{Bind}(\vec{x}).i_1@j_1 =_\alpha \texttt{Bind}(\vec{a}).i_2@j_2$ only requires checking that the telescopes $\vec{x}$ and $\vec{a}$ are of the same length, and that the indices are equal; $i_1 = i_2$, $j_1 = j_2$.

The advantage of this method, over simple de Bruijn Indices, is we retain the original variable names in the binders, which makes it easier to return the terms to the user (e.g. for error reporting); we can use the same names they originally gave. If the user inputs a term like $\lambda x.\lambda x.m$, then unbound is able to generate a fresh name for the second x binding, (and all free occurences of x in m are bound to this second x), which keeps the binding unambiguous.

When type checking, we need to be able to descend down a bound term. The `unbind` function lets us do this by replacing all the bound indices by the (now free) variable being unbound. For example, for the term $\texttt{Bind}(x).\texttt{Bind}(y).1@0\ M$, calling `unbind` gives us the variable name x, and the term $\texttt{Bind}(y).x\ M$; note that x is now free in this term. In case we want a binder to carry extra information (like a type annotation), we can `embed` this into the binder.

As lambdas (and other similar binding term constructors) a defined via the `Bind` type constructor, one cannot descend to the subterm of a lambda without *unbinding* the term. In our definition of terms, lambda abstractions have the constructor `Lam Bind (VName, Embed TyAnnot) Term`; which means that the variable name (of type `VName`) is bound over the subterm and the type annotation is stored along with the variable name, without that name being bound over the annotation.

A constructed term will pattern match with `Lam bnd`, where we explicitly have to call `unbind bnd` to obtain `((x, unembed -> ma), m)`. Note that the only way to get to the subterm `m` is by unbinding the lambda; this means we can't descend to subterms without handling the bound variable, which makes it hard to forget to add their type to the context. To construct a lambda term, we need have a variable `x`, an (optional) type annotation `ma` and a subterm `m`. Then, we just

bind the variable over the subterm (and embed the annotation) by `Lam (Bind (x, embed a) m)`. The unbound library will then switch free occurences of `x` in `m` to be bound, that is, indexed to point to this current binder. Similar to unpacking a lambda term, we can't construct a lambda abstraction without explicitly binding a variable over the subterm.

Another popular variable handling library is bound[1]. We found this less intuitive, as substitution was defined via the monad typeclass – which the author wasn't very experienced with at the start of this project – and terms are parameterised by the type(s) of their variables. bound also has some difficulties with telescopes [47, p50], which are needed for data and record definitions, whereas unbound has support for nested binding (thus telescopes) built in [80, p1]. Put together, this makes unbound a much more attractive library for implementing a dependently typed language.

8.1.3 Using Unbound

The fantastic thing about the unbound library is that all the machinery we need – term substitution, free variables, fresh variables, alpha equivalence, etc... – is all handled automatically. All we need to do is define which parts of our terms are the variables.

After defining variable names by `VName`, we make our Term and Type data types instances of the `Alpha` (for determining two terms/types alpha-equivalent) and `Subst` (for showing how to substitute variables for terms/types) classes. Luckily, unbound-generics uses haskell's `GHC.Generics` library to make defining these instances very simple; the `Alpha` instance can be automatically derived, and the `Subst` instance only requires we identify which of our terms are variables, and it then figures out the rest.

We get all the machinery we need for variables and substitution in 6 lines of haskell;

```
type VName = Name Term
data Term  = Var VName | ...

instance Alpha Term where

instance Subst Term Term where
    isvar (Var v) = Just (SubstName v)
    isvar _       = Nothing
```

For readability reasons, we add the type synonym `type Type = Term`; this makes it easier to tell what the various functions are doing from their type signatures. For example, our type checker returns a type `(Term, Type)`, representing a term and its type; this wouldn't be as obvious were the type written `(Term, Term)`, even though these are equal.

8.1.4 Structural Substitution

As part of the implementation, we developed a generic structural substitution type class to work with `unbound`, as this wasn't (easily) supported by the library. This was achieved by modifying a copy of the `Subst` class, creating a `StrSubst` class. Instead of a substitution fuction: `subst :: Name b -> b -> a -> a` (where `a` and `b` in our case are `Term`), we allow the user to dictate what happens to the subterm by parsing a function instead of a term. All the user needs to do is say when a term is a named term via `isStrvar`.

```
class StrSubst b a where
    strsubst :: Name b -> (b -> b) -> a -> a
    isStrvar :: a -> Maybe (StrSubstName a b)

data StrSubstName a b = (a ~ b) => StrSubstName {
    strCstrct :: Name b -> a -> a,
    strSubVar :: Name b,
    strSubTm  :: a
    }
```

[1] https://hackage.haskell.org/package/bound

Where `strSubVar` is the field to get the name of a named term, `strSubTm` returns the subterm, `strCstrct` returns the constructor for the named term. This could be further generalised, by allowing multiple subterms, and a constructor that uses those multiple subterms; but this implementation was enough for our purposes.

Following the style of `unbound`, we also made a structural substitution type class for haskell generics; `GStrSubst`. This means that, as `Term` was an instance of `Generic`, we get automatically derived definitions for structural substitution. Our instance for structural substitution, then, looks like;

```
instance StrSubst Term Term where
    isStrvar (N b m) = Just (StrSubstName N b m)
    isStrvar _       = Nothing
    subName a b m = N b m
```

Then, to implement the different structural substitutions, we just pass the term constructor that will be applied to each subterm;

$$
\begin{array}{rcl}
v(\mu\alpha.m) \to \mu\alpha.m[[\alpha]v\bullet/[\alpha]\bullet] & \Longrightarrow & \texttt{strsubst a (App v) m} \\
(\mu\alpha.m)n \to \mu\alpha.m[[\alpha]\bullet n/[\alpha]\bullet] & \Longrightarrow & \texttt{strsubst a (`App` n) m} \\
\pi_i(\mu\alpha.m) \to \mu\alpha.m[[\alpha]\pi_i(\bullet)/[\alpha]\bullet] & \Longrightarrow & \texttt{strsubst a (Proji) m} \\
\text{case } \mu\alpha.m \triangleright z.A \text{ of } (x_1.n_1|x_2.n_2) \to & & \\
\mu\alpha.m[[\alpha]\text{case } \bullet \triangleright z.A \text{ of } (x_1.n_1|x_2.n_2)/[\alpha]\bullet] & \Longrightarrow & \texttt{strsubst a (\textbackslash p -> Case p zA n1 n2) m}
\end{array}
$$

8.2 Syntax

8.2.1 Parsing

There are a few lexer/parsers for haskell; including parsec [21], alex/happy [17, 32], BNFC [1] and megaparsec [22]. Parsec is considered the 'main' parsing library for haskell, but it is not being actively developed. Alex and Happy are another common way to parse in haskell, but they are written in (slightly) different languages; given the author was also learning a lot of haskell for this project, learning other languages felt a bit overkill, given the alternatives.

We very briefly tried using the BNFC library [1], which generates a compiler based on a labelled BNF grammar. At first, this idea seemed very attractive. However, we found it very difficult to even get the grammar correct, as it didn't particularly enjoy the haskell-style syntax we were aiming for. The reliance on its generation also made the workflow slower; each time we made a change, we had to recompile the parser and test it through either a generated executable or by calling it from our own code (which was difficult in itself when using the stack tooling). Given these difficulties, and the lack of up to date documentation, we quickly abandoned this library.

We settled on megaparsec [22], an active fork of parsec. In addition to its solid foundations in the well-established parsec library, it had by far the best tutorial we could find out of any of the parsers[2]. This let us quickly code up the parsers for both implementations, so we could spend much more time focusing on the fun stuff.

The parser is built from small combinators defined in the megaparsec library, which are combined into larger combinators that handle each term/type constructor. In turn, these are used to create large term and type parsers, which are used in the declaration parsers.

8.2.2 User Syntax

Perhaps the hardest part of the entire project was coming up with an intuitive ascii representation of $\mu\alpha.[\beta]m$. After a few alternatives (visible in the code's README), we decided on the syntax \a : b\. The intuition is that it's similar to the usuall haskell style for lambda binding, \x -> ..., but the ':' represents the 'switch' to the name 'b'. We hope that the reader finds this syntax agreeable.

[2] Found here: https://markkarpov.com/tutorial/megaparsec.html

As for the rest of the syntax, we aimed to mimic that of Agda [60]. The main reason for this choice is to make it as easy as possible for many users to try out the implementations. It's also an attempt not to add to the clutter of similar but different programming languages.

8.2.3 AST and Parser Errors

The input is parsed into an AST, rather than directly into a core term. This allows us to have some simple syntactic sugar on the user level; in particular, the ↔ and ¬ symbols, and binding multiple variables under one lambda, like in \x y z -> ... .

The AST is then converted to core syntax under the `ASTM` monad, which is able to report errors not immediately picked up by the parser; for example a function signature without a definition (or vice versa). The monad also passes the source locations of the original code onto the core syntax, which helps to report more useful errors back to the user if they occur in the type checking stage.

8.3 Type Checking Monad

Both typing algorithms operate under a monad, `TcMonad`, that is able to maintain the current typing environment and type assignments, as well as error reporting. This monad is largely inspired by the one used in [79].

Errors are reported under the `Err` data type, which contains the error message and the location of the error in the source code.

`Env` is used to represent the current environment: local (variable) type assignments are stored in `ctx`; the type assignments of the covariables are in `coctx`; the types of functions previously defined and type checked are in `globCtx`; the source locations of the currently checked terms/types are in `srcLoc`.

`TcState` is used in both provers to keep track of the holes left by the user. In the propositional prover, it also keeps track of the type substitutions returned from unification.

```
data Err = Err [SourceLocation] String

data Env = Env {
    ctx     :: Ctx,
    globCtx :: Ctx,
    coctx   :: CoCtx,
    srcLoc  :: [SourceLocation]
}

newtype TcState = TcState {
    holes :: [FilledHole]
    subs :: [TypeSub] -- Only in the Propositional Prover
}

type TcMonad = FreshMT (StateT TcState (ReaderT Env (ExceptT Err IO)))
```

`TcMonad` is constructed via various *monad transformers*; they essentially equip a given monad with extra functionality. This design builds on the type checking monad of [79].

- The core of the monad is the haskell `IO` monad; it allows for user input and printing.
- `ExceptT` equips the `IO` monad with the ability to catch and throw errors of type `Err`. This is especially useful, as we need to be able to form different errors depending on how type checking has failed.
- `ReaderT` adds the ability to acces the `Env` data. Although this might seem stateful, we keep it under the `ReaderT` transformer as it has a handy function `local`, which allows us to make recursive calls under a modified `Env`, without changing the `Env` of the calling function; for example, when adding the type of a variable to the context, after obtaining it from

opening a lambda binding. If `Env` were kept under the `StateT` transformer, we would have to manually modify and then unmodify the state on each recursive call.

- `StateT` allows stateful use of the `TcState`; letting the type checker modify and access the state. We use the state to store (checked) function definitions and type signatures, and to save the record/data types and their constructors/destructors.
- `FreshMT` is from the unbound library; it allows us to generate fresh (unique) variable names when needed. This is especially helpful in the principal typing algorithm; which relies very heavily on fresh variable generation. Under the hood, it keeps track of the lowest index that hasn't yet been used, incrementing it each time a fresh name is needed[3].

`TcMonad` has kind `* -> *`; so it is implicitly parameterised by a type variable. During type checking, it is typed by what we want the type checking function to return. This issually a pair (`Term, Type`); representing a term and its checked type.

8.4 Simply Typed Theorem Prover

We fully implemented the principal pairing algorithm of Chapter 6 as the type system for a functional language. This was achieved in a little over 2000 lines of Haskell, with the core type-checking and term represenation implemented in about 1200 lines. The compact nature of this implementation makes it much simpler to check its correctness. This is a technique employed by many proof assisstants out there today; to have a small, trusted core that everything else is built on top and simplifies/is elaborated to [4].

The code reads a lot like Haskell; users declare a function by giving a type signature and then a definition. The full syntax as BNF can be seen in A.2. Lambda expressions are given by `\x -> m`, *control* expressions, $\mu\alpha.[\beta]m$ are given by `\a :b\ m` – we don't force the user to use Greek names for the co-variables. If the same names are used for a lambda and control binding, then the innermost name is used. For example, in:

```
f x = \a:_\ \a -> x a
g x = \a -> \a:_\ x a
```

`f` will typecheck, as `a` is bound to the lambda abstraction `\a ->`. `g` will not typecheck, as it is bound by `\a :_\`; so is seen as a covariable, with type stored in the co-context. This means it won't have a type in the current context, and will be seen as out of scope.

8.4.1 Name Polymorhpism and Type Instantiation

```
axiom : A -> A
axiom x = x

arrE : (A -> B) -> A -> B
arrE = ax
```

We can see that `arrE` calls `ax` with its type instantiated as `(A -> B) -> (A -> B)`. This highlights a nice feature of the principal pairing algorithm when used for name polymorphism. It's not much work (from an implementation perspective) to instantiate the type `A -> A` to the one needed – we don't need an extra subtyping algorithm to check that this instantiation is safe.

In general, the type signature of a name is checked to be valid by trying to *instantiate* its principle type to that of the type signature. The instantiation is achieved through a modification of Robinson's unification algorithm, only allowing the principal type (and not the type signature) to have substitutions applied to it. If we call the first argument the *scrutinee*, and the second argument the *target*, we can more precisely say that the instantiation algorithm is achieved by restricting `unify` to only allow substitutions to the scrutinee.

[3] As explained in:
https://hackage.haskell.org/package/unbound-generics-0.4.1/docs/Unbound-Generics-LocallyNameless-Fresh.html

In the case above, when typing `arrE`, we will get its principal type as `A' -> A'`. To check this type against the signature given, we call: `instantiate (A' -> A') ((A -> B) -> A -> B)`.

By using `instantiate`, and not `unify`, we avoid allowing the type signature to be more general than that of the principal type. Indeed, `instantiate ((A -> B) -> A -> B) (A' -> A')` will fail, as it will attempt to find `instantiate (A -> B) A'`, which fails; there is no way to apply substitutions to `(A->B)`, to obtain `A'`.

8.4.2 Expressing Logic

As the typing algorithm is based in propositional logic, structures like data types and type constants aren't present in the language; there wouldn't really be a way to reason about them. Recursion isn't available either, as there are no structures to recurse on.

As a logical system, we are fully able to reason about classical propositional logic. For example, we are able to prove the law of the excluded middle:

```
lem : (A + ¬A)
lem = \a:a\ (in2 (\x -> \_:a\ (in1 x)))
```

We show the encoding of the standard rules for natural deduction in A.2.1.

8.4.3 Diagram

8.5 Dependently Typed Theorem Prover

The implementation of $\mathbf{ECC}_{\lambda\mu}$ is achieved in a little over 3000 lines of Haskell Code. The type checking and evaluation, under the folder `Core`, is implemented in around 1500 lines of Haskell. As explained for the simply typed language, it is ideal that the implementation is small.

The full syntax can be seen in B.5.1, and is reminiscent of Agda's syntax, with a few minor differences.

8.5.1 Evaluation

Evaluation is necessary for the type-checking algorithm, in both evaluating to WHNF and checking if terms are RNEF. We implemented evaluation by rewriting rules. These were based on `unbound`'s `Subst` type class, which lets us define, for example, reduction of let expressions by:

```
eval Let bnd = do
    ((x,m),n) <- unbind bnd
    v <- toValue m
    return (subst x v n)
```

which represents the reduction $\texttt{let}\ x = v\ \texttt{in}\ n \longrightarrow n[v/x]$. The structural substitution was implemented via our `StrSubst` type class, which we discussed earlier in this chapter.

It should be noted that, while a simple way to implement evaluation, substitution is quite inefficient. As explained in [47], we could increase performance by *normalisation by evaluation* [44], where terms are converted to Haskell values, computed in Haskell, and then turned back into the terms of the language. Due to time restrictions, we felt that performance wasn't much of a priority, so we stuck with the simple (structural) substitution methods for evaluation.

8.5.2 (Co)Data

Unlike in the presentation of Chapter 7, constructors and projectors are not treated as built-in functions, and are instead represented separately. Crucially, they must be fully applied, and will produce an error if otherwise. The reasoning behind this is it allows us to distinguish between generic term application, and constructor/projector application, which in turn lets us identify when the application of constructors and projectors are NEF. This design choice separates it from Coq and Agda, but with good reason.

Defining data and records is very similar to how one would in Agda:

```
data Vector (A : Univ) : (n : Nat) -> Univ where
    empty : Vector A 0
    cons : (n : Nat)(x : A)(xs : Vec A n) -> Vec A (suc n)

record Stream (A : Univ) : Univ where
    head : A
    tail : Stream A
```

The main difference with Agda is that projectors and constructors use telescopes rather than function types in their definition. This emphasises the fact that constructors aren't introducing generic functions to the context, and instead specifically need to be given all their respective arguments.

Data and record types are handled via case and build trees:

```
headOrZero : (n : Nat)(A : Univ) -> (Vec A n) -> A
headOrZero n A v = case v of
    empty -> 0
    cons i x xs -> x

from : Nat -> Stream Nat
from n = build
    head -> n
    tail -> from (suc n)
```

Due to the time scope of this project, we were unable to implement termination checkers for recursive functions, so a user would be able to define a function `f = f`. As we've explained previously, there are known methods to conservatively check for termination, and, in future, it will be almost certainly possible to apply these methods to this language.

8.6 REPL

Finally, to encompass the 'assistant' aspect of proof assistants, both languages have a REPL that let the user interact with the type checker through *holes*.

A user can leave holes in functions by marking them with a number, for example, `?1` marks a hole with index 1. The indices are used so the user is able to leave multiple holes and know which type reported by the type checker corresponds with which hole. A simple example of this would be:

```
foo : A -> A -> A
foo x = ?0

bar : B -> C -> D
bar x = \y -> ?1
```

The user can interact with the REPL by loading the file with `:l <filename>`, and then checking through the file for holes by the command `:r`. In our example, the REPL will report back:

```
Hole 0:
    Goal: A -> A
    Scope: x:A;
Hole 1:
    Goal: D
    Scope: {y:C, x:B};
```

The goal is the type that the term replacing the hole should have, the scope represents the variables that are in scope at the hole.

For the propositional language specifically, the user is able to exploit the inference power of the principal pairing algorithm by asking it to infer the type of a closed term. If a file is loaded, the user is also able to reference functions in said file:

```
> :t \x -> \y -> x y
\x -> \y -> x y : (A3 -> A4) -> A3 -> A4
> :t \x -> x x
SourcePos 1:7
Unify: Unable to substitute: type A1 occurs in type A1 -> A2
In the expression: x x
> :t \x -> \y -> foo x
\x -> \y -> foo x : A1 -> A3 -> A1 -> A1
> :t \x -> \y -> in1 (bar x)
\x -> \y -> in1(bar x) : A1 -> A3 -> (C -> D) + A5
```

On the other hand, the dependently typed language is able to evaluate terms via `:e`:

```
    > :e (\x -> \y -> y x) foo (\z -> z)
    foo
    > :e (\a:_\ \x -> y (x, \_:a\ z)) w
    \a:_\ \x -> y (x, \_:a\ z w)
```

9 | Evaluation

In this chapter, we will evaluate the work we have achieved in this project. We will split our discussion into theory and implementation for both the propositional and dependent calculi. A more general point that applies across the project, is perhaps about the disconnect between the work done for $\lambda\mu^N$ and $\mathbf{ECC}_{\lambda\mu}$. From the author's perspective, it felt very much like working on two different sub-projects. We do think, however, that it was still very much beneficial to have started the project working on the simply typed $\lambda\mu^N$, as it allowed us to focus on the control operators in a familiar setting, without dependent types muddying our intuition.

9.1 $\lambda\mu^N$

9.1.1 Theory

We successfully found a principal pairing algorithm for the $\lambda\mu$-calculus, as well an extension to include named functions, product types and sum types. Soundness and completness of the algorithm were also proved, meaning we can be sure that the algorithm is suitable for checking proofs of statements in propositional logic. Using a principal pairing algorithm, as apposed to a bidirectional algorithm, also allows the type for a term to be inferred without any hints; this makes life easier for the user, and also improves readability of definitions.

The addition of name polymorphism to the calculus allows for proving smaller lemmas that can be re-used in larger proofs, saving the user from having to rewrite the proofs of the lemmas in order to use them.

The separation of the theoretical work from the implementation has the advantage that the algorithm for the μ and $[\cdot]$ terms can be used in other languages that are based on principal pair/type algorithms, like ML.

9.1.2 Implementation

Although the algorithm is able to infer the type of a given term, for ease of parsing the user must still give a type signature for a function before its definition. From the perspective of the theory this is an arbitrary limitation, but it is encouraged practice in Haskell-like languages to define a functions type before its definition. A more practical language would certainly drop this restriction and let the system infer the type of a function without a signature.

The implementation does not feature program evaluation. This separates it from languages like Agda, in which files can be executed. This does relegate the prover to only being able to check that a program is well typed, rather than being able to run said program. We argue that this isn't much of an issue; it's well known how to implement evaluation of λ-calculus terms, and these ideas should be expandable to $\lambda\mu$-calculus, perhaps using the work behind the `call/cc` control operator of the Scheme language.

9.2 $\mathbf{ECC}_{\lambda\mu}$

9.2.1 Theory

We used the ideas behind $\mathbf{ECC}_{\mathbf{K}}$ [53] to generalise dPA^{ω} to arbitrary types, not just functions on $\mathbb{N}$. The achievement of the core calculus, when compared with $\mathbf{ECC}_{\mathbf{K}}$ is the addition of coproducts, found by generalising those in dPA^{ω}. We also generalised dependent pairs and coproducts to inductive families and records, which allowed us to know when we allowed their dependent elimination. Put together, this means we have successfully defined an expressive calculus with dependent types and control operators, capable of reasoning in classical first order logic.

During our work in trying to add coproducts and data types, we devised a technique to reason about when new term constructs are NEF, and when we allow their dependent elimination. To

check when a term construct c was NEF, we looked at its reduction to a term d for which we know what restrictions we need on d to determine $d \in \text{NEF}$, and then applied these same restrictions to c. To check when we allow dependent elimination of these constructs, we checked the conditions of the reduced term. If our elimination term c reduces to d, we allow c to be typed only when we allow d to be typed.

For example, to allow coproducts in $\mathbf{ECC}_{\lambda\mu}$, we checked when we can allow dependent elimination of coproducts by inspecting the reduction of case constructs. We know that the term $\texttt{case}\ \texttt{in}_i(m)\ \texttt{of}\ (x_1.p_1|x_2.p_2)$ will reduce to $(x_i.p_i)m$. We also know we allow dependent elimination of an application tu only when u is NEF. Thus, we only allow dependent elimination of coproducts when the target term is NEF. To know when a case construct is NEF, we used the fact that $\texttt{case}\ \texttt{in}_i(m)\ \texttt{of}\ (x_1.p_1|x_2.p_2)$ will reduce to $p_i[m/x_i]$, and thus we know, by 7.8, this is NEF for p_i and m NEF.

We were able to prove important properties of the calculus, in particular subject reduction. However, our proof of its consistency was only a sketch, and a formal proof is certainly needed. This would likely rely on a proof that well typed terms in $\mathbf{ECC}_{\lambda\mu}$ are strongly normalising, as this was used to prove the sequent calculus version of dPA^{ω} was normalising [52]. We do already know that **ECC** is strongly normalising, and that we are able to encode $\mathbf{ECC}_{\lambda\mu}$ into $\mathbf{ECC}_{\mathbf{K}}$, which in turn can be encoded into a polarised sequent calculus $\mathbf{L}_{\mathbf{dep}}$, which is strongly normalising [53]. However, we haven't proved that the translation from $\mathbf{ECC}_{\lambda\mu}$ preserves types and reductions, so it is quite tenuous to assert normalisation of the calculus based on this.

We successfully defined a bidirectional algorithm for $\mathbf{ECC}_{\lambda\mu}$, including novel rules for typing the control operators μ and $[\cdot]$. Bidirectional type checking, although easier to implement, tends to need a lot more annotations from the user than algorithms based on *pattern unification* [49, 35], which are much closer in inference power to the principal pairing algorithm [47]. This does leave our implementation somewhat weaker in type checking ability than Agda and Coq.

Some Classical Examples

We briefly explore some examples in $\mathbf{ECC}_{\lambda\mu}$, to see exactly what sort of a logic we can get.

To start with, we are able to prove (LEM);

$$\lambda a.\mu\alpha.[\alpha]\texttt{in}_2(\lambda x.\mu\delta.[\alpha]\texttt{in}_1(x)) : (A : \mathcal{U}) \to (A + \neg A)$$

This proof holds for arbitrary types A; there is no restriction to, say, A being in $\mathbb{P}$. Certainly, then, the corresponding logic is classical in nature, suggesting we have stuck to our goal of theorem proving for classical logic.

More interestingly, we are able to prove the equivalent of the first order classical property: $\neg\forall w.B \to \exists w.\neg B$. The derivation of this type assignment can be found in B.4.

$$P := \lambda x.\mu\alpha[top]x(\lambda y.\mu\beta.[\alpha](y, \lambda z.\mu\delta[\beta]z)) : \neg((w : A) \to B) \to (w : A) \times \neg B$$

We are still able to prove the intuitionistic reverse: $\exists w.\neg B \to \neg\forall w.B$:

$$Q := \lambda xy.\pi_2(x)\ (y\ \pi_1(x)) : (w : A) \times \neg B \to \neg((w : A) \to B)$$

Meaning we have the classical notion $\exists w.\neg B \leftrightarrow \neg\forall w.B$, with the proof given by (Q, P).This suggests we do have a calculus capable of classical reasoning in first order logic. Of course, P is not a NEF term, so it is somewhat restricted in where it can be used; for dependent application, it can only be used as the operator (i.e. the leftmost term in the application).

What is apparent, is that P is a much more complicated term than Q, even though their canonical proofs in natural deduction are similar in length[1];

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B(a) \vdash \neg B(a)}}{B(a) \vdash \exists x.\neg B(x)} \quad \dfrac{}{\neg\exists x.\neg B(x) \vdash \neg\exists x.\neg B(x)}}{\neg B(a), \neg\exists x.\neg B(x) \vdash \bot}}{\neg\exists x.\neg B(x) \vdash B(a)}}{\neg\exists x.\neg B(x) \vdash \forall x.B(x)}}{\neg\forall x.B(x) \vdash \neg\forall x.B(x)}}{\neg\forall x.B(x), \neg\exists x.\neg B(x) \vdash \bot}}{\neg\forall x.B(x) \vdash \exists x.\neg B(x)}$$

[1] These proofs are thanks to [16]

$$\dfrac{\dfrac{\exists x.\neg B(x) \vdash \exists x.\neg B(x)}{\exists x.\neg B(x) \vdash B(a)} \qquad \dfrac{\dfrac{\dfrac{\forall x.B(x) \vdash \forall x.B(x)}{\forall x.B(x) \vdash B(a)} \qquad \neg B(a) \vdash \neg B(a)}{\forall x.B(x), \neg B(a) \vdash \bot}}{\forall x.B(x) \vdash B(a)}}{\dfrac{\exists x.\neg B(x), \forall x.B(x) \vdash \bot}{\exists x.\neg B(x) \vdash \neg \forall x.B(x)}}$$

This stems from the limitation of the $\lambda\mu$-calculus discussed at the end of Chapter 4; we are only able to use (RAA) on the 'special' assumptions. What is certainly worth exploring, is if we can combine the $\nu\lambda\mu$-calculus [70] with dependent types in a similar manner to how we did with $\lambda\mu$. As a rough example, this would let us produce the term:

$$\lambda x \mu y.[y](\lambda z.\mu a.[(z, a)]x) : \neg(w : A) \to \neg B \to (w : A) \times \neg B$$

The derivation of this can be found in B.4. A presentation of the $\nu\lambda\mu$ calculus can be found in [70, p95].

9.2.2 Implementation

We successfully implemented a language based on $\mathbf{ECC}_{\lambda\mu}$, allowing for dependently typed functions with control operators to be defined. Looking back to the initial goals of this project, theorem proving in classical logic, we are very happy that we were able to implement a proof assisstant for first order classical logic. As well, we are satisfied that the implementation avoids the issue of degeneracy of the domain of discourse outlined in Chapter 5, as the function:

```
f : (A : Univ) -> A -> A -> A
f _ x y = proj1(\a:a\(x,\d:a\(y,refl)))
```

does not pass the type checking process, as the subterm `\a :a\ (x,\d :a\ (y,refl))` is correctly detected to be non-reducible to NEF.

There are, however, some important parts of our implementation that are missing due to time limitations. The type checker for data and record declarations is missing a check for strict positivity in the constructors/derivations, although we suspect this shoudln't be too difficult a check to implement. More difficult issues are the recursion checks and the universe hierarchy; currently, we allow general recursion, and the universe hierarchy is collapsed to $\mathcal{U} : \mathcal{U}$, which we know to be inconsistent. We discuss how we might rectify these problems in Chapter 11.

Our subsystem for evaluating terms to NEF is simple; it just attempts to normalise them. This of course could lead to a non-terminating compilation process, for example, if it attempts to check if the term $(\lambda x.xx)(\lambda y.yy)$ evaluates to an NEF term. Our solution to this was to first type check assuming every term is NEF and, if the term passes this check, we then type it again, but checking if terms evaluate to NEF (when they need to). This method could be seen as a bit of a 'hack', and it essentially doubles the computational cost of type checking. It also relies on the property that the naive version of $\mathbf{ECC}_{\lambda\mu}$ (i.e. without the NEF checks) is normalising, which remains unproven. To have a terminating type checker, we either need a proof of normalisation, or use another method for evaluating to NEF entirely.

10 | Ethical Discussion

As this project is based in a very theoretical area of computer science, there aren't too many 'real world' problems to have to worry about. Our project made no use of human particpants or personal data, nor does it (immediately) involve developing countries or have environmental consequences.

The main ethical issue to consider about this project, is that of perhaps any theoretical project; that the work we have done is correct – especially as our work is closely tied to program verification. As well as growing use in mathematics, theorem provers see use in industry for verification of both software and hardware. Any work based on our own here would be tied to any mistakes we might not have noticed. Thus, we have taken care to explain exactly the properties we have proven for sure, and those for which we have only been able to claim. Although, as our implementations are more as a proof of concept, we don't expect them to see any serious use industry, at least not before they are greatly expanded upon.

11 | Conclusion

Following, we will summarise our work in Section 11.1. In Section 11.2 and highlight areas that could extend and improve upon the theoretical and practical work.

11.1 Review

Our aim was to explore the correspondence of classical logic and computation, and to see if a theorem prover could be based on this correspondence. This project was undertaken in an open-ended manner, where we had no specific goal to complete. Instead, our approach was to work back and forth between study and implementation; both avenues of work ended up complementing each other nicely.

We explained the differences between intuitionistic and classical logic, and gave an account of their corresponding calculi. We also gave an account of the $\lambda\mu$-calculus as both a syntax for describing proofs in classical logic, and as a computational calculus with control operators, with a particular focus on evaluation contexts and how they are manipulated by the μ operator. We feel our review of dependent types and their interaction with control serves as a good introduction to the research that has been done in trying to combine the two.

We defined and implemented a principal pairing algorithm for $\lambda\mu$ with sums, products and name polymorphism. The algorithm enjoyed soundness and completeness, meaning it is able to serve as a checker for classical propositional proofs and we can be certain about its correctness. This led to our development of a proof assisstant for classical propositional logic.

Our exploration into dependent types led to our development of $\mathbf{ECC}_{\lambda\mu}$, which expands upon previous work by safely introducing coproduct, inductive data and record types to the calculus. We also presented and implemented a bidirectional type checking algorithm, forming the core of a dependently typed functional programming language with control – which serves as a proof assisstant for classical logic. Due to its need by the subtyping algorithm, the programs written in this language are 'runnable', with a call-by-value evaluation strategy. During the development of program evaluation, we created a Haskell type class that implements structural substitution with an automatic definition for generic types, much in the style of `unbound`'s `Subst` type class.

11.2 Future Work

11.2.1 Simple Types

Non-logical Extension Although the propositional proof assisstant is fully capable of reasoning about any classical proposition, it is somewhat limited as a programming language. It might be worth considering abandoning the logical soundness, and instead developing a strongly typed, Haskell-like language with control operators. This could be compared with languages like Typed Scheme (or Typed Racket, a Scheme dialect), which have capabilities for control with the `callcc` and `abort` operators, although the μ operator allows for perhaps a finer control over the context. These languages however were made by retroactively adding a strong type system to Lisp dialects, which are usually dynamically typed. Haskell also features a version of `callcc`, but is implemented through a monad, meaning a computation involving it cannot be 'pure'. A language based on $\lambda\mu$ would be able to have context control within 'pure' functions.

In fact, our principal pairing algorithm cases for dealing with μ and $[\cdot]$ terms can be transformed and then added to Milner's Algorithm $\mathcal{W}$ [50]:

$$
\begin{array}{lcll}
\mathcal{W}\ \Gamma\ \Delta\ \mu\alpha.M & = & \langle S, S\varphi\rangle & \\
 & & \text{where} & \langle S, \bot\rangle = \mathcal{W}\ \Gamma\ (\alpha : \varphi, \Delta)\ M \\
 & & & \varphi \quad \text{is fresh} \\
 & & & \\
\mathcal{W}\ \Gamma\ \Delta\ [\alpha].M & = & \langle S_2 \circ S_1, \bot\rangle & \\
 & & \text{where} & \langle S_1, A\rangle = \mathcal{W}\ \Gamma\ \Delta\ M \\
 & & & \alpha : B \in \Delta \\
 & & & S_2 = \mathit{unify}\ A\ (S_1 B)
\end{array}
$$

11.2.2 Dependent Types

$\nu\lambda\mu$-Calculus In Chapter 9, we briefly discussed how a conjectured $\nu\lambda\mu$-calculus with dependent types could lead to a more succinct proof of $\neg\forall w.B \to \exists w.\neg B$. Although the calculus has issues with its reductions, we do think it would be very interesting to explore if this calculus can be safely equipped with dependent types, as it would give a language that is much more intuitive from a logical perspective.

Equality Although equality highlighted the problems caused by mixing control and dependent types, we gave it a minimal treatment in this report. We expect the usual equality rules from Martin Löf type theory (like those in [75, A.2]) can be safely added to the theory, with maybe some care to when terms are NEF. An example of such a rule would be,

$$
\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)a = b[a/x] : B[a/x]}\ (\Pi =)
$$

which expresses how to reason about equality of a function application. As this involves an application in the left hand side of the equality, it is likely that we would need $a \in \text{NEF}$. Similar rules can be found for the other types we presented.

Homotopy Type Theory (HoTT) One of the main areas of research into equality in type theory is HoTT [75], which sees types as a homotopical space, and elements of types as objects in those spaces. Equality between two elements of a type is identified by a path between them; they can be seen as being different points on the same 'object'. The notion of equality is further extended by the axiom of *Univalence*, that states an isomorphism between two types means those two types are equal, where an isomorphism is a bijective function between the two types that preserves equalities. This axiom, however, is incompatible with classical logic; in particular, there are types for which $(\neg\neg E)$ and ((LEM)) are false [75, p110]. It is worth exploring if this is still the case in the classical dependent calculi due to the NEF restrictions of dependent elimination; explicitly, in a classical calculus (like $\mathbf{ECC}_{\lambda\mu}$), are we able to soundly add univalence?

Weak Existential The dependent pair types in $\mathbf{ECC}_{\lambda\mu}$ are also known as *strong sigma types* [45, p41]. They are characterised by the fact that we can ask for both the witness and the proof of the type. Of course, we now know this isn't completely compatible with the control operators, and a lot of the work we've covered have been efforts towards getting them to play together nicely. There are so-called 'weak' existential types, that logically relate to $\exists$, but don't allow projections. Such types usually have elimination rules with a motive. An example of such a rule is given by [71],

$$
\frac{\Gamma \vdash m : \exists(x : A).B \qquad \Gamma, x : A, y : B \vdash n : C \qquad \Gamma \vdash C : \mathcal{U}}{\Gamma \vdash \mathbf{E}_{x:A,y:B}(m, n) : C}
$$

In [45, p43], Luo explains that when adding a weak existential quantifier type to a type system, if the system also has strong Σ-types then we are able to define projections for the weak existential, making the two isomorphic (meaning any weak existential must also be strong). It is worth exploring if this property still holds in $\mathbf{ECC}_{\lambda\mu}$, where there are cases when we aren't able to project out of the strong Σ-types, and if we can characterise the strong Σ-types by only those we are able to apply projections to.

Implementation

Given the limited time frame of this project, there are many ways in which the language can be improved upon. We will briefly discuss some of the features found in other theorem provers that are missing from ours. Some of these are necessary in order to ensure consistency of the language, whereas the rest don't affect the underlying logic of the type system, but do make the user level syntax much nicer to work with.

Universe Hierarchy To simplify the implementation, we currently have a single universe $\mathcal{U}$, with a typing rule $\mathcal{U} : \mathcal{U}$. As discussed in Chapter 5, this makes the system inconsistent. A simple way to implement the univer hierarchy given by $\mathcal{U}_i : \mathcal{U}_{i+1}$, would be to have the user explicitly state the level of each universe. The issue with this method is that the user needs to know in advance the highest level that they expect a function will need. Using the example from [47], we could define our own version of logical disjunction by $\texttt{Or} : \mathcal{U}_{100} \to \mathcal{U}_{100} \to \mathcal{U}_{100}$, and hope that $\mathcal{U}_{100}$ is large enough to contain all the types that will be used with `Or`. A far better solution is to have a *typical ambiguity* [37], where the user needs not annotate universe levels, and instead the compiler ensures that the hierarchy is maintained. In a sense, the definitions $\texttt{A} : \mathcal{U}$ become polymorphic in the universe level, allowing them to be used in arbitrarily large levels.

Safe Recursion We currently allow generally recursive functions, which means it is possible to define non-terminating functions. This of course is another source of inconsistency, as we are able to define a proof of $\bot$,

```
foo : Bot
foo = foo
```

Although undecideable in general [73], we can employ conservative heuristics to avoid non-terminating functions. Agda employs two methods of termination checking; Primitive and Structural recursion. In primitive recursion, the arguments supplied to the recursive call must be exactly one constructor smaller than those given. For example, `foo (suc n) = foo n` will pass this check, but `foo n = foo n` will not. Structural recursion generalises this notion, and allows recursive calls with subexpressions of the given arguments, where at least one argument is a strict subexpression. A subexpression of a term m is a term that is 0 or more constructors smallers; a subexpression is strictly smaller when it is 1 or more constructors smaller.

Implicit Arguments Currently, all arguments to our functions are *explicit*. This means that each argument needs to be explicitly passed to a function, making many functions rather annoying to handle. Implicit arguments allow the user to omit terms when calling a function, and letting the type checker infer what term should be used. For example, the function `id : (A : Univ) -> A -> A` has an explicit argument `A`, but this can always be inferred as the type of the second argument. This function could be rewritten `id : {A : Set} -> A -> A`, which makes the first argument implicit, so we can apply `id` without supplying it the first argument; for example, for a typed term `m : B`, we need only write `id m`, the type `B` is inferred from `m`.

(Co)Pattern Matching Defining functions by `case` and `build` expressions is rather cumbersome for the user. Agda, and indeed most functional programming languages in general, employs *pattern matching* and *copattern matching* for defining functions on data and codata, respectively. (Co)Pattern matching makes code far more readable, and is much more intuitive to code with. Pattern matching with dependent types is a little more complex than with simple types, as the variables bound by the pattern can also cause the types of the subsequent terms to have the corresponding variable bound by this pattern.

A | $\lambda\mu^N$

A.1 Proofs

Completeness for $pp_{\lambda\mu}$

Proof. By induction on the structure of M. This proof is strongly guided by the proof of completeness of pp for Curry types in [8, p17].

Base Case x

Assume that $\Gamma \vdash x : B \mid \Delta$.

As x is a variable, the only typing rule that can apply is (Ax), so we must have $(x : B) \in \Gamma$.

Now, the algorithm succeeds, by definition, with $pp\ x = \langle \{x : \varphi\}, \varnothing, \varphi \rangle$, and we show there is a substitution S that satisfies the requirements; in particular, choose $S = \varphi \mapsto B$. Then,

$$\begin{aligned} S\{x : \varphi\} = \{x : B\} &\subseteq \Gamma \\ S\,\varnothing = \varnothing &\subseteq \Delta \\ S\varphi &= B \end{aligned}$$

Inductive Case $\lambda x.M$

Assume that $\Gamma \vdash \lambda x.M : B \mid \Delta$.

By $(\to I)$, we must have types C and D such that $B = C \to D$ and,

$$\Gamma, x : C \vdash M : D \mid \Delta$$

Thus, by induction: $pp\ M$ succeeds, $pp\ M = \langle \Pi, \Sigma, A \rangle$, and there is a substitution S such that,

$$\begin{aligned} S\,\Pi &\subseteq \Gamma, x : C \\ S\,\Sigma &\subseteq \Delta \\ SA &= D \end{aligned}$$

Then, by definition, $pp\ \lambda x.M$ will succeed.

- **Case:** $x : P \in \Pi$

  $pp\ \lambda x.M = \langle \Pi \setminus x : P, \Sigma, P \to A \rangle$. As we have that $S\,\Pi \subseteq \Gamma, x : C$, it follows that both,

  $$S(\Pi \setminus x : P) \subseteq \Gamma \qquad \text{as } x \notin \Gamma, \text{ by } (\to I)$$

  and,

  $$S\,P = C \qquad \text{as } x : P \in \Pi \implies x : C \in S\,\Pi$$

  We already have that $S\,\Sigma \subseteq \Delta$.

  As $SA = D$ and $SP = C$, it follows that $S(P \to A) = C \to D = B$. Thus we have found a substitution satisfying the theorem.

- **Case:** $x \notin \Pi$

  $pp\ \lambda x.M = \langle \Pi, \Sigma, \varphi \to A \rangle$.

  As $x \notin \Pi$ and $S\,\Pi \subseteq \Gamma, x : C$, we can deduce that $S\,\Pi \subseteq \Gamma$. We set $S' = S \circ (\varphi \mapsto C)$. As φ is fresh, it doesn't appear in Π or Σ, which implies that,

  $$S'\Pi = S\,\Pi \subseteq \Gamma \quad \text{and} \quad S'\Sigma = S\,\Sigma \subseteq \Delta$$

  Finally, we know $S'\varphi = C$ and $S'A = D$, so then $S'(\varphi \to A) = C \to D = B$. So S' is a satisfactory substitution.

Inductive Case MN

Assume that $\Gamma \vdash MN : B \mid \Delta$.

By rule $(\to E)$, there is a type C such that,

$$\Gamma \vdash M : C \to B \mid \Delta \quad \text{and} \quad \Gamma \vdash N : C \mid \Delta$$

Then, by induction, $pp\ M$ and $pp\ N$ both succeed, and $pp\ M = \langle \Pi_1, \Sigma_1, P_1 \rangle$ and $pp\ B = \langle \Pi_2, \Sigma_2, P_2 \rangle$; and we have substitutions S_1 and S_2 such that:

$$\begin{aligned} S_1\,\Pi_1 &\subseteq \Gamma & S_2\,\Pi_2 &\subseteq \Gamma \\ S_1\,\Sigma_1 &\subseteq \Delta & S_2\,\Sigma_2 &\subseteq \Delta \\ S_1\,P_1 &= C \to D & S_2\,P_2 &= C \end{aligned}$$

With φ fresh, we now need substitutions satisfying,

$$\begin{aligned} S_u &= \mathit{unify}\ P_1\ (P_2 \to \varphi) \\ S_\Gamma &= \mathit{unifyCtxt}\ (S_u \Pi_1)\ (S_u \Pi_2) \\ S_\Delta &= \mathit{unifyConcs}\ (S_\Gamma \circ S_u \Sigma_1)\ (S_\Gamma \circ S_u \Sigma_2) \end{aligned}$$

By the definition of pp, the type variables in Π_1 and Σ_1 are separate from those in Π_2 and Σ_2. This implies that S_1 and S_2 will act on distinct sets of type variables. We argue that we need only check S_u exists, as this implies that S_Γ and S_Δ exist.

If we let $S'_u = S_2 \circ S_1 \circ (\varphi \mapsto D)$, we can see that,

$$S'_u P_1 = C \to D \quad \text{and} \quad S'_u(P_2 \to \varphi) = C \to D$$

By 3.10, there exists some S''_u such that $S'_u = S''_u \circ S_u$. Thus, S_u exists and the algorithm succeeds.

We now show there is some S such that $S(S_\Delta \circ S_\Gamma \circ S_u(\Pi_1 \cup \Pi_2)) \subseteq \Gamma$

$$S(S_\Delta \circ S_\Gamma \circ S_u(\Pi_1 \cup \Pi_2)) \subseteq \Gamma \quad \text{and} \quad S(S_\Delta \circ S_\Gamma \circ S_u(\Sigma_1 \cup \Sigma_2)) \subseteq \Delta$$

Well, taking $S_3 = S_2 \circ S_1 \circ (\varphi \mapsto D)$, we note again S_2 and S_1 don't act on the same types, and that the types in Π_1 and Σ_1 are separate from those in Π_2 and Σ_2. This gives us,

$$\begin{aligned} S_3\,\Pi_1 &\subseteq \Gamma & S_3\,\Pi_2 &\subseteq \Gamma \\ S_3\,\Sigma_1 &\subseteq \Delta & S_3\,\Sigma_2 &\subseteq \Delta \end{aligned}$$

As we know from S'_u that S_3 will unify P_1 and $P_2 \to \varphi$, so, by 3.10, there is some S_4 such that $S_3 = S_4 \circ S_u$, which implies

$$\begin{aligned} S_4 \circ S_u\,\Pi_1 &\subseteq \Gamma & S_4 \circ S_u\,\Pi_2 &\subseteq \Gamma \\ S_4 \circ S_u\,\Sigma_1 &\subseteq \Delta & S_4 \circ S_u\,\Sigma_2 &\subseteq \Delta \end{aligned}$$

As both $S_4 \circ S_u \Pi_1$ and $S_4 \circ S_u \Pi_2$ are subsets of gamma, this means any shared variables will have the same type. This means S_4 must unify Π_1 and Π_2. So, by 3.10, there is some S_5 such that $S_4 = S_5 \circ S_\Gamma$, which gives us,

$$\begin{aligned} S_5 \circ S_\Gamma \circ S_u\,(\Pi_1 \cup \Pi_2) &\subseteq \Gamma \\ S_5 \circ S_\Gamma \circ S_u\,\Sigma_1 \subseteq \Delta \qquad & S_5 \circ S_\Gamma \circ S_u\,\Sigma_2 \subseteq \Delta \end{aligned}$$

Similarly, as both $S_5 \circ S_\Gamma \circ S_u \Sigma_1$ and $S_5 \circ S_\Gamma \circ S_u \Sigma_2$ are subsets of Δ, then any shared variables must have the same types. This implies S_5 must unify $S_5 \circ S_\Gamma \circ S_u \Sigma_1$ and $S_5 \circ S_\Gamma \circ S_u \Sigma_2$. Again, using 3.10, there is some S_6 such that $S_5 = S_6 \circ S_\Delta$. This (at last) gives us,

$$\begin{aligned} S_6 \circ (S_\Delta \circ S_\Gamma \circ S_u\,(\Pi_1 \cup \Pi_2)) &\subseteq \Gamma \\ S_6 \circ (S_\Delta \circ S_\Gamma \circ S_u\,(\Sigma_1 \cup \Sigma_2)) &\subseteq \Delta \end{aligned}$$

And we still have $S_6 \circ S_\Delta \circ S_\Gamma \circ S_u \varphi = B$, as this will contain the substitution $S_2 \circ S_1 \circ (\varphi \mapsto D)$. So we choose S_6 to be the substitution, as it satisfies the required properties.

Inductive Case $\mu\alpha.M$

Assume $\Gamma \vdash \mu\alpha.M : A \mid \Delta$.

By (μ), we know $\Gamma \vdash M : \bot \mid \alpha : A,\ \Delta$

By induction, $pp\ M$ succeeds, and;

$$\begin{aligned} pp\ M &= \langle \Pi, \Sigma.P \rangle \\ \exists S \text{ such that} \quad S\,\Pi &\subseteq \Gamma \\ S\,\Sigma &\subseteq \Delta \\ S\,P &= \bot \end{aligned}$$

As we know $S'P = \bot$, we know P can be unified with $\bot$; so the algorithm succeeds.

- **Case** $\alpha : B \in \Sigma$

  Then, by definition, $pp\ \mu\alpha.M = \langle \Pi, \Sigma \setminus (\alpha : B), B \rangle$.

  As we know $S'\Sigma \subseteq \Delta, \alpha : A$

  $$\begin{aligned} \implies \qquad\qquad S'B &= A, \\ \text{and} \quad S'\Sigma \setminus (\alpha : B) &\subseteq \Delta \end{aligned}$$

  We already have that $S'\Pi \subseteq \Gamma$, so we choose $S = S'$.

- **Case** $\alpha \notin \Sigma$

  Then, by definition, $pp\ \mu\alpha.M = \langle \Pi, \Sigma, \varphi \rangle$.

  Noting that φ doesn't occur in Π or Δ, we choose $S = S' \circ (\varphi \mapsto A)$.

  This gives us;

  $$\begin{aligned} S\,\Pi &\subseteq \Gamma \\ S\,\Sigma &\subseteq \Delta \\ S\,\varphi &= A \end{aligned}$$

Inductive Case $[\alpha]M$

Assume that $\Gamma \vdash [\alpha]M : \bot \mid \alpha : A,\ \Delta$.

By $(name)$: $\Gamma \vdash M : A \mid \Delta$.

By induction, $pp\ M$ succeeds, and $pp\ M = \langle \Pi, \Sigma, B \rangle$ and $\exists S$ such that:

$$\begin{aligned} S\,\Pi &\subseteq \Gamma \\ S\,\Sigma &\subseteq \Delta \\ S\,B &= A \end{aligned}$$

- **Case** $\alpha : C \in \Sigma$

  As we know $S\,\Sigma \subseteq \Delta$, it follows that $(\alpha : S\,C) \in \Delta$

  Now, as we have assumed $[\alpha]M$ is well formed and typed, the types in $\alpha : A$ and $\alpha : S\,C \in \Delta$ must be consistent; thus $S\,C = A$. By 3.10, unification of C and A succeeds, and we have, for some S', $S = S' \circ S_U$, where $S_U = \mathit{unify}\ A\ C$.

  Thus $pp\ [\alpha]M$ succeeds, and $pp\ [\alpha]M = S_U\langle \Pi, \Sigma, \bot \rangle$. As we know $S\,\Pi \subseteq \Gamma$ and $S\,\Sigma \subseteq \Delta$, and $S = S' \circ S_U$, we have

  $$\begin{aligned} S'(S_U\Pi) &\subseteq \Gamma \\ S'(S_U\Sigma) &\subseteq \Delta \\ S'\bot &= \bot \end{aligned}$$

  So S' is a satisfactory substitution.

- **Case** $\alpha \notin \Sigma$

  Then, by definition, $pp\ [\alpha]M$ succeeds, and $pp\ [\alpha]M = \langle \Pi, (\Sigma, \alpha : B), \bot \rangle$.

  We already know that S satisfies $S\,\Sigma \subseteq \Delta$ and $SB = A$, so it follows that;

  $$S\,(\Sigma, \alpha : B) \subseteq \Delta, \alpha : A$$

  Thus we choose S as the substitution, and we are done.

□

Soundness for $pp_{\lambda\mu}$

Proof. By induction on the structure of M.

Base Case x

By definition, $pp\ x = \langle \{x : \varphi\}, \varnothing, \varphi \rangle$.

We immediately have; $x : \varphi \vdash x : \varphi \mid \varnothing$, from rule *(Ax)*.

Inductive Case $\lambda x.M$

Inductive Hypothesis:

$$pp\ M = \langle \Gamma, \Delta, B \rangle \implies \Gamma \vdash M : B \mid \Delta$$

- **Case:** $x \in \Gamma$

  Now; we have that $pp\ \lambda x.M = \langle \Gamma \setminus (x : A), \Delta, A \to B \rangle$

  As we know that $\Gamma \vdash M : B \mid \Delta$, and that $x : A \in \Gamma$, then let Γ' be such that $\Gamma = \Gamma', x : A$. Then we have:

  $$\begin{aligned} & \Gamma', x : A \vdash M : B \mid \Delta \\ \implies & \Gamma' \vdash \lambda x.M : A \to B \mid \Delta && \text{by } (\to I) \\ \implies & \Gamma \setminus (x : A) \vdash \lambda x.M : A \to B \mid \Delta && \text{by defn of } \Gamma \end{aligned}$$

- **Case** $x \notin \Gamma$

  Then $pp\ \lambda x.M = \langle \Gamma, \Delta, \varphi \to B \rangle$

  Well, as we know $\Gamma \vdash M : B \mid \Delta$, and that $x \notin fv(M)$, we can extend Γ with $x : \varphi$,

  $$\begin{aligned} & \Gamma, x : \varphi \vdash M : B \mid \Delta \\ \implies & \Gamma \vdash \lambda x.M : \varphi \to B \mid \Delta && \text{by } (\to I) \end{aligned}$$

Inductive Case MN

Inductive Hypotheses:

$$\begin{aligned} pp\ M = \langle \Gamma_1, \Delta_1, P_1 \rangle &\implies \Gamma_1 \vdash M : P_1 \mid \Delta_1 \\ pp\ N = \langle \Gamma_2, \Delta_2, P_2 \rangle &\implies \Gamma_2 \vdash N : P_2 \mid \Delta_2 \end{aligned}$$

We need only consider the case that $pp\ MN$ will succeed. So we know

$$pp\ MN = S_\Delta \circ S_\Gamma \circ S_U \langle \Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2, \varphi \rangle$$

We write $S := S_\Delta \circ S_\Gamma \circ S_U$. As S_U unifies P_1 and $P_2 \to \varphi$, we can write $SP_1 = S(P_2 \to \varphi) = A \to B$, for some A and B.

We can apply S to our inductive hypotheses;

$$\begin{aligned} S\Gamma_1 \vdash M : SP_1 \mid S\Delta_1 &\implies S\Gamma_1 \vdash M : A \to B \mid S\Delta_1 \\ S\Gamma_2 \vdash N : SP_2 \mid S\Delta_2 &\implies S\Gamma_2 \vdash N : A \mid S\Delta_2 \end{aligned}$$

As we know Γ_1 can be unified with Γ_2 (and Δ_1 with Δ_2), we can apply weakening to both derivations. Writing $\Gamma = S(\Gamma_1 \cup \Gamma_2)$ and $\Delta = S(\Delta_1 \cup \Delta_2)$, we apply weakening,

$$\Gamma \vdash M : A \to B \mid \Delta \qquad\qquad \Gamma \vdash N : A \mid \Delta$$

Then we apply rule $(\to E)$,

$$\Gamma \vdash MN : B \mid \Delta$$

And we note that $S\varphi = B$, so S satisfies the needed properties.

Inductive Case $\mu\alpha.M$

Inductive Hypothesis:

$$pp\ M = \langle \Gamma, \Delta, P \rangle \implies \Gamma \vdash M : P \mid \Delta$$

As we need only consider when the algorithm succeeds, we know $P = \bot$, so

$$\Gamma \vdash M : \bot \mid \Delta$$

- **Case** $\alpha \in \Delta$

  Then we can write $\Delta = \Delta', \alpha : A$ and $pp\ \mu\alpha.M = \langle \Gamma, \Delta', A \rangle$ And we can now derive,

$$\begin{aligned} & \Gamma \vdash M : \bot \mid \Delta', \alpha : A \\ \implies & \Gamma \vdash \mu\alpha.M : A \mid \Delta' && \text{by } (\mu) \end{aligned}$$

- **Case** $\alpha \notin \Delta$

  Then $pp\ \mu\alpha.M = \langle \Gamma, \Delta, \varphi \rangle$. We note that, as $\alpha \notin fn(M)$, we can weaken Δ,

$$\begin{aligned} & \Gamma \vdash M : \bot \mid \Delta \\ \implies & \Gamma \vdash M : \bot \mid \Delta, \alpha : \varphi && \text{by weakening} \\ \implies & \Gamma \vdash \mu\alpha.M : \varphi \mid \Delta && \text{by rule } (\mu) \end{aligned}$$

Inductive Case $[\alpha]M$

Inductive Hypothesis:

$$pp\ M = \langle \Gamma, \Delta, A \rangle \implies \Gamma \vdash M : A \mid \Delta$$

- **Case** $\alpha : B \in \Delta$

  Then $pp\ [\alpha]M = S\langle \Gamma, \Delta, \bot \rangle$, where S unifies A and B. We write $P := SA = SB$. As $\alpha \in \Delta$, we can write $\Delta = \Delta', \alpha : B$.

  Applying S to the inductive hypothesis gives us,

$$\begin{aligned} & S\Gamma \vdash M : SA \mid S\Delta', \alpha : SB \\ \implies & S\Gamma \vdash M : P \mid S\Delta', \alpha : P \end{aligned}$$

  And now use of the (*name*) rule is permissable as α and M have the same types;

$$\begin{aligned} & S\Gamma \vdash M : P \mid S\Delta', \alpha : P \\ \implies & S\Gamma \vdash [\alpha]M : \bot \mid S\Delta', \alpha : P && \text{by } (name) \end{aligned}$$

  Finally, we note that $S\Delta = S(\Delta', \alpha : B) = S\,\Delta', \alpha : P$, so we can rewrite the conclusion of the derivation above;

$$S\Gamma \vdash [\alpha]M : \bot \mid S\,\Delta$$

- **Case** $\alpha \notin \Delta$

  Then $pp\ [\alpha]M = \langle \Gamma, (\Delta, \alpha : A), \bot \rangle$

  And we apply the (*name*) rule to the inductive hypothesis,

$$\begin{aligned} & \Gamma \vdash M : A \mid \Delta \\ \implies & \Gamma \vdash [\alpha]M : \bot \mid \Delta, \alpha : A && \text{by } (name) \end{aligned}$$

□

Completeness for $pp_{\lambda\mu_N}$

Proof. It is easy to see that, by A.1, we already have the cases for $x, \lambda x.M, MN, \mu\alpha.M, [\alpha]M$ done, as the environment $\mathcal{E}$ is invariant in this cases, and is only passed as an extra parameter.

Base Case n

Assume that $\mathcal{E};\Gamma \vdash \mathrm{n} : B \mid \Delta$.

As n is a function name, the only rule we can apply is (n), so we must have $\mathrm{n} \in \mathcal{E}$.

By definition, the algorithm succeeds with $pp\ \mathrm{n} = \langle \varnothing, \varnothing, \mathit{FreshInstance}(\mathcal{E}\mathrm{n}) \rangle$. By definition of *FreshInstance*, we can find a substitution $S = \mathit{FreshInstance}(\mathcal{E}\mathrm{n}) \mapsto B$. Then,

$$\begin{aligned} S\{\mathrm{n} : \mathit{FreshInstance}(\mathcal{E}\mathrm{n})\} = \{\mathrm{n} : B\} &\subseteq \mathcal{E} \\ S\varnothing = \varnothing &\subseteq \Gamma \\ S\varnothing = \varnothing &\subseteq \Delta \\ S\varphi &= B \end{aligned}$$

Case $\mathrm{in}_i(M)$

Assume that $\mathcal{E};\Gamma \vdash \mathrm{in}_i(M) : B \mid \Delta$.

Then, by $(+I_i)$, we know B is a sum type, so there are B_1, B_2 such that $B = B_1 + B_2$.

- **Case:** $i = 1$
  By induction, pp succeeds and $pp\ \mathcal{E}\ M = \langle \Pi, \Sigma, A \rangle$, and there is a substitution S such that $S\Pi \subseteq \Gamma$, $S\Sigma \subseteq \Delta$, $SA = B_1$.
  Then, $pp\ \mathcal{E}\ \mathrm{in}_1(M) = \langle \Pi, \Sigma, A + \varphi \rangle$. So we choose $S' = S \circ (\varphi \mapsto B_2)$. Then we have $S'\Pi \subseteq \Gamma$, $S'\Sigma \subseteq \Delta$, $S'(A + \varphi) = B_1 + B_2 = B$.
- **Case:** $i = 2$
  Follows similarly.

Case $\mathrm{case}(M, N, L)$

Assume that $\mathcal{E};\Gamma \vdash \mathrm{case}(M, N, L) : B \mid \Delta$.

Then, by the rule $(+E)$, there are types C, D such that:

$$\mathcal{E};\Gamma \vdash M : C + D \mid \Delta \qquad \mathcal{E};\Gamma \vdash N : C \to B \mid \Delta \qquad \mathcal{E};\Gamma \vdash M : D \to B \mid \Delta$$

By induction, pp succeeds on M, N, L with:

$$pp\ \mathcal{E}\ M = \langle \Pi_1, \Sigma_1, P_1 \rangle \qquad pp\ \mathcal{E}\ N \qquad = \langle \Pi_2, \Sigma_2, P_2 \rangle \qquad pp\ \mathcal{E}\ L = \langle \Pi_3, \Sigma_3, P_3 \rangle$$

And we have (mutually exclusive) substitutions S_1, S_2, S_3 such that

$$\begin{array}{lll} S_1\Pi_1 \subseteq \Gamma & S_2\Pi_2 \subseteq \Gamma & S_3\Pi_3 \subseteq \Gamma \\ S_1\Sigma_1 \subseteq \Gamma & S_2\Sigma_2 \subseteq \Gamma & S_3\Sigma_3 \subseteq \Gamma \\ S_1P_1 = C + D & S_2P_2 = C \to B & S_3P_3 = D \to B \end{array}$$

Taking $\varphi, \varphi_1, \varphi_2$ fresh, we need substitutions satisfying:

$$\begin{aligned} S_M &= \mathit{unify}\ P_1\ (\varphi_1 + \varphi_2) \\ S_N &= \mathit{unify}\ (S_M P_2)\ (S_M(\varphi_1 \to \varphi)) \\ S_L &= \mathit{unify}\ (S_N \circ S_M P_3)\ (S_N \circ S_M(\varphi_2 \to \varphi)) \\ S_u &= S_L \circ S_N \circ S_M \\ S_\Gamma &= \mathit{unifyCtxts}\ (S_u \Pi_1)\ (S_u \Pi_2)\ (S_u \Pi_3) \\ S_\Delta &= \mathit{unifyCtxts}\ (S_\Gamma \circ S_u \Sigma_1)\ (S_\Gamma \circ S_u \Sigma_2)\ (S_\Gamma \circ S_u \Sigma_3) \\ S' &= S_\Delta \circ S_\Gamma \circ S_u \end{aligned}$$

Similar to our argument for the case MN in A.1, the type variables in Π_1, Σ_1 are separate from those in Π_2, Σ_2 and Π_3, Σ_3 (which are also separate from each other); and we only need to prove S_M, S_N, S_L exist for S_Γ and S_Δ to exist.

- Defining $S'_M = S_3 \circ S_2 \circ S_1 \circ (\varphi_1 \mapsto C) \circ (\varphi_2 \mapsto D)$, we see that $S'_M P_1 = C + D$ and $S'_M(\varphi_1 + \varphi_2) = C + D$. Thus, by 3.10, S_M exists.
- Defining $S'_N = S_3 \circ S_2 \circ S_1 \circ S_M \circ (\varphi \mapsto B)$, we see that $S'_N P_2 = C \to B$ and $S'_N(\varphi_1 \to \varphi) = C \to B$. Thus, by 3.10, S_N exists.
- Defining $S'_L = S_3 \circ S_2 \circ S_1 \circ S_L \circ S_M$, we see that $S'_L P_3 = D \to B$ and $S'_L(\varphi_2 \to \varphi) = D \to B$. Thus, by 3.10, S_L exists.

Now we have a unification of the type variables, we know S_Γ and S_Δ exist. Using an argument similar to the MN case in A.1, we get that there is a substitution S'' (containing S_1, S_2, S_3) such that:

$$\begin{aligned} S'' \circ (S'(\Pi_1 \cup \Pi_2 \cup \Pi_3)) &\subseteq \Gamma \\ S'' \circ (S'(\Sigma_1 \cup \Sigma_2 \cup \Sigma_3)) &\subseteq \Delta \\ S''\varphi &= B \end{aligned}$$

And we note that $pp\ \mathcal{E}\ \mathrm{case}(M, N, L) = S'\langle \Pi_1 \cup \Pi_2 \cup \Pi_3, \Sigma_1 \cup \Sigma_2 \cup \Sigma_3, \varphi \rangle$, so we are done.

Case $\pi_i(M)$

Assume that $\mathcal{E}; \Gamma \vdash \pi_i(M) : B \mid \Delta$.

By $(\times E)$, there exist B_1, B_2 such that $B = B_i$ and $M : B_1 \times B_2$.

By induction, pp succeeds on M and $pp\ \mathcal{E}\ M = \langle \Pi, \Sigma, A \rangle$ and there is a substitution S such that $S\Pi \subseteq \Gamma,\ S\Sigma \subseteq \Delta,\ SA = B_1 \times B_2$.

Taking φ_1, φ_2 fresh, define $S' = S \circ (\varphi_1 \mapsto B_1) \circ (\varphi \mapsto B_2)$. Then we can see that $S'A = B_1 \times B_2$ and $S'(\varphi_1 \times \varphi_2) = B_1 \times B_2$, so S' is a unifying substitution of A and $\varphi_1 \times \varphi_2$. Thus, by 3.10, $S'' = \mathit{unify}\ A\ (\varphi_1 \times \varphi_2)$ exists.

Then, the algorithm succeeds with $pp\ \mathcal{E}\ \pi_i(M) = \langle \Gamma, \Delta, S''\varphi_i \rangle$. Taking $S_1 = S \circ S''$, we get $S_1\Pi \subseteq \Gamma,\ S_1\Sigma \subseteq \Delta,\ S_1 A = B_i$.

Case (M, N)

Assume that $\mathcal{E}; \Gamma \vdash (M, N) : B \mid \Delta$.

Then, by $(\times I)$, there are types B_1, B_2 such that $B = B_1 \times B_2$, $M : B_1$ and $N : B_2$.

By induction, pp succeeds on M and N with $pp\ \mathcal{E}\ M = \langle \Pi_1, \Sigma_1, P_1 \rangle$ and $pp\ \mathcal{E}\ N = \langle \Pi_2, \Sigma_2, P_2 \rangle$; and there are substitutions S_1, S_2 such that;

$$\begin{aligned} S_1\,\Pi_1 &\subseteq \Gamma & \qquad S_2\,\Pi_2 &\subseteq \Gamma \\ S_1\,\Sigma_1 &\subseteq \Delta & \qquad S_2\,\Sigma_2 &\subseteq \Delta \\ S_1\,P_1 &= B_1 & \qquad S_2\,P_2 &= B_2 \end{aligned}$$

We now consider the context unifications:

$$S_\Gamma = \mathit{unifyCtxts}\ \Pi_1\ \Pi_2 \qquad\qquad S_\Delta = \mathit{unifyConcs}\ (S_\Gamma\ \Sigma_1)\ (S_\Gamma\ \Sigma_2)$$

As type variables aren't shared between Π_1, Σ_1 and Π_2, Σ_2, then their unifications will succeed. Again, using an argument similar to the MN case of A.1, there is a substitution S' (containing S_1 and S_2) such that

$$\begin{aligned} S' \circ (S_\Delta \circ S_\Gamma(\Pi_1 \cup \Pi_2)) &\subseteq \Gamma \\ S' \circ (S_\Delta \circ S_\Gamma(\Sigma_1 \cup \Sigma_2)) &\subseteq \Delta \\ S'\varphi &= B \end{aligned}$$

Case $\langle \epsilon; M \rangle$

Assume that $\mathcal{E}; \Gamma \vdash \langle \epsilon; M \rangle : B \mid \Delta$. Then, by (Defs), $\mathcal{E}; \Gamma \vdash M : B \mid \Delta$.

By induction, $pp\ \mathcal{E}\ M = \langle \Pi, \Sigma, A \rangle$ and there is a substitution S such that $S\Pi \subseteq \Gamma$, $S\Sigma \subseteq \Delta$, $SA = B$.

Then, by definition, $pp\ \mathcal{E}\ \langle \epsilon; M \rangle = \langle \Pi, \Sigma, A \rangle$, and S is a satisfactory substitution.

Case $\langle (\mathrm{n} = M); \mathrm{Defs},\ N \rangle$

Assume that $\mathcal{E}; \Gamma \vdash \langle (\mathrm{n} = M); \mathrm{Defs},\ N \rangle : B \mid \Delta$.

By (Defs), there is a type C such that $\mathcal{E}; \varnothing \vdash M : C \mid \varnothing$ and $\mathcal{E}, \mathrm{n} : C \vdash \langle \mathrm{Defs},\ N \rangle : B \mid \Delta$

By induction $pp\ \mathcal{E}\ M = \langle \varnothing, \varnothing, P \rangle$ and $pp\ (\mathcal{E}, \mathrm{n} : P)\ \langle \mathrm{Defs};\ N \rangle = \langle \Pi, \Sigma, A \rangle$ and there is are substitutions S, S' such that $S\Pi \subseteq \Gamma$, $S\Sigma \subseteq \Delta$, $SA = B$, $S'P = C$. Just choose $S'' = S' \circ S$, and we are done.

□

Soundness for $pp_{\lambda\mu_N}$

Proof. It's easy to see that, by A.1, we already have the cases for $x, \lambda x.M, MN, \mu\alpha.M, [\alpha]M$ done, as the environment $\mathcal{E}$ is invariant in this cases, and is only passed as an extra parameter.

Case $(\mathrm{in}_i(M))$

By induction, $pp\ \mathcal{E}\ M = \langle \Gamma, \Delta, A \rangle \implies \mathcal{E}; \Gamma \vdash M : A \mid \Delta$

- **Case** $i = 1$
  Then we have $pp\ \mathcal{E}\ \mathrm{in}_1(M) = \langle \Gamma, \Delta, A + \varphi \rangle$. By $(+I_1)$ we know $\mathcal{E}; \Gamma \vdash M : A + \varphi \mid \Delta$.
- **Case** $i = 2$ follows a similar argument.

Case $(\mathrm{case}(M, N, L))$

By induction,

$$\begin{aligned} pp\ \mathcal{E}\ M = \langle \Gamma_1, \Delta_1, P_1 \rangle &\implies \mathcal{E}; \Gamma_1 \vdash M : P_1 \mid \Delta_1 \\ pp\ \mathcal{E}\ N = \langle \Gamma_2, \Delta_2, P_2 \rangle &\implies \mathcal{E}; \Gamma_2 \vdash N : P_2 \mid \Delta_2 \\ pp\ \mathcal{E}\ L = \langle \Gamma_3, \Delta_3, P_3 \rangle &\implies \mathcal{E}; \Gamma_3 \vdash L : P_3 \mid \Delta_3 \end{aligned}$$

Assuming pp succeeds, we get $pp\ \mathcal{E}\ \mathrm{case}(M, N, L) = S_5 \circ S_4 \circ S\langle \Gamma_1 \cup \Gamma_2 \cup \Gamma_3, \Delta_1 \cup \Delta_2 \cup \Delta_3, \varphi \rangle$. We write $S' := S_5 \circ S_4 \circ S$. As S unifies P_1 with $\varphi_1 + \varphi_2$, P_2 with $\varphi_1 \to \varphi$ and P_3 with $\varphi_2 \to \varphi$, we can write; $SP_1 = S(\varphi_1 + \varphi_2) = A + B$, $SP_2 = S(\varphi_1 \to \varphi) = A \to C$ and $SP_3 = S(\varphi_2 \to \varphi) = B \to C$ for some A, B, C. As S' unifies $\Gamma_1, \Gamma_2, \Gamma_3$, and $\Delta_1, \Delta_2, \Delta_3$, we can write $\Gamma := S'(\Gamma_1 \cup \Gamma_2 \cup \Gamma_3)$ and $\Delta := S'(\Delta_1 \cup \Delta_2 \cup \Delta_3)$. Then we can apply weakening to our hypotheses, and substiting the types with S';

$$\begin{aligned} &\mathcal{E}; \Gamma \vdash M : A + B \mid \Delta \\ &\mathcal{E}; \Gamma \vdash N : A \to C \mid \Delta \\ &\mathcal{E}; \Gamma \vdash L : B \to C \mid \Delta \end{aligned}$$

By $(+E)$, we deduce $\mathcal{E}; \Gamma \vdash \mathrm{case}(M, N, L) : C \mid \Delta$, and we note that $S'\varphi = C$.

Case $\pi_i(M)$

By induction $pp\ \mathcal{E}\ M = \langle \Gamma, \Delta, A \rangle \implies \mathcal{E}; \Gamma \vdash M : A \mid \Delta$ As S unifies A with $\varphi_1 \times \varphi_2$, we can write $SA = S(\varphi_1 \times \varphi_2) = A_1 \times A_2$ for some A_1, A_2. Then we get $pp\ \mathcal{E}\ \pi_i(M) = \langle \Gamma, \Delta, S\varphi_i \rangle = \langle \Gamma, \Delta, A_i \rangle$ and $\mathcal{E}; \Gamma \vdash M : A_1 \times A_2 \mid \Delta$.

Thus, by $(\times E_i)$, we can derive $\mathcal{E}; \Gamma \vdash \pi_i(M) : A_i \mid \Delta$.

Case (M, N)

By induction,

$$pp\ \mathcal{E}\ M = \langle \Gamma_1, \Delta_1, A \rangle \implies \mathcal{E}; \Gamma_1 \vdash M : A \mid \Delta_1$$
$$pp\ \mathcal{E}\ N = \langle \Gamma_2, \Delta_2, B \rangle \implies \mathcal{E}; \Gamma_2 \vdash N : B \mid \Delta_2$$

Then, $pp\ \mathcal{E}\ (M, N) = S_2 \circ S_1 \langle \Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2, A \times B \rangle$. Writing $S := S_2 \circ S_1$ we know there are C, D such that $S(A \times B) = C \times D$, and we can define $\Gamma = S(\Gamma_1 \cup \Gamma_2)$ and $\Delta = S(\Delta_1 \cup \Delta_2)$, so we know (by weakening and applying S):

$$\mathcal{E}; \Gamma \vdash M : C \mid \Delta \qquad\qquad \mathcal{E}; \Gamma \vdash N : D \mid \Delta$$

By $(\times I)$, we get $\mathcal{E}; \Gamma \vdash (M, N) : C \times D \mid \Delta$.

Case $\langle \epsilon; M \rangle$

By induction $pp\ \mathcal{E}\ M = \langle \Gamma, \Delta, A \rangle \implies \mathcal{E}; \Gamma \vdash M : A \mid \Delta$.

Then $pp\ \mathcal{E}\ (\epsilon; M) = \langle \Gamma, \Delta, A \rangle$, and thus we can derive:

$$\frac{\overline{\mathcal{E} \vdash \epsilon} \qquad \mathcal{E}, \Gamma \vdash M : A \mid \Delta}{\mathcal{E}; \Gamma \vdash \langle \epsilon; M \rangle : A \mid \Delta}$$

Case $\langle (\mathrm{n} = M) : \mathrm{Defs}; N \rangle$

By induction

$$pp\ \mathcal{E}\ M = \langle \varnothing, \varnothing, A \rangle \implies \mathcal{E}; \varnothing \vdash M : A \mid \varnothing$$
$$pp\ (\mathcal{E}, \mathrm{n} : A)\ \langle \mathrm{Defs}; N \rangle = \langle \Gamma, \Delta, B \rangle \implies \mathcal{E}, \mathrm{n} : A; \Gamma \vdash \langle \mathrm{Defs}; N \rangle : B \mid \Delta$$

This means that we know

$$\frac{\mathcal{E}, \mathrm{n} : A \vdash \mathrm{Defs} \qquad \mathcal{E}, \mathrm{n} : A; \Gamma \vdash N : B \mid \Delta}{\mathcal{E}, \mathrm{n} : A; \Gamma \vdash \langle \mathrm{Defs}; N \rangle : B \mid \Delta}$$

Then $pp\ \mathcal{E}\ \langle (\mathrm{n} = M) : \mathrm{Defs}; N \rangle = \langle \Gamma, \Delta, B \rangle$, and thus we can derive:

$$\frac{\dfrac{\mathcal{E}; \varnothing \vdash M : A \mid \varnothing \qquad \mathcal{E}, \mathrm{n} : A \vdash \mathrm{Defs}}{\mathcal{E}, \mathrm{n} : A \vdash (\mathrm{n} = M); \mathrm{Defs}} \qquad \mathcal{E}, \mathrm{n} : A, \Gamma \vdash N : B \mid \Delta}{\mathcal{E}, \mathrm{n} : A; \Gamma \vdash \langle (\mathrm{n} = M) : \mathrm{Defs}; N \rangle : B \mid \Delta}$$

□

A.2 Implementation

Syntax

Propositional Prover Syntax

$$
\begin{array}{rcl}
\langle name \rangle & ::= & [\text{unicode letters}] \\
\langle var \rangle & ::= & \text{`\_'} \mid \langle name \rangle \\
\langle term \rangle & ::= & \langle name \rangle \\
& \mid & \backslash(\langle var \rangle +) \rightarrow \langle term \rangle \\
& \mid & \backslash(\langle var \rangle : \langle type \rangle) \rightarrow \langle term \rangle \\
& \mid & \langle term \rangle + \\
& \mid & \backslash \langle var \rangle : \langle var \rangle \backslash \ \langle term \rangle \\
& \mid & \texttt{in}(1|2)\ \langle term \rangle \\
& \mid & \texttt{case}\ \langle term \rangle\ \texttt{of}\ (\langle term \rangle | \langle term \rangle) \\
& \mid & (\langle term \rangle, \langle term \rangle) \\
& \mid & \texttt{proj}(1|2)\ \langle term \rangle \\
& \mid & (\langle term \rangle) \\
& \mid & () \\
& \mid & ?[0-9]+ \\
\langle type \rangle & ::= & \langle name \rangle \\
& \mid & \texttt{Top} \\
& \mid & \texttt{Bot} \\
& \mid & \langle type \rangle \rightarrow \langle type \rangle \\
& \mid & \langle type \rangle + \langle type \rangle \\
& \mid & \langle type \rangle * \langle type \rangle \\
& \mid & (\langle type \rangle) \\
\langle decl \rangle & ::= & \langle name \rangle\ :\ \langle type \rangle \\
& \mid & \langle name \rangle\ =\ \langle term \rangle
\end{array}
$$

A.2.1 Natural Deduction

The following is an encoding of the usual rules for natural deduction. References for these rules can be found in [23].

```
-- Axiom
ax : A -> A
ax x = x

-- Implication
arrI : A -> B -> (A -> B)
arrI x y = (\ _ -> y)

arrE : (A -> B) -> A -> B
arrE = ax

-- Falsum
botE : Bot -> a
botE x = \a:_\ x

-- Negation
nne : ¬¬a -> a
nne y = \a:_\ y(\ x -> \_:a\ x)

negI : a -> Bot -> ¬a
negI x y = \_ -> y
```

```
negE : ¬A -> A -> Bot
negE = ax

-- Conjuction
andI : A -> B -> A * B
andI x y = (x, y)

andE1 : A * B -> A
andE1 x = proj1 x

andE2 : A * B -> B
andE2 x = proj2 x

-- Disjunction
lem : (A + ¬A)
lem = \a:a\ (in2 (\ x -> \_:a\ (in1 x)))

orI1 : A -> A + B
orI1 x = in1 x

orI2 : B -> A + B
orI2 x = in2 x

orE : (A + B) -> (A -> C) -> (B -> C) -> C
orE x y z = case x of {y | z}

-- Iff
iffI : (A -> B) -> (B -> A) -> (A <-> B)
iffI x y = (x, y)

iffE1 : (A <-> B) -> A -> B
iffE1 = andE1

iffE2 : (A <-> B) -> B -> A
iffE2 = andE2

-- Derived
pierce : ((A -> B) -> A) -> A
pierce y = \a:a\ y (\  x . \_:a\ x)

weak_pierce : ((A -> Bot) -> A) -> A
weak_pierce = pierce
```

B | $\mathbf{ECC}_{\lambda\mu}$

B.1 Proofs

NEF-Substitution Closure in Collapsed dPA$^\omega$

Proof. By induction on the structure of NEF terms. $x \notin fv(m) \implies m[n/x] = m \in \text{NEF}$. This covers terms $y, \langle\rangle, \texttt{refl}$. From now on, we assume $x \in fv(m)$.

Base Case Assume $n \in \text{NEF}$; then $x[n/x] = n \in \text{NEF}$.

Inductive Cases Assume $p[n/x], q[n/x], m[n/x] \in \text{NEF}$. Then:

- $(\lambda y.m)[n/x] = \lambda y.(m[n/x]) \in \text{NEF}$
- $\texttt{in}_i(m)[n/x] = \texttt{in}_i(m[n/x]) \in \text{NEF}$
- $\pi_i(m)[n/x] = \pi_i(m[n/x]) \in \text{NEF}$
- $(p,q)[n/x] = (p[n/x], q[n/x]) \in \text{NEF}$
- $(\texttt{case}\ m\ \texttt{of}\ (p,q))[n/x] = \texttt{case}\ m[n/x]\ \texttt{of}\ (p[n/x], q[n/x]) \in \text{NEF}$
- $(\texttt{subst}\ p\ q)[n/x] = \texttt{subst}\ p[n/x]\ q[n/x] \in \text{NEF}$
- $(\texttt{let}\ y = p\ \texttt{in}\ q)[n/x] = \texttt{let}\ y = p[n/x]\ \texttt{in}\ q[n/x]$
- $(\texttt{ind}\ t\ \texttt{of}\ (p|(y,z).q))[n/x] = \texttt{ind}\ t[n/x]\ \texttt{of}\ (p[n/x]|(y,z).q[n/x]) \in \text{NEF}$
- $(\texttt{cofix}\ t\ \texttt{of}\ (y,z).q)[n/x] = \texttt{cofix}\ t[n/x]\ \texttt{of}\ (y,z).q[n/x] \in \text{NEF}$

□

NEF-Reduction Closure in Collapsed dPA$^\omega$

Proof. By induction on the definition of reductions. We don't consider the reductions that are defined on non-NEF terms, like the standard β reduction (as application isn't NEF).

- $\texttt{let}\ x = \texttt{in}_i(p)\ \texttt{in}\ q \in \text{NEF} \implies p, q \in \text{NEF} \implies \texttt{let}\ y = p\ \texttt{in}\ q[\texttt{in}_i(y)/a] \in \text{NEF}$
- $\pi_i(\texttt{let}\ x = p\ \texttt{in}\ q) \in \text{NEF} \implies p, q \in \text{NEF} \implies \texttt{let}\ x = p\ \texttt{in}\ \pi_i(q) \in \text{NEF}$
- $\texttt{let}\ x = p\ \texttt{in}\ q \in \text{NEF} \implies p, q \in \text{NEF} \implies p[n/x] \in \text{NEF}$
- $\texttt{case}\ \texttt{in}_i(m)\ \texttt{of}\ (x_1.n_1|x_2.n_2) \in \text{NEF} \implies m, p, q \in \text{NEF} \implies \texttt{let}\ x = m\ \texttt{in}\ n_i \in \text{NEF}$
- $\pi_i(m_1, m_2) \in \text{NEF} \implies m_1, m_2 \in \text{NEF}$
- $\texttt{subst}\ \texttt{refl}\ m \in \text{NEF} \implies m \in \text{NEF}$
- $\texttt{ind}\ 0\ \texttt{of}\ (p|(x,y).q) \in \text{NEF} \implies m, p, q \in \text{NEF} \implies p \in \text{NEF}$
- $\texttt{ind}\ S(t)\ \texttt{of}\ (p|(x,y).q) \in \text{NEF} \implies m, p, q \in \text{NEF} \implies p, q\Big[m/x, (\texttt{ind}\ t\ \texttt{of}\ (p|(x_2,y_2).q))/y\Big] \in \text{NEF}$

For the `cofix` operator, we need only consider evaluation contexts F such that $F\{\texttt{let}\ z = \texttt{cofix}\ m\ \texttt{of}\ (x,y).p\ \texttt{in}\ q\} \in \text{NEF}$. This certainly means that $m, p, q \in \text{NEF}$, and, one can see that for any $n \in \text{NEF}$, $F\{n\} \in \text{NEF}$. Thus $\texttt{let}\ z = \texttt{cofix}\ m\ \texttt{of}\ (x,y).p\ \texttt{in}\ F\{q\} \in \text{NEF}$.

For the other reduction, assume $\texttt{let}\ z = \texttt{cofix}\ m\ \texttt{of}\ (x,y).p\ \texttt{in}\ D\{z\} \in \text{NEF}$. Then we have $m, p, D\{z\} \in \text{NEF}$, thus $\texttt{let}\ z = p\Big[(\lambda w.\texttt{cofix}\ w\ \texttt{of}\ (x_2,y_2).p)/x,\ m/y\Big]\ \texttt{in}\ D\{z\} \in \text{NEF}$ □

Lemma 7.8: NEF-Substitution and Reduction Closure in $\mathrm{ECC}_{\lambda\mu}$

Proof. (**i**) By induction on the structure of NEF terms. $x \notin fv(m) \implies m[n/x] = m \in \mathrm{NEF}$. This covers terms $y, \langle\rangle, \mathbf{1}, \mathcal{U}_i, \mathbf{0}, \mathtt{refl}$. From now on, we assume $x \in fv(m)$.

Base Case Assume $n \in \mathrm{NEF}$; then $x[n/x] = n \in \mathrm{NEF}$.

Inductive Cases Assume $p[n/x], q[n/x], m[n/x], A[n/x], B[n/x] \in \mathrm{NEF}$. Then:

- $((y : A) \to B)[n/x] = (y : A[n/x]) \to B[n/x] \in \mathrm{NEF}$
- $(\lambda y.m)[n/x] = \lambda y.(m[n/x]) \in \mathrm{NEF}$
- $(\mathtt{let}\ y = p\ \mathtt{in}\ q)[n/x] = (\mathtt{let}\ y = p[n/x]\ \mathtt{in}\ q[n/x]) \in \mathrm{NEF}$
- $((y : A) \times B)[n/x] = (y : A[n/x]) \times B[n/x] \in \mathrm{NEF}$
- $\pi_i(m)[n/x] = \pi_i(m[n/x]) \in \mathrm{NEF}$
- $(p, q)[n/x] = (p[n/x], q[n/x]) \in \mathrm{NEF}$
- $(A + B)[n/x] = (A[n/x] + B[n/x] \in \mathrm{NEF})$
- $\mathtt{in}_i(m)[n/x] = \mathtt{in}_i(m[n/x]) \in \mathrm{NEF}$
- $(\mathtt{case}\ m\ \mathtt{of}\ (y_1.p, y_2.q))[n/x] = \mathtt{case}\ m[n/x]\ \mathtt{of}\ (y_1.p[n/x], y_2.q[n/x]) \in \mathrm{NEF}$
- $(p =_A q)[n/x] = (p[n/x] =_A q[n/x]) \in \mathrm{NEF}$
- $(\mathtt{subst}\ p\ q)[n/x] = \mathtt{subst}\ p[n/x]\ q[n/x] \in \mathrm{NEF}$

(**ii**) By induction on the definition of reductions. Note that the (μ) reductions involve terms of the form $\mu\alpha.m$, and thus won't be NEF.

- $\mathtt{let}\ x = \mathtt{in}_i(p)\ \mathtt{in}\ q \in \mathrm{NEF} \implies p, q \in \mathrm{NEF} \implies \mathtt{let}\ y = p\ \mathtt{in}\ q[\mathtt{in}_i(y)/a] \in \mathrm{NEF}$
- $\pi_i(\mathtt{let}\ x = p\ \mathtt{in}\ q) \in \mathrm{NEF} \implies p, q \in \mathrm{NEF} \implies \mathtt{let}\ x = p\ \mathtt{in}\ \pi_i(q) \in \mathrm{NEF}$
- $\mathtt{let}\ x = p\ \mathtt{in}\ q \in \mathrm{NEF} \implies p, q \in \mathrm{NEF} \implies p[n/x] \in \mathrm{NEF}$
- $\mathtt{case}\ \mathtt{in}_i(m)\ \mathtt{of}\ (x_1.n_1 | x_2.n_2) \in \mathrm{NEF} \implies m, p, q \in \mathrm{NEF} \implies \mathtt{let}\ x = m\ \mathtt{in}\ n_i \in \mathrm{NEF}$
- $\pi_i(m_1, m_2) \in \mathrm{NEF} \implies m_1, m_2 \in \mathrm{NEF}$
- $\mathtt{subst}\ \mathtt{refl}\ m \in \mathrm{NEF} \implies m \in \mathrm{NEF}$
- $\kappa\{\mathtt{let}\ x = m\ \mathtt{in}\ n\} \in \mathrm{NEF}$ means that all terms appearing in the context κ are NEF[1], and that $m, n \in \mathrm{NEF} \implies \mathtt{let}\ x = m\ \mathtt{in}\ \kappa\{n\} \in \mathrm{NEF}$.

As it holds for the single step reductions, this holds by transitivity for $\to^*$, and it is easy to see this holds for the contextual closure relations. □

Lemma 7.9: NEF-Term Substitution

Proof by induction on the structure of M. Assume $\Gamma, x : C \vdash M : A \mid \Delta$ and $\Gamma \vdash_{\mathrm{NEF}} N : C \mid \Delta$. We write Γ' and Δ' for $\Gamma[N/x]$ and $\Delta[N/x]$, respectively.

- x:

$$
\begin{aligned}
& \Gamma, x : C \vdash x : C \mid \Delta \text{ and } \Gamma \vdash C : \mathcal{U}_i \mid \Delta && (Ax) \\
\implies & \Gamma' \vdash x[N/x] : C[N/x] \mid \Delta', \\
& \Gamma' \vdash C[N/x] : \mathcal{U}_i \mid \Delta' && \text{Induction} \\
\implies & \Gamma' \vdash x[N/x] : C \mid \Delta' \text{ and } \Gamma' \vdash C : \mathcal{U}_i \mid \Delta' && (x \notin fv(C), \text{ by } (Ax)) \\
\implies & \Gamma' \vdash N : C \mid \Delta' && \text{Defn}
\end{aligned}
$$

[1] This can be proved with a very simple induction on the definition of κ, needing only consider when it is of the form $\mathtt{in}_i(\kappa'), (\kappa', m), (v, \kappa), \mathtt{case}\ \kappa'\ \mathtt{of}\ (x_1.n_1 | x_2.n_2), pi_i(\kappa'), \mathtt{subst}\ \kappa'\ m, \mathtt{let}\ x = \kappa'\ \mathtt{in}\ m$.

- y:

$$\begin{array}{rll} & \Gamma, y : C \vdash y : C \mid \Delta \text{ and } \Gamma \vdash C : \mathcal{U}_i \mid \Delta & (Ax) \\ \Longrightarrow & \Gamma' \vdash y[N/x] : C[N/x] \mid \Delta', & \\ & \Gamma' \vdash C[N/x] : \mathcal{U}_i \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash y[N/x] : C[N/x] \mid \Delta' & (Ax) \\ \Longrightarrow & \Gamma' \vdash y : C[N/x] \mid \Delta' & \text{Defn} \end{array}$$

- $(y : m) \to n$

$$\begin{array}{rll} & A = \mathcal{U}_i, \quad \Gamma, x : C \vdash m : \mathcal{U}_i \mid \Delta \text{ and, } \Gamma, x : C, y : m \vdash n : \mathcal{U}_i \mid \Delta & (\Pi) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : \mathcal{U}_i \mid \Delta', & \\ & \Gamma', y : m[N/x] \vdash n[N/x] : \mathcal{U}_i \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash (y : m[N/x]) \to n[N/x] : \mathcal{U}_i \mid \Delta' & (\Pi) \\ \Longrightarrow & \Gamma' \vdash ((y : m) \to n)[N/x] : \mathcal{U}_i \mid \Delta' & \text{Defn} \end{array}$$

- $\lambda y.m$

$$\begin{array}{rll} & A = (y : E) \to F \text{ and } \Gamma, x : C, y : E \vdash m : F \mid \Delta & (\to I) \\ \Longrightarrow & \Gamma', y : E[N/x] \vdash m[N/x] : F[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash \lambda y.(m[N/x]) : (y : E[N/x]) \to F[N/x] \mid \Delta' & (\to I) \\ \Longrightarrow & \Gamma' \vdash (\lambda y.m)[N/x] : ((y : E) \to F)[N/x] \mid \Delta' & \text{Defn} \end{array}$$

- $\texttt{let}\ y = m\ \texttt{in}\ n : A$

Non-dependent:

$$\begin{array}{rll} & \Gamma, x : C \vdash m : B \mid \Delta \text{ and } \Gamma, x : C, y : B \vdash n : A \mid \Delta & (\texttt{let}) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : B[N/x] \mid \Delta', & \\ & \Gamma', y : B[N/x] \vdash n[N/x] : A[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash \texttt{let}\ y = m[N/x]\ \texttt{in}\ n[N/x] : A[N/x] \mid \Delta' & (\texttt{let}) \\ \Longrightarrow & \Gamma' \vdash (\texttt{let}\ y = m\ \texttt{in}\ n)[N/x] : A[N/x] \mid \Delta' & \text{Defn} \end{array}$$

Dependent:

$$\begin{array}{rll} & A = A'[m/y]\Gamma, x : C \vdash m : B \mid \Delta, & \\ & \Gamma, x : C, y : B \vdash_{\text{NEF}} n : A' \mid \Delta & (\texttt{let}^d) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : B \mid \Delta', & \\ & \Gamma', y : B \vdash_{\text{NEF}} n[N/x] : A'[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash \texttt{let}\ y = m[N/x]\ \texttt{in}\ n[N/x] : A'[N/x][(m[N/x])/y] \mid \Delta' & (\texttt{let}^d) \\ \Longrightarrow & \Gamma' \vdash (\texttt{let}\ y = m\ \texttt{in}\ n)[N/x] : A'[m/y][N/x] \mid \Delta' & \text{Defn} \end{array}$$

- mn

Non-dependent:

$$\begin{array}{rll} & \Gamma, x : C \vdash m : B \to A \mid \Delta \text{ and } \Gamma, x : C \vdash n : B \mid \Delta & (\to E) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : (B \to A)[N/x] \mid \Delta', & \\ & \Gamma' \vdash n[N/x] : B[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash m[N/x] : (B[N/x] \to A[N/x]) \mid \Delta' & \text{Defn} \\ \Longrightarrow & \Gamma' \vdash m[N/x]n[N/x] : A[N/x] \mid \Delta' & (\to E) \\ \Longrightarrow & \Gamma' \vdash (mn)[N/x] : A[N/x] \mid \Delta' & \text{Defn} \end{array}$$

Dependent:

$$\begin{array}{rll} & A = A'[n/y], \Gamma, x : C \vdash m : (y : B) \to A' \mid \Delta, & \\ & \Gamma, x : C \vdash_{\text{NEF}} n : B \mid \Delta & (\to E^d) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : ((y : B) \to A')[N/x] \mid \Delta', & \\ & \Gamma' \vdash_{\text{NEF}} n[N/x] : B[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash m[N/x] : (y : B[N/x]) \to A'[N/x] \mid \Delta' & \text{Defn} \\ \Longrightarrow & \Gamma' \vdash m[N/x]n[N/x] : A'[N/x][(n[N/x])/y] \mid \Delta' & (\to E^d) \\ \Longrightarrow & \Gamma' \vdash (mn)[N/x] : A'[n/y][N/x] \mid \Delta' & \text{Defn} \end{array}$$

- $(y : m) \times n$

$$\begin{array}{rll} & A = \mathcal{U}_i, \quad \Gamma, x : C \vdash m : \mathcal{U}_i \mid \Delta \text{ and } \Gamma, x : C, y : m \vdash n : \mathcal{U}_i \mid \Delta & (\Sigma) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : \mathcal{U}_i \mid \Delta', & \\ & \Gamma, y : m[N/x] \vdash n[N/x] : \mathcal{U}_i \mid \Delta & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash (y : m[N/x]) \times n[N/x] : \mathcal{U}_i \mid \Delta' & (\Sigma) \\ \Longrightarrow & \Gamma' \vdash ((y : m) \times n)[N/x] : \mathcal{U}_i \mid \Delta' & \text{Defn} \end{array}$$

- (m, n)

$$\begin{array}{rll} & A = (y : A_1) \times A_2, \quad \Gamma, x : C \vdash m : A_1 \mid \Delta \text{ and } \Gamma, x : C \vdash n : A_2[m/y] \mid \Delta & (\times I) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : A_1[N/x] \mid \Delta', & \\ & \Gamma' \vdash n[N/x] : (A_2[(m[N/x])/y])[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash n[N/x] : A_2[N/x][(m[N/x])/y] \mid \Delta' & \text{Defn} \\ \Longrightarrow & \Gamma' \vdash (m[N/x], n[N/x]) : (y : A_1[N/x]) \times A_2[N/x] \mid \Delta' & (\times I) \\ \Longrightarrow & \Gamma' \vdash (m, n)[N/x] : ((y : A_1) \times A_2)[N/x] \mid \Delta' & \text{Defn} \end{array}$$

- $\pi_1(m)$

$$\begin{array}{rll} \text{Non-dependent:} & \Gamma, x : C \vdash m : A \times B \mid \Delta & (\times E_1) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : (A \times B)[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash m[N/x] : A[N/x] \times B[N/x] \mid \Delta' & \text{Defn} \\ \Longrightarrow & \Gamma' \vdash \pi_1(m[N/x] : A[N/x]) \mid \Delta' & (\times E_1) \\ \Longrightarrow & \Gamma' \vdash (\pi_1(m))[N/x] : A[N/x] \mid \Delta' & \text{Defn} \end{array}$$

$$\begin{array}{rll} \text{Dependent:} & \Gamma, x : C \vdash m : (y : A) \times B \mid \Delta & (\times E_1^d) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : ((y : A) \times B)[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash m[N/x] : ((y : A[N/x]) \times B[N/x]) \mid \Delta' & \text{Defn} \\ \Longrightarrow & \Gamma' \vdash \pi_1(m[N/x] : A[N/x]) \mid \Delta' & (\times E_1) \\ \Longrightarrow & \Gamma' \vdash (\pi_1(m))[N/x] : A[N/x] \mid \Delta' & \text{Defn} \end{array}$$

- $\pi_2(m)$

$$\begin{array}{rll} \text{Non-dependent:} & \Gamma, x : C \vdash m : B \times A \mid \Delta & (\times E_2) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : (B \times A)[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash m[N/x] : B[N/x] \times A[N/x] \mid \Delta' & \text{Defn} \\ \Longrightarrow & \Gamma' \vdash \pi_2(m[N/x] : A[N/x]) \mid \Delta' & (\times E_2) \\ \Longrightarrow & \Gamma' \vdash (\pi_2(m))[N/x] : A[N/x] \mid \Delta' & \text{Defn} \end{array}$$

$$\begin{array}{rll} \text{Dependent:} & A = A'[\pi_1(m)/y], \quad \Gamma, x : C \vdash m : (y : B) \times A' \mid \Delta & (\times E_2^d) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : ((y : B) \times A')[N/x] \mid \Delta' & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash m[N/x] : (y : B[N/x]) \times A'[N/x] \mid \Delta' & \text{Defn} \\ \Longrightarrow & \Gamma' \vdash \pi_2(m[N/x] : A'[\pi_1(m[N/x])/y]) \mid \Delta' & (\times E_2) \\ \Longrightarrow & \Gamma' \vdash (\pi_2(m))[N/x] : A'[\pi_1(m)/y][N/x] \mid \Delta' & \text{Defn} \end{array}$$

- $m + n$

$$\begin{array}{rll} & A = \mathcal{U}_i, \quad \Gamma, x : C \vdash m : \mathcal{U}_i \mid \Delta \text{ and } \Gamma, x : C \vdash n : \mathcal{U}_i \mid \Delta & (+F) \\ \Longrightarrow & \Gamma' \vdash m[N/x] : \mathcal{U}_i \mid \Delta' \text{ and } \Gamma \vdash n[N/x] \mid \Delta & \text{Induction} \\ \Longrightarrow & \Gamma' \vdash m[N/x] + n[N/x] : \mathcal{U}_i \mid \Delta' & (+F) \\ \Longrightarrow & \Gamma' \vdash (m + n)[N/x] : \mathcal{U}_i \mid \Delta' & \text{Defn} \end{array}$$

- $\mathtt{in}_i(m)$: We show for $i = 1$; it is almost exactly the same for $i = 2$ (as there is no dependency).

$$
\begin{array}{lr}
\quad A = A_1 + A_2, \quad \Gamma, x : C \vdash m : A_1 \mid \Delta \text{ and } \Gamma, x : C \vdash A_i : \mathcal{U}_i \mid \Delta & (+I) \\
\Longrightarrow \Gamma' \vdash m[N/x] : A_1[N/x] \mid \Delta', & \\
\Longrightarrow \Gamma' \vdash A_i[N/x] : \mathcal{U}_i \mid \Delta' & \text{Induction} \\
\Longrightarrow \Gamma' \vdash \mathtt{in}_1(m[N/x]) : A_1[N/x] + A_2[N/x] \mid \Delta' & (+I) \\
\Longrightarrow \Gamma' \vdash (\mathtt{in}_1(m))[N/x] : (A_1 + A_2)[N/x] \mid \Delta' & \text{Defn}
\end{array}
$$

- $\mathtt{case}\ m \triangleright z.M\ \mathtt{of}\ (x_1.n_1 | x_2.n_2)$

Non-dependent: $M = A, \quad \Gamma, x : C \vdash m : B_1 + B_2 \mid \Delta,$

$$
\begin{array}{lr}
\quad \Gamma, x : C, x_i : B_i \vdash n_i : A \mid \Delta, & (+E) \\
\quad \Gamma, x : C \vdash A : \mathcal{U}_i \mid \Delta & \\
\Longrightarrow \Gamma' \vdash m[N/x] : (B_1 + B_2)[N/x] \mid \Delta', & \\
\quad \Gamma', x_i : B_i[N/x] \vdash n_i[N/x] : (A[N/x]) \mid \Delta', & \text{(Induction)} \\
\quad \Gamma' \vdash A[N/x] : \mathcal{U}_i \mid \Delta' & \\
\Longrightarrow \Gamma' \vdash m[N/x] : B_1[N/x] + B_2[N/x] \mid \Delta' & \text{(Defn)} \\
\Longrightarrow \Gamma[N/x] \vdash & \\
\qquad \mathtt{case}\ m[N/x] \triangleright z.(A[N/x])\ \mathtt{of}\ (x_1.(n_1[N/x]) | x_2.(n_2[N/x])) : A[N/x] & (+E) \\
\quad \mid \Delta[N/x] & \\
\Longrightarrow \Gamma' \vdash (\mathtt{case}\ m \triangleright z.A'\ \mathtt{of}\ (x_1.n_1 | x_2.n_2))[N/x] : A[N/x] \mid \Delta' & \text{Defn}
\end{array}
$$

Dependent: $A = A'[m/z], \quad \Gamma, x : C \vdash m : B_1 + B_2 \mid \Delta,$

$$
\begin{array}{lr}
\quad \Gamma, x : C, x_i : B_i \vdash n_i : A[\mathtt{in}_i(x_i)/x] \mid \Delta, & (+E^d) \\
\quad \Gamma x : C, z : B_1 + B_2 \vdash M : \mathcal{U}_i \mid \Delta & \\
\Longrightarrow \Gamma' \vdash m[N/x] : (B_1 + B_2)[N/x] \mid \Delta', & \\
\quad \Gamma', x_i : B_i[N/x] \vdash n_i[N/x] : (A'[\mathtt{in}_i(x_i)/z][N/x]) \mid \Delta', & \text{(Induction)} \\
\quad \Gamma', z : (B_1 + B_2)[N/x] \vdash A'[N/x] : \mathcal{U}_i \mid \Delta' & \\
\Longrightarrow \Gamma' \vdash m[N/x] : B_1[N/x] + B_2[N/x] \mid \Delta', & \\
\quad \Gamma', x_i : B_i[N/x] \vdash n_i[N/x] : A'[N/x][\mathtt{in}_i(x_i)/z] \mid \Delta', & \text{(Defn)} \\
\quad \Gamma', z : B_1[N/x] + B_2[N/x] \vdash A'[N/x] : \mathcal{U}_i \mid \Delta' & \\
\Longrightarrow \Gamma[N/x] \vdash & \\
\qquad \mathtt{case}\ m[N/x] \triangleright z.(A'[N/x])\ \mathtt{of}\ (x_1.(n_1[N/x]) | x_2.(n_2[N/x])) : A'[N/x][m/z] & (+E^d) \\
\quad \mid \Delta[N/x] & \\
\Longrightarrow \Gamma' \vdash (\mathtt{case}\ m \triangleright z.A'\ \mathtt{of}\ (x_1.n_1 | x_2.n_2))[N/x] : A'[m/z][N/x] \mid \Delta' & \text{Defn}
\end{array}
$$

- $m =_B n$

$$
\begin{array}{lr}
\quad A = \mathcal{U}_i, \quad \Gamma, x : C \vdash m : B \mid \Delta \text{ and } \Gamma, x : C \vdash n : B \mid \Delta & (=) \\
\Longrightarrow \Gamma' \vdash m[N/x] : [N/x]B \mid \Delta', & \\
\quad \Gamma \vdash n[N/x] : B[N/x] \mid \Delta & \text{Induction} \\
\Longrightarrow \Gamma' \vdash m[N/x] =_B n[N/x] \mid \Delta' & (=) \\
\Longrightarrow \Gamma' \vdash (m =_B n)[N/x] \mid \Delta' & \text{Defn}
\end{array}
$$

- $\mathtt{refl}$

$$
\begin{array}{lr}
\quad A = (m =_B m), \quad \Gamma, x : C \vdash B : \mathcal{U}_i \mid \Delta \text{ and } \Gamma, x : C \vdash m : B \mid \Delta & (\mathtt{refl}) \\
\Longrightarrow \Gamma' \vdash B[N/x] : \mathcal{U}_i \mid \Delta' \Gamma' \vdash m[N/x] : B[N/x] \mid \Delta' & \text{Induction} \\
\Longrightarrow \Gamma' \vdash \mathtt{refl}_m[N/x] : m[N/x] =_{B[N/x]} m[N/x] \mid \Delta' & (\mathtt{refl}) \\
\Longrightarrow \Gamma' \vdash \mathtt{refl}_m[N/x] : (m =_B m)[N/x] \mid \Delta' & \text{Defn}
\end{array}
$$

- $\texttt{subst}\ m\ n$

$$\begin{array}{rl}
& \Gamma, x : C, z : B \vdash A : \mathcal{U}_i \mid \Delta, \Gamma, x : C \vdash n : A[p/z] \mid \Delta, \Gamma, x : C \vdash m : p = q \mid \Delta \qquad (\texttt{subst}) \\
\Longrightarrow & \Gamma', z : B[N/x] \vdash A[N/x] : \mathcal{U}_i \mid \Delta', \\
& \Gamma' \vdash n[N/x] : A[p/z][N/x] \mid \Delta', \qquad \text{Induction} \\
& \Gamma' \vdash m[N/x] : (p =_B q)[N/x] \mid \Delta' \\
\Longrightarrow & \Gamma' \vdash n[N/x] : A[N/x][(p[N/x])/z] \mid \Delta', \\
& \Gamma' \vdash m[N/x] : p[N/x] =_{B[N/x]} q[N/x] \mid \Delta' \qquad \text{Defn} \\
\Longrightarrow & \Gamma' \vdash \texttt{subst}\ m[N/x]\ n[N/x] : A[N/x][(q[N/x])/z] \mid \Delta' \qquad (\texttt{subst}) \\
\Longrightarrow & \Gamma' \vdash (\texttt{subst}\ m\ n)[N/x] : A[q/z][N/x] \mid \Delta' \qquad \text{Defn}
\end{array}$$

- $\mu\alpha.m$

$$\begin{array}{rl}
& \Gamma, x : C \vdash m : \bot \mid \alpha : A, \Delta \qquad (\mu) \\
\Longrightarrow & \Gamma \vdash m[N/x] \mid \alpha : A[N/x], \Delta \qquad \text{Induction} \\
\Longrightarrow & \Gamma \vdash \mu\alpha.(m[N/x]) : A[N/x] \mid \Delta \qquad (\texttt{subst}) \\
\Longrightarrow & \Gamma \vdash (\mu\alpha.m)[N/x] : A[N/x] \mid \Delta \qquad \text{Defn}
\end{array}$$

- $[\alpha]m$

$$\begin{array}{rl}
& A = \bot \quad \Gamma, x : C \vdash m : B \mid \Delta \qquad (\mathit{name}) \\
\Longrightarrow & \Gamma' \vdash m[N/x] : B[N/x] \mid \Delta' \qquad \text{Induction} \\
\Longrightarrow & \Gamma' \vdash [\alpha](m[N/x]) : \bot \mid \alpha : B[N/x], \Delta' \qquad (\mathit{name}) \\
\Longrightarrow & \Gamma' \vdash ([\alpha]m)[N/x] : \bot[N/x] \mid \alpha : B[N/x], \Delta' \qquad \text{Defn}
\end{array}$$

- $\langle\rangle$: by (unit), $\Gamma' \vdash \langle\rangle : \mathbf{1} \mid \Delta' \Longrightarrow \Gamma' \vdash \langle\rangle[N/x] : \mathbf{1}[N/x] \mid \Delta'$
- $\mathbf{1}$: by $(\mathbf{1})$, $\Gamma' \vdash \mathbf{1} : \mathcal{U}_i \mid \Delta' \Longrightarrow \Gamma' \vdash \mathbf{1}[N/x] : \mathcal{U}_i[N/x] \mid \Delta'$
- $\mathbf{0}$: by $(\mathbf{0})$, $\Gamma' \vdash \mathbf{0} : \mathcal{U}_i \mid \Delta' \Longrightarrow \Gamma' \vdash \mathbf{0}[N/x] : \mathcal{U}_i[N/x] \mid \Delta'$
- $\mathcal{U}_i$: by $(\mathcal{U}_i)$, $\Gamma' \vdash \mathcal{U}_i : \mathcal{U}_{i+1} \mid \Delta' \Longrightarrow \Gamma' \vdash \mathcal{U}_i[N/x] : \mathcal{U}_{i+1}[N/x] \mid \Delta'$

Lemma 7.10: Subject Reduction

Proof by induction on reductions. For each reduction $M \to N$, assume $\Gamma \vdash M : A \mid \Delta$.

- $(\lambda x.m)n \to \texttt{let}\ x = n\ \texttt{in}\ m$

$$\begin{array}{rll}
\text{Non-dependent:} & \Gamma \vdash \lambda x.m : B \to A \mid \Delta \text{ and } \Gamma \vdash n : B \mid \Delta & (\to E) \\
& \Longrightarrow \Gamma, x : B \vdash m : A \mid \Delta & (\to I) \\
& \Longrightarrow \Gamma \vdash \texttt{let}\ x = n\ \texttt{in}\ m : A \mid \Delta & (\texttt{let})
\end{array}$$

$$\begin{array}{rll}
\text{Dependent:} & A = A'[n/x]\Gamma \vdash \lambda x.m : (x : B) \to A' \mid \Delta \text{ and } \Gamma \vdash_{\text{NEF}} n : B \mid \Delta & (\to E^d) \\
& \Longrightarrow \Gamma, x : B \vdash m : A' \mid \Delta & (\to I) \\
& \Longrightarrow \Gamma \vdash \texttt{let}\ x = n\ \texttt{in}\ m : A'[n/x] \mid \Delta & (\texttt{let}^d)
\end{array}$$

- $\texttt{let}\ x = v\ \texttt{in}\ m \to m[v/x]$

$$\begin{array}{rll}
\text{Non-dependent:} & \Gamma, x : B \vdash m : A \mid \Delta, \quad \Gamma \vdash v : B \mid \Delta & (\texttt{let}) \\
& \Longrightarrow \Gamma \vdash m[v/x] : A \mid \Delta & \text{Lemma 7.9}
\end{array}$$

$$\begin{array}{rll}
\text{Dependent:} & A = A'[v/x]\Gamma, x : B \vdash m : A \mid \Delta, \quad \Gamma \vdash_{\text{NEF}} v : B \mid \Delta & (\texttt{let}^d) \\
& \Longrightarrow \Gamma \vdash m[v/x] : A'[v/x] \mid \Delta & \text{Lemma 7.9}
\end{array}$$

- $\kappa\{\texttt{let}\ x = m\ \texttt{in}\ n\} \to \texttt{let}\ x = m\ \texttt{in}\ \kappa\{n\}$

Non-dependent: $\Gamma \vdash \kappa\{\texttt{let}\ x = m\ \texttt{in}\ n\} : A \mid \Delta$

Assume that there is a type B (a subterm of A) such that:

$\Gamma \vdash \texttt{let}\ x = m\ \texttt{in}\ n : B \mid \Delta$.

$\Longrightarrow$ The hole in κ has type B, and $\Gamma \vdash n : B \mid \Delta$ (let)

$\Longrightarrow \Gamma \vdash \kappa\{n\} : A \mid \Delta$, as n has the same type as $\bullet$

$\Longrightarrow \Gamma \vdash \texttt{let}\ x = m\ \texttt{in}\ \kappa\{n\} : A \mid \Delta$ (let)

Dependent: $\Gamma \vdash \kappa\{\texttt{let}\ x = m\ \texttt{in}\ n\} : A \mid \Delta$

$\Longrightarrow A = A'[m/x]$, as the type assignment for $\kappa\{\texttt{let}\ x = m\ \texttt{in}\ n\}$ will at some point use the $\texttt{let}^d$ rule, which will bind x in a subterm of A.

Assume that there is a type B (a subterm of A) such that:

$\Gamma \vdash \texttt{let}\ x = m\ \texttt{in}\ n : B \mid \Delta$.

$\Longrightarrow B = B'[m/x]$, $\Gamma \vdash_{\text{NEF}} m : C \mid \Delta$, $\Gamma, x : C \vdash n : B' \mid \Delta$ ($\texttt{let}^d$)

$\Longrightarrow$ The hole in κ has type B', with $x \in fv(B')$

$\Longrightarrow \Gamma, x : C \vdash \kappa\{n\} : A' \mid \Delta$, as n has the same type as $\bullet$

$\Longrightarrow \Gamma \vdash \texttt{let}\ x = m\ \texttt{in}\ \kappa\{n\} : A'[m/x] \mid \Delta$ ($\texttt{let}^d$)

- $\texttt{case}\ \texttt{in}_i(m) \triangleright z.C\ \texttt{of}\ (x_1.n_1 | x_2.n_2) \to \texttt{let}\ x_i = m\ \texttt{in}\ n_i$

Non-dependent: $A = C[\texttt{in}_i(m)/z], \Gamma \vdash \texttt{in}_i(m) : B_1 + B_2 \mid \Delta$, $\Gamma \vdash C : \mathcal{U}_i \mid \Delta$,

$\Gamma, x_i : B_i \vdash n_i : C \mid \Delta$ ($+E$)

$\Longrightarrow \Gamma \vdash m : B_i \mid \Delta$ ($+I_i$)

$\Longrightarrow \Gamma \vdash \texttt{let}\ x_i = m\ \texttt{in}\ n_i : C \mid \Delta$ (let)

$\Longrightarrow \Gamma \vdash \texttt{let}\ x_i = m\ \texttt{in}\ n_i : C \mid \Delta$ (Defn)

Dependent: $A = C[\texttt{in}_i(m)/z], \Gamma \vdash_{\text{NEF}} \texttt{in}_i(m) : B_1 + B_2 \mid \Delta$,

$\Gamma, z : B_1 + B_2 \vdash C : \mathcal{U}_i \mid \Delta$, ($+E^d$)

$\Gamma, x_i : B_i \vdash n_i : C[\texttt{in}_i(x_i)/z] \mid \Delta$

$\Longrightarrow \Gamma \vdash_{\text{NEF}} m : B_i \mid \Delta$ ($+I_i$)

$\Longrightarrow \Gamma \vdash \texttt{let}\ x_i = m\ \texttt{in}\ n_i : (C[\texttt{in}_i(x_i)/z])[m/x_i] \mid \Delta$ ($\texttt{let}^d$)

$\Longrightarrow \Gamma \vdash \texttt{let}\ x_i = m\ \texttt{in}\ n_i : C[\texttt{in}_i(m)/z] \mid \Delta$ (Defn)

- $\pi_1(m_1, m_2) \to m_1$

Non-dependent: $\Gamma \vdash (m_1, m_2) : A \times B \mid \Delta$ ($\times E_1$)

$\Longrightarrow \Gamma \vdash m_1 : A \mid \Delta$ ($\times I$)

Dependent: $\Gamma \vdash_{\text{NEF}} (m_1, m_2) : (x : A) \times B \mid \Delta$ ($\times E_1^d$)

$\Longrightarrow \Gamma \vdash_{\text{NEF}} m_1 : A \mid \Delta$ ($\times I$)

- $\pi_2(m_1, m_2) \to m_2$

Non-dependent: $\Gamma \vdash (m_1, m_2) : B \times A \mid \Delta$ ($\times E_2$)

$\Longrightarrow \Gamma \vdash m_2 : A \mid \Delta$ ($\times I$)

Dependent: $A = A'[\pi_1(m_1, m_2)/x]$, $\Gamma \vdash_{\text{NEF}} (m_1, m_2) : (x : B) \times A' \mid \Delta$ ($\times E_2^d$)

$\Longrightarrow \Gamma \vdash_{\text{NEF}} m_2 : A'[\pi_1(m_1, m_2)/x] \mid \Delta$ ($\times I$)

- $\texttt{subst refl}\ m \to m$

$$\begin{aligned}
& A = B[q/x], \quad \Gamma \vdash \texttt{refl} : p = q \mid \Delta, \quad \Gamma \vdash m : B[p/x] \mid \Delta && (\texttt{subst}) \\
\Longrightarrow\ & \Gamma \vdash \texttt{refl} : p = p \mid \Delta, \text{ so } q \text{ is syntactically equal to } p && (\texttt{refl}) \\
\Longrightarrow\ & B[q/x] = B[p/x] = A \\
\Longrightarrow\ & \Gamma \vdash m : A \mid \Delta
\end{aligned}$$

- $v(\mu\alpha.m) \to \mu\alpha.m[[\alpha]v \bullet /[\alpha]\bullet]$

$$\begin{aligned}
& \Gamma \vdash v : B \to A \mid \Delta, \quad \Gamma \vdash \mu\alpha.m : B \mid \Delta && (\to E) \\
\Longrightarrow\ & \Gamma \vdash m : \bot \mid \alpha : B, \Delta && (\mu) \\
\Longrightarrow\ & \Gamma \vdash n : B \mid \Delta, \text{ for each } n \text{ such that } [\alpha]n \text{ is a subterm of } m && (name) \\
\Longrightarrow\ & \Gamma \vdash vn : A \mid \Delta && (\to E) \\
\Longrightarrow\ & \Gamma \vdash [\alpha]vn : \bot \mid \alpha : A, \Delta && (name) \\
\Longrightarrow\ & \Gamma \vdash \mu\alpha.m[[\alpha]v \bullet /[\alpha]\bullet] : A \mid \Delta
\end{aligned}$$

- $(\mu\alpha.m)n \to \mu\alpha.m[[\alpha] \bullet n/[\alpha]\bullet]$

$$\begin{aligned}
& \Gamma \vdash n : B \mid \Delta, \quad \Gamma \vdash \mu\alpha.m : B \to A \mid \Delta && (\to E) \\
\Longrightarrow\ & \Gamma \vdash m : \bot \mid \alpha : B \to A, \Delta && (\mu) \\
\Longrightarrow\ & \Gamma \vdash p : B \to A \mid \Delta, \text{ for each } p \text{ such that } [\alpha]p \text{ is a subterm of } m && (name) \\
\Longrightarrow\ & \Gamma \vdash pn : A \mid \Delta && (\to E) \\
\Longrightarrow\ & \Gamma \vdash [\alpha]pn : \bot \mid \alpha : A, \Delta && (name) \\
\Longrightarrow\ & \Gamma \vdash \mu\alpha.m[[\alpha] \bullet n/[\alpha]\bullet] : A \mid \Delta
\end{aligned}$$

- $\texttt{let}\ x = \mu\alpha.m\ \texttt{in}\ n \to \mu\alpha.m[[\alpha]\texttt{let}\ x = \bullet\ \texttt{in}\ n/[\alpha]\bullet]$

$$\begin{aligned}
& \Gamma, x : B \vdash n : A \mid \Delta, \quad \Gamma \vdash \mu\alpha.m : B \mid \Delta && (\texttt{let}) \\
\Longrightarrow\ & \Gamma \vdash m : \bot \mid \alpha : B, \Delta && (\mu) \\
\Longrightarrow\ & \Gamma \vdash p : B \mid \Delta, \text{ for each } p \text{ such that } [\alpha]p \text{ is a subterm of } m && (name) \\
\Longrightarrow\ & \Gamma \vdash \texttt{let}\ x = p\ \texttt{in}\ n : A \mid \Delta && (\texttt{let}) \\
\Longrightarrow\ & \Gamma \vdash [\alpha]\texttt{let}\ x = p\ \texttt{in}\ n : \bot \mid \alpha : A, \Delta && (name) \\
\Longrightarrow\ & \Gamma \vdash \mu\alpha.m[[\alpha] \bullet n/[\alpha]\bullet] : A \mid \Delta
\end{aligned}$$

- $\mu\alpha.[\alpha]m \to m \qquad (\alpha \notin fn(m))$

$$\begin{aligned}
& \Gamma \vdash [\alpha]m : \bot \mid \alpha : A, \Delta && (\mu) \\
\Longrightarrow\ & \Gamma \vdash m : A \mid \Delta && (name)
\end{aligned}$$

- $[\beta]\mu\delta.m \to m[\beta/\delta]$

  So $A : \bot$ by (*name*), and;

$$\begin{aligned}
& \Gamma \vdash \mu\delta.m : B \mid \beta : B, \Delta && (name) \\
\Longrightarrow\ & \Gamma \vdash m : \bot \mid \beta : B, \delta : B, \Delta && (\mu)
\end{aligned}$$

  We also need that $m[\beta/\delta] : A \implies m : A$. As $\beta : B$ and $\delta : B$, then substituting β for δ will not change the type of a term. Indeed, for any named term $[\delta]n$:

$$\begin{aligned}
& \Gamma \vdash [\delta]n : \bot \mid \delta : B, \beta : B, \Delta \\
\Longrightarrow\ & \Gamma \vdash n : B \mid \delta : B, \beta : B, \Delta && (name) \\
\Longrightarrow\ & \Gamma \vdash [\beta]n : \bot \mid \delta : B, \beta : B, \Delta && (name)
\end{aligned}$$

- $\pi_i(\mu\alpha.m) \to \mu\alpha.m[[\alpha]\pi_i(\bullet)/[\alpha]\bullet]$
  $A = A_i$

$$\begin{aligned}
&\Gamma \vdash \mu\alpha.m : A_1 \times A_2 \mid \Delta && (\times E)\\
\Longrightarrow\ &\Gamma \vdash m : \bot \mid \alpha : A_1 \times A_2, \Delta && (\mu)\\
\Longrightarrow\ &\Gamma \vdash n : A_1 \times A_2 \mid \Delta, \text{ for each } n \text{ such that } [\alpha]n \text{ is a subterm of } m && (name)\\
\Longrightarrow\ &\Gamma \vdash \pi_i(n) : A_i \mid \Delta && (\times E)\\
\Longrightarrow\ &\Gamma \vdash [\alpha]\pi_i(n) : \bot \mid \alpha : A_i, \Delta && (name)\\
\Longrightarrow\ &\Gamma \vdash \mu\alpha.m[[\alpha]v\ \bullet/[\alpha]\bullet] : A_i \mid \Delta
\end{aligned}$$

- $\mathtt{in}_i(\mu\alpha.m) \to \mu\alpha.m[[\alpha]\mathtt{in}_i(\bullet)/[\alpha]\bullet]$
  $A = A_1 + A_2$

$$\begin{aligned}
&\Gamma \vdash \mu\alpha.m : A_i \mid \Delta && (+I)\\
\Longrightarrow\ &\Gamma \vdash m : \bot \mid \alpha : A_i, \Delta && (\mu)\\
\Longrightarrow\ &\Gamma \vdash n : A_i \mid \Delta, \text{ for each } n \text{ such that } [\alpha]n \text{ is a subterm of } m && (name)\\
\Longrightarrow\ &\Gamma \vdash \mathtt{in}_i(n) : A_1 + A_2 \mid \Delta && (+I)\\
\Longrightarrow\ &\Gamma \vdash [\alpha]\mathtt{in}_i(n) : \bot \mid \alpha : A_1 + A_2, \Delta && (name)\\
\Longrightarrow\ &\Gamma \vdash \mu\alpha.m[[\alpha]v\ \bullet/[\alpha]\bullet] : A_1 + A_2 \mid \Delta
\end{aligned}$$

- $(v, \mu\alpha.m) \to \mu\alpha.m[[\alpha](v, \bullet)/[\alpha]\bullet]$

$$\begin{aligned}
&A = A_1 \times A_2, \quad \Gamma \vdash v : A_1 \mid \Delta, \quad \Gamma \vdash \mu\alpha.m : A_2 \mid \Delta && (\times I)\\
\Longrightarrow\ &\Gamma \vdash m : \bot \mid \alpha : A_2, \Delta && (\mu)\\
\Longrightarrow\ &\Gamma \vdash n : A_2 \mid \Delta, \text{ for each } n \text{ such that } [\alpha]n \text{ is a subterm of } m && (name)\\
\Longrightarrow\ &\Gamma \vdash (v, n) : A_1 \times A_2 \mid \Delta && (\times I)\\
\Longrightarrow\ &\Gamma \vdash [\alpha](v, n) : \bot \mid \alpha : A_1 \times A_2, \Delta && (name)\\
\Longrightarrow\ &\Gamma \vdash \mu\alpha.m[[\alpha](v, \bullet)/[\alpha]\bullet] : A_1 \times A_2 \mid \Delta
\end{aligned}$$

- $(\mu\alpha.m, n) \to \mu\alpha.m[[\alpha](\bullet, n)/[\alpha]\bullet]$

$$\begin{aligned}
&A = A_1 \times A_2, \quad \Gamma \vdash \mu\alpha.m : A_1 \mid \Delta, \quad \Gamma \vdash n : A_2 \mid \Delta && (\times I)\\
\Longrightarrow\ &\Gamma \vdash m : \bot \mid \alpha : A_1, \Delta && (\mu)\\
\Longrightarrow\ &\Gamma \vdash p : A_1 \mid \Delta, \text{ for each } p \text{ such that } [\alpha]p \text{ is a subterm of } m && (name)\\
\Longrightarrow\ &\Gamma \vdash (p, n) : A_1 \times A_2 \mid \Delta && (\times I)\\
\Longrightarrow\ &\Gamma \vdash [\alpha](p, n) : \bot \mid \alpha : A_1 \times A_2, \Delta && (name)\\
\Longrightarrow\ &\Gamma \vdash \mu\alpha.m[[\alpha](\bullet, n)/[\alpha]\bullet] : A_1 \times A_2 \mid \Delta
\end{aligned}$$

- $\mathtt{case}\ \mu\alpha.m \triangleright z.A\ \mathtt{of}\ (x_1.n_1 | x_2.n_2) \to \mu\alpha.m[[\alpha]\mathtt{case}\ \bullet \triangleright z.A\ \mathtt{of}\ (x_1.n_1 | x_2.n_2)/[\alpha]\bullet]$

$$\begin{aligned}
&\Gamma \vdash \mu\alpha.m : A_1 + A_2 \mid \Delta, \quad \Gamma, x_i : A_i \vdash n_i : A \mid \Delta, \quad \Gamma \vdash A : \mathcal{U}_i \mid \Delta && (+E)\\
\Longrightarrow\ &\Gamma \vdash m : \bot \mid \alpha : A_1 + A_2, \Delta && (\mu)\\
\Longrightarrow\ &\Gamma \vdash p : A_1 + A_2 \mid \Delta, \text{ for each } p \text{ such that } [\alpha]p \text{ is a subterm of } m && (name)\\
\Longrightarrow\ &\Gamma \vdash \mathtt{case}\ p \triangleright z.A\ \mathtt{of}\ (x_1.n_1 | x_2.n_2) : A \mid \Delta && (+E)\\
\Longrightarrow\ &\Gamma \vdash [\alpha]\mathtt{case}\ p \triangleright z.A\ \mathtt{of}\ (x_1.n_1 | x_2.n_2) : \bot \mid \alpha : A, \Delta && (name)\\
\Longrightarrow\ &\Gamma \vdash \mu\alpha.m\Big[[\alpha]\mathtt{case}\ \bullet \triangleright z.A\ \mathtt{of}\ (x_1.n_1 | x_2.n_2)/[\alpha]\bullet\Big] : A \mid \Delta
\end{aligned}$$

- $\mathtt{subst}\ (\mu\alpha.m)\ n \to \mu\alpha.m[[\alpha]\mathtt{subst}\ (\bullet)\ n/[\alpha]\bullet]$

$$\begin{aligned}
&A = B[q/x], \quad \Gamma \vdash \mu\alpha.m : p = q \mid \Delta, \quad \Gamma \vdash n : B[p/x] \mid \Delta && (\mathtt{subst})\\
\Longrightarrow\ &\Gamma \vdash m : \bot \mid \alpha : p = q, \Delta && (name)\\
\Longrightarrow\ &\Gamma \vdash l : p = q \mid \Delta, \text{ for each } l \text{ such that } [\alpha]l \text{ is a subterm of } m && (name)\\
\Longrightarrow\ &\Gamma \vdash \mathtt{subst}\ l\ n : B[q/x] \mid \Delta && (\mathtt{subst})\\
\Longrightarrow\ &\Gamma \vdash [\alpha]\mathtt{subst}\ l\ n : \bot \mid \alpha : B[q/x], \Delta && (\mathtt{subst})\\
\Longrightarrow\ &\Gamma \vdash \mu\alpha.m[[\alpha]\mathtt{subst}\ (\bullet)\ n/[\alpha]\bullet] : B[q/x] \mid \Delta
\end{aligned}$$

Lemma 7.12

By induction on the structure of normal forms. We need only consider normal forms that aren't values. Assume $\vdash m : A$. The main technique we use is that the typing rule that applies to a normal form is determined by the head of the normal form; and we show that this head cannot be given the correct type. If m contains a free (co)variable, it cannot be typed by an empty (co)context. Thus, we only consider closed normal terms. For $m, n, N \in \text{NF}$, these are:

- $m = \mu\alpha.[\beta]n \quad (n \neq \mu\delta.n')$: Then β must be *top*, as the co-context is empty. Thus $\vdash n : \bot$, which contradicts consistency. Thus m is not typeable.

- $m = \texttt{case}\ n \triangleright z.N\ \texttt{of}\ (x_1.n_1|x_2.n_2) \quad (n \neq \texttt{in}_i(n'), \mu\alpha.n')$ Then $n : B_1 + B_2$; but $n \neq \texttt{in}_i(n')$ and $n \in \text{NF}$, so won't evaluate to a term of the form $\texttt{in}_i(n')$, and thus cannot be typed by $B_1 + B_2$, meaning m is not typeable by (case).

- $m = \pi_i(n) \quad (n \neq (n_1, n_2), \mu\alpha.n')$ Then $n : (x : A) \times B$, but n doesn't reduce to terms of the form $(n_1, n_2), \mu\alpha.n'$, so cannot be typed by $(x : A) \times B$, so m is not typeable by $(\times E_i)$.

- $m = xn_1 \cdots n_k \quad (n_i \neq \mu\alpha.n')$ x is an free variable, so m can't be typed by an empty context.

- $m = \texttt{subst}\ n_1\ n_2 \quad (n_1 \neq \texttt{refl})$ Then $n_1 : m = n$, but n_1 doesn't reduce to $\texttt{refl}$, so $n_1 \in \text{NF} \setminus \texttt{refl}$, thus no form of n_1 can be typed by $m = n$, so m is not typeable by (subst)

B.2 Type Systems

$\text{ECC}_{\lambda\mu}$ Reductions

$\text{ECC}_{\lambda\mu}$ Reductions

$$
\begin{array}{rcl}
(\lambda x.m)n & \to & \texttt{let}\ x = n\ \texttt{in}\ m, (m \neq \mu\alpha.m') \\
\texttt{let}\ x = v\ \texttt{in}\ m & \to & m[v/x] \\
\kappa\{\texttt{let}\ x = m\ \texttt{in}\ n\} & \to & \texttt{let}\ x = m\ \texttt{in}\ \kappa\{n\} \\
\texttt{case}\ \texttt{in}_i(m) \triangleright z.A\ \texttt{of}\ (x_1.n_1|x_2.n_2) & \to & \texttt{let}\ x_i = m\ \texttt{in}\ n_i \\
\pi_i(v_1, v_2) & \to & v_i \\
\texttt{subst}\ \texttt{refl}\ m & \to & m \\
\kappa\{\mu\alpha.m\} & \to & \mu\alpha.m[[\alpha]\kappa\{\bullet\}/[\alpha]\bullet]
\end{array}
$$

Explicitly, the (μ) reductions are:

$$
\begin{array}{rcll}
(\mu\alpha.m)n & \to & \mu\alpha.m[[\alpha]\bullet\, n/[\alpha]\bullet] & \\
v(\mu\alpha.m) & \to & \mu\alpha.m[[\alpha]v\,\bullet/[\alpha]\bullet] & \\
\mu\alpha.[\alpha]m & \to & m & (\alpha \notin fn(m)) \\
[\beta]\mu\delta.m & \to & m[\beta/\delta] & \\
\pi_i(\mu\alpha.m) & \to & \mu\alpha.m[[\alpha]\pi_i(\bullet)/[\alpha]\bullet] & \\
(v, \mu\alpha.m) & \to & \mu\alpha.m[[\alpha](v, \bullet)/[\alpha]\bullet] & \\
(\mu\alpha.m, n) & \to & \mu\alpha.m[[\alpha](\bullet, n)/[\alpha]\bullet] & \\
\texttt{let}\ x = \mu\alpha.m\ \texttt{in}\ n & \to & \mu\alpha.m[[\alpha]\texttt{let}\ x = \bullet\ \texttt{in}\ n/[\alpha]\bullet] & \\
\texttt{in}_i(\mu\alpha.m) & \to & \mu\alpha.m[[\alpha]\texttt{in}_i(\bullet)/[\alpha]\bullet] & \\
\texttt{case}\ \mu\alpha.m \triangleright z.A\ \texttt{of}\ (x_1.n_1|x_2.n_2) & \to & \mu\alpha.m[[\alpha]\texttt{case}\ \bullet \triangleright z.A\ \texttt{of}\ (x_1.n_1|x_2.n_2)/[\alpha]\bullet] & \\
\texttt{subst}\ (\mu\alpha.m)\ n & \to & \mu\alpha.m[[\alpha]\texttt{subst}\ (\bullet)\ n/[\alpha]\bullet] &
\end{array}
$$

Where, in the critical pair $(\lambda x.m)(\mu\alpha.m)$, we prioritise the μ-reduction.

$\mathbf{ECC}_{\lambda\mu}$ Subtyping

Subtyping Rules for $\mathbf{ECC}_{\lambda\mu}$ [59, 53]

$$\frac{\Gamma \vdash t : A \mid \Delta \qquad \Gamma \vdash A \leqslant B \mid \Delta}{\Gamma \vdash t : B \mid \Delta} \qquad \frac{}{\Gamma \vdash \mathcal{U}_i \leqslant \mathcal{U}_{i+1} \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_2 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_1 \simeq A_2 : \mathcal{U} \mid \Delta \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2 \mid \Delta}{\Gamma \vdash (x : A_1) \to B_1 \leqslant (x : A_2) \to B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_2 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_1 \simeq A_2 : \mathcal{U} \mid \Delta \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2 \mid \Delta}{\Gamma \vdash (x : A_1) \times B_1 \leqslant (x : A_2) \times B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 \leqslant A_2 \mid \Delta \qquad \Gamma \vdash B_1 \leqslant B_2 \mid \Delta}{\Gamma \vdash A_1 + B_1 \leqslant A_2 + B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash A_2 : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash A_1 \simeq A_2 : \mathcal{U}_i \mid \Delta}{\Gamma \vdash A_1 \leqslant A_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 \leqslant A_2 \mid \Delta \qquad \Gamma \vdash A_2 \leqslant A_3 \mid \Delta}{\Gamma \vdash A_1 \leqslant A_3 \mid \Delta}$$

B.3 Bidirectional Algorithms for $\mathbf{ECC}_{\lambda\mu}$

Bidirectional Type Assignments

The rules are derived by combining the bidirectional style of [59] with the type system in Definition 7.7

Bidirectional Typing

Valid Contexts

$$\frac{}{\varnothing \vdash \cdot \mid \varnothing}\ (\cdot) \qquad \frac{\Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta \triangleright B}{\Gamma, x : A \vdash x \Rightarrow A \mid \Delta \triangleright x}\ (Ax) \qquad \frac{\Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash \cdot \mid \alpha : A, \Delta}\ (\alpha x)$$

Function Introduction/Formation

$$\frac{C \to_{whnf} (x : A) \to B \qquad \Gamma, x : A \vdash m \Leftarrow B \mid \Delta \triangleright t}{\Gamma \vdash \lambda x.m \Leftarrow C \mid \Delta \triangleright \lambda x.t}\ (\to I)$$

$$\frac{\begin{array}{cc} C_1 \to_{whnf} \mathcal{U}_i & C_2 \to_{whnf} \mathcal{U}_j \\ \Gamma \vdash e_1 \Rightarrow C_1 \mid \Delta \triangleright A & \Gamma, x : A \vdash e_2 \Rightarrow C_2 \mid \Delta \triangleright B \end{array}}{\Gamma \vdash (x : e_1) \to e_2 \Rightarrow \mathcal{U}_{i \sqcup j} \mid \Delta \triangleright (x : A) \to B}\ (\Pi)$$

Pair Introduction/Formation

$$\frac{C \to_{whnf} (x : A) \times B \qquad \Gamma \vdash m \Leftarrow A \mid \Delta \triangleright t \qquad \Gamma \vdash n \Leftarrow B[t/x] \mid \Delta \triangleright u}{\Gamma \vdash (m, n) \Leftarrow C \mid \Delta \triangleright (t, u)}\ (\times I)$$

$$\frac{\begin{array}{cc} C_1 \to_{whnf} \mathcal{U}_i & C_2 \to_{whnf} \mathcal{U}_j \\ \Gamma \vdash e_1 \Rightarrow C_1 \mid \Delta \triangleright A & \Gamma, x : A \vdash e_2 \Rightarrow C_2 \mid \Delta \triangleright B \end{array}}{\Gamma \vdash (x : e_1) \times e_2 \Rightarrow \mathcal{U}_{i \sqcup j} \mid \Delta \triangleright (x : A) \times B}\ (\Sigma)$$

Coproduct Introduction/Formation

$$\frac{C \to_{whnf} A + B \qquad \Gamma \vdash m \Leftarrow A \mid \Delta \triangleright t \qquad \Gamma \vdash B \Leftarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash \mathrm{in}_1(m) \Leftarrow C \mid \Delta \triangleright \mathrm{in}_1(t)}\ (+I_1)$$

$$\frac{C \to_{whnf} A + B \qquad \Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta \triangleright \qquad \Gamma \vdash m \Leftarrow B \mid \Delta \triangleright t}{\Gamma \vdash \mathrm{in}_2(m) \Leftarrow C \mid \Delta \triangleright \mathrm{in}_2(t)}\ (+I_2)$$

Non-Dependent Elimination

$$\frac{\Gamma \vdash m \Rightarrow C \mid \Delta \triangleright t \qquad C \to_{whnf} (x : A) \to B \qquad x \notin fv(B) \qquad \Gamma \vdash n \Leftarrow A \mid \Delta \triangleright u}{\Gamma \vdash mn \Rightarrow B \mid \Delta \triangleright tu}\ (\to E)$$

$$\frac{\Gamma \vdash m \Rightarrow A \mid \Delta \triangleright t \qquad \Gamma, x : A \vdash n \Rightarrow B \mid \Delta \triangleright u \qquad x \notin fv(B)}{\Gamma \vdash \mathtt{let}\ x = m\ \mathtt{in}\ n \Rightarrow B \mid \Delta \triangleright \mathtt{let}\ x = t\ \mathtt{in}\ u}\ (\mathrm{let})$$

$$\frac{\Gamma \vdash m \Rightarrow C \mid \Delta \triangleright t \qquad C \to_{whnf} (x : A) \times B \qquad x \notin fv(B)}{\Gamma \vdash \pi_1(m) \Rightarrow A \mid \Delta \triangleright \pi_1(t)}\ (\times E_1)$$

$$\frac{\Gamma \vdash m \Rightarrow C \mid \Delta \triangleright t \qquad C \to_{whnf} (x : A) \times B \qquad x \notin fv(B)}{\Gamma \vdash \pi_2(m) \Rightarrow B \mid \Delta \triangleright \pi_2(t)}\ (\times E_2)$$

$$\frac{\begin{array}{lll}\Gamma \vdash m \Rightarrow D \mid \Delta \triangleright t & z \notin fv(C) & \Gamma, x : A \vdash n_1 \Rightarrow E \mid \Delta \triangleright u_1 \\ D \to_{whnf} A + B & \Gamma \vdash C \Rightarrow \mathcal{U}_i \mid \Delta \triangleright E & \Gamma, y : B \vdash n_2 \Rightarrow E \mid \Delta \triangleright u_2\end{array}}{\Gamma \vdash \mathtt{case}\ m \triangleright z.C\ \mathtt{of}\ (x.n_1 | y.n_2) \Rightarrow E \mid \Delta \triangleright \mathtt{case}\ t \triangleright z.E\ \mathtt{of}\ (x.u_1 | y.u_2)}\ (+E)$$

Dependent Elimination

$$\frac{\Gamma \vdash m \Rightarrow C \mid \Delta \triangleright t \qquad C \to_{whnf} (x : A) \to B \qquad \Gamma \vdash_{\mathrm{NEF}} n \Leftarrow A \mid \Delta \triangleright u}{\Gamma \vdash mn \Rightarrow B[n/x] \mid \Delta \triangleright tu}\ (\to E^d)$$

$$\frac{\Gamma \vdash_{\mathrm{NEF}} m \Rightarrow A \mid \Delta \triangleright t \qquad \Gamma, x : A \vdash n \Rightarrow B \mid \Delta \triangleright u}{\Gamma \vdash \mathtt{let}\ x = m\ \mathtt{in}\ n \Rightarrow B[m/a] \mid \Delta \triangleright \mathtt{let}\ x = t\ \mathtt{in}\ u}\ (\mathrm{let}^d)$$

$$\frac{\Gamma \vdash_{\mathrm{NEF}} m \Rightarrow C \mid \Delta \triangleright t \qquad C \to_{whnf} (x : A) \times B}{\Gamma \vdash \pi_1(m) \Rightarrow A \mid \Delta \triangleright \pi_1(t)}\ (\times E_1^d)$$

$$\frac{\Gamma \vdash_{\mathrm{NEF}} m \Rightarrow C \mid \Delta \triangleright t \qquad C \to_{whnf} (x : A) \times B}{\Gamma \vdash \pi_2(m) \Rightarrow B[\pi_1(m)/x] \mid \Delta \triangleright \pi_2(t)}\ (\times E_2)$$

$$\frac{\begin{array}{lll}\Gamma \vdash_{\mathrm{NEF}} m \Rightarrow D \mid \Delta \triangleright t & & \Gamma, x_1 : A \vdash n_1 \Rightarrow E[\mathrm{in}_1(n_1)/z] \mid \Delta \triangleright u_1 \\ D \to_{whnf} A + B & \Gamma, z : A + B \vdash C \Rightarrow \mathcal{U}_i \mid \Delta \triangleright E & \Gamma, x_2 : B \vdash n_2 \Rightarrow E[\mathrm{in}_2(n_2)/z] \mid \Delta \triangleright u_2\end{array}}{\Gamma \vdash \mathtt{case}\ m \triangleright z.C\ \mathtt{of}\ (x.n_1 | y.n_2) \Rightarrow C[m/z] \mid \Delta \triangleright \mathtt{case}\ t \triangleright z.E\ \mathtt{of}\ (x_1.u_1 | x_2.u_2)}\ (+E)$$

NEF

$$\frac{m \in \mathrm{NEF} \qquad \Gamma \vdash m \Rightarrow A \mid \Delta \triangleright t}{\Gamma \vdash_{\mathrm{NEF}} m \Rightarrow A \mid \Delta \triangleright t}\ (\mathrm{NEF}I) \qquad \frac{\Gamma \vdash_{\mathrm{NEF}} m \Rightarrow A \mid \Delta \triangleright t}{\Gamma \vdash m \Rightarrow A \mid \Delta \triangleright t}\ (\mathrm{NEF}E)$$

$$\frac{m \in \mathrm{NEF} \qquad \Gamma \vdash m \Leftarrow A \mid \Delta \triangleright t}{\Gamma \vdash_{\mathrm{NEF}} m \Leftarrow A \mid \Delta \triangleright t}\ (\mathrm{NEF}I) \qquad \frac{\Gamma \vdash_{\mathrm{NEF}} m \Leftarrow A \mid \Delta \triangleright t}{\Gamma \vdash m \Leftarrow A \mid \Delta \triangleright t}\ (\mathrm{NEF}E)$$

Control

$$\frac{\Gamma \vdash m \Leftarrow \mathbf{0} \mid \alpha : A, \Delta \triangleright t}{\Gamma \vdash \mu\alpha.m \Leftarrow A \mid \Delta \triangleright \mu\alpha.t}\ (\mu)$$

$$\frac{\Gamma \vdash m \Leftarrow A \mid \Delta \triangleright t}{\Gamma \vdash [\alpha]m \Rightarrow \mathbf{0} \mid \alpha : A, \Delta \triangleright [\alpha]t}\ (name) \qquad \frac{\Gamma \vdash m \Leftarrow \mathbf{0} \mid \Delta \triangleright t}{\Gamma \vdash [top]m \Rightarrow \mathbf{0} \mid \Delta \triangleright [top]t}\ (top)$$

Types

$$\frac{}{\Gamma \vdash \mathbf{1} \Rightarrow \mathcal{U}_i \mid \Delta \triangleright \mathbf{1}}\ (\mathbf{1}) \qquad \frac{}{\Gamma \vdash \langle\rangle \Rightarrow \mathbf{1} \mid \Delta \triangleright \langle\rangle}\ (\text{unit}) \qquad \frac{}{\Gamma \vdash \mathcal{U}_i \Rightarrow \mathcal{U}_{i+1} \mid \Delta \triangleright \mathcal{U}_{i+1}}\ (\mathcal{U}_i)$$

Propositions

$$\frac{}{\Gamma \vdash \mathbb{P} \Rightarrow \mathcal{U}_0 \mid \Delta \triangleright \mathbb{P}}\ (\mathbb{P}) \qquad \frac{\begin{array}{cc} C_1 \to_{whnf} \mathcal{U}_i & C_2 \to_{whnf} \mathbb{P} \\ \Gamma \vdash e_1 \Rightarrow C_1 \mid \Delta \triangleright A & \Gamma, x : A \vdash e_2 \Rightarrow C_2 \mid \Delta \triangleright B \end{array}}{\Gamma \vdash (x : e_1) \to e_2 \Rightarrow \mathbb{P} \mid \Delta \triangleright (x : A) \to B}\ (\Pi_{\mathbb{P}})$$

Equality

$$\frac{\Gamma \vdash p \equiv q \mid \Delta \triangleright t \equiv u}{\Gamma \vdash \mathtt{refl} \Leftarrow p = q \mid \Delta \triangleright \mathtt{refl}_{t=u}}\ (\text{refl})$$

$$\frac{\Gamma \vdash m \Rightarrow p = q \mid \Delta \triangleright t \qquad \Gamma \vdash n \Rightarrow B[p/x] \mid \Delta \triangleright u \qquad \Gamma, x : A \vdash B \Rightarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash \mathtt{subst}\ m\ n \Rightarrow B[q/x] \mid \Delta \triangleright \mathtt{subst}\ t\ u}\ (\text{subst})$$

Bidirectional Subtyping

Subtyping [59]

$$\frac{A \to_{whnf} A' \qquad B \to_{whnf} B' \qquad \Gamma \vdash A' \leqslant: B' \mid \Delta}{\Gamma \vdash A \leqslant B \mid \Delta}$$

Subtyping for Types in WHNF

$$\frac{}{\Gamma \vdash \mathcal{U}_i \leqslant: \mathcal{U}_{i+1} \mid \Delta} \qquad \frac{\Gamma \vdash A_1 \leqslant: A_2 \mid \Delta \qquad \Gamma \vdash B_1 \leqslant: B_2 \mid \Delta}{\Gamma \vdash A_1 + B_1 \leqslant: A_2 + B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \Leftarrow \mathcal{U}_i \mid \Delta \text{ for some } i \qquad \Gamma, x : A_1 \vdash B_1 \leqslant: B_2 \mid \Delta}{\Gamma \vdash (x : A_1) \times B_1 \leqslant: (x : A_2) \times B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \Leftarrow \mathcal{U}_i \mid \Delta \text{ for some } i \qquad \Gamma, x : A_1 \vdash B_1 \leqslant: B_2 \mid \Delta}{\Gamma \vdash (x : A_1) \to B_1 \leqslant: (x : A_2) \to B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A \equiv B \mid \Delta \qquad \Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash B \Leftarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash A \leqslant: B \mid \Delta}$$

B.4 Derivations

$\neg\forall(\mathbf{x} : \mathbf{A}).\mathbf{B} \to \exists(\mathbf{x} : \mathbf{A}).\neg\mathbf{B}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z : B[y/w] \vdash z : B[y/w] \mid \delta : \bot}}{z : B[y/w] \vdash [\beta]z : \bot \mid \delta : \bot, \beta : B[y/w]}}{z : B[y/w] \vdash \mu\delta[\beta]z : \bot \mid \beta : B[y/w]}}{\vdash \lambda z.\mu\delta[\beta]z : \neg B[y/w] \mid \beta : B[y/w]}}{y : A \vdash (y, \lambda z.\mu\delta[\beta]z) : (w : A) \times \neg B \mid \beta : B[y/w]}}{y : A \vdash [\alpha](y, \lambda z.\mu\delta[\beta]z) : \bot \mid \alpha : (w : A) \times \neg B, \beta : B[y/w]}}{y : A \vdash \mu\beta.[\alpha](y, \lambda z.\mu\delta[\beta]z) : B[y/w] \mid \alpha : (w : A) \times \neg B}}{\vdash \lambda y.\mu\beta.[\alpha](y, \lambda z.\mu\delta[\beta]z) : (w : A) \to B \mid \alpha : (w : A) \times \neg B}}{x : \neg((w : A) \to B) \vdash x(\lambda y.\mu\beta.[\alpha](y, \lambda z.\mu\delta[\beta]z)) : \bot \mid \alpha : (w : A) \times \neg B}}{x : \neg((w : A) \to B) \vdash [top]x(\lambda y.\mu\beta.[\alpha](y, \lambda z.\mu\delta[\beta]z)) : \bot \mid \alpha : (w : A) \times \neg B}}{x : \neg((w : A) \to B) \vdash \mu\alpha[top]x(\lambda y.\mu\beta.[\alpha](y, \lambda z.\mu\delta[\beta]z)) : (w : A) \times \neg B \mid \alpha : (w : A) \times \neg B}}{\vdash \lambda x.\mu\alpha[top]x(\lambda y.\mu\beta.[\alpha](y, \lambda z.\mu\delta[\beta]z)) : \neg((w : A) \to B) \to (w : A) \times \neg B}$$

A proof of $\neg\forall(\mathbf{x} : \mathbf{A}).\mathbf{B} \to \exists(\mathbf{x} : \mathbf{A}).\neg\mathbf{B}$ in a $\nu\lambda\mu$-style calculus with dependent types.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z : A \vdash z : A} \quad \dfrac{}{a : \neg B[z/w] \vdash a : \neg B[z/w]}}{a : \neg B[z/w] \vdash (z, a) : (w : A) \times \neg B} \quad \dfrac{}{x : \neg(w : A) \times \neg B \vdash x : \neg(w : A) \times \neg B}}{a : \neg B[z/w], x : \neg(w : A) \times \neg B \vdash [(z, a)]x : \bot}}{x : \neg(w : A) \times \neg B, z : A \vdash \mu a.[(z, a)]xB[z/w]}}{x : \neg(w : A) \times \neg B \vdash \lambda z.\mu a.[(z, a)]x(w : A) \to B} \quad \dfrac{}{y : \neg(w : A) \to B \vdash y : \neg(w : A) \to B}}{x : \neg(w : A) \to B, y : \neg(w : A) \times \neg B \vdash [y](\lambda z.\mu a.[(z, a)]x) : \bot}}{x : \neg(w : A) \to B \vdash \mu y.[y](\lambda z.\mu a.[(z, a)]x)(w : A) \times \neg B}}{\vdash \lambda x\mu y.[y](\lambda z.\mu a.[(z, a)]x) : \neg(w : A) \to \neg B \to (w : A) \times \neg B}$$

B.5 Implementation

B.5.1 Syntax

Dependently Typed Theorem Prover Syntax

$$
\begin{array}{rcl}
\langle name \rangle & ::= & [\text{unicode letters}] \\
\langle var \rangle & ::= & \text{`\_'} \mid \langle name \rangle \\
\langle term \rangle, \langle type \rangle & ::= & \langle name \rangle \\
& \mid & \backslash(\langle var \rangle +) \to \langle term \rangle \\
& \mid & \backslash(\langle var \rangle : \langle type \rangle) \to \langle term \rangle \\
& \mid & \langle term \rangle + \\
& \mid & \backslash \langle var \rangle : \langle var \rangle \backslash \ \langle term \rangle \\
& \mid & \texttt{in}(1|2)\ \langle term \rangle \\
& \mid & \texttt{case-or}\ \langle term \rangle\ \texttt{of}\ (\langle term \rangle | \langle term \rangle) \\
& \mid & (\langle term \rangle, \langle term \rangle) \\
& \mid & \texttt{proj}(1|2)\ \langle term \rangle \\
& \mid & (\langle term \rangle) \\
& \mid & () \\
& \mid & ?[0-9]+ \\
& \mid & \texttt{Top} \\
& \mid & \texttt{Bot} \\
& \mid & \langle type \rangle \to \langle type \rangle \\
& \mid & \langle telescope \rangle \to \langle type \rangle \\
& \mid & \langle type \rangle\ \text{`+'}\ \langle type \rangle \\
& \mid & \langle type \rangle\ \text{`*''}\ \langle type \rangle \\
& \mid & (\langle name \rangle : \langle type \rangle)\ \text{`*'}\ \langle type \rangle \\
& \mid & (\langle type \rangle) \\
& \mid & \texttt{case}\ \langle term \rangle\ \texttt{of}\ \langle pattTree \rangle \\
& \mid & \texttt{elim}\ \langle term \rangle\ \texttt{by}\ \langle pattTree \rangle \\
& \mid & \texttt{build}\ \langle pattTree \rangle \\
& \mid & \texttt{Univ} \mid \texttt{Prop} \\
& \mid & \texttt{refl} \\
& \mid & \texttt{subst}\ \langle term \rangle\ \langle term \rangle \\
& \mid & \langle term \rangle = \langle term \rangle \\
\langle pattTree \rangle & ::= & (\langle var \rangle + \to \langle term \rangle)* \\
\langle decl \rangle & ::= & \langle name \rangle\ :\ \langle type \rangle \\
& \mid & \langle name \rangle\ =\ \langle term \rangle \\
& \mid & \texttt{variable}\langle name \rangle : \langle type \rangle \\
& \mid & \texttt{data}\ \langle name \rangle\ \langle telescope \rangle : \langle telescope \rangle\ \texttt{where} \\
& & \quad (\langle name \rangle\ :\ \langle telescope \rangle \to \langle type \rangle)* \\
& \mid & \texttt{record}\ \langle name \rangle\ \langle telescope \rangle : \langle telescope \rangle\ \texttt{where} \\
& & \quad (\langle name \rangle\ :\ \langle type \rangle)* \\
\langle telescope \rangle & ::= & (\langle name \rangle + : \langle type \rangle) \langle telescope \rangle \\
& \mid & \langle type \rangle\ \langle telescope \rangle \\
& \mid & \epsilon
\end{array}
$$

References

[1] Abel A and al. et. *BNFC: A compiler front-end generator*. Hackage. Version 2.9.2. 2021. URL: https://hackage.haskell.org/package/BNFC.

[2] Ariola ZM and Herbelin H. "Minimal classical logic and control operators". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2003, pp. 871–885.

[3] Ariola ZM, Herbelin H, and Sabry A. "A proof-theoretic foundation of abortive continuations". In: *Higher-order and symbolic computation* 20.4 (2007), pp. 403–429.

[4] Asperti A, Ricciotti W, Coen CS, and Tassi E. "A compact kernel for the calculus of inductive constructions". In: *Sadhana* 34.1 (2009), pp. 71–144.

[5] Audebaud P and Bakel S van. "A completeness result for $\lambda\mu$". In: *preprint* (Feb. 2007).

[6] Bakel S van. "Exception Handling and Classical Logic". In: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 2019, pp. 1–14.

[7] Bakel S van. "Sound and complete typing for lambda-mu". In: *arXiv preprint arXiv:1101.4425* (2011).

[8] Bakel S van. *Type Systems for Programming Languages*. online. 2019. URL: http://www.doc.ic.ac.uk/~svb/TSfPL.

[9] Bakel S van, Barbanera F, and de'Liguoro U. "Characterisation of strongly normalising lambda-mu-terms". In: *arXiv preprint arXiv:1307.8202* (2013).

[10] Bakel S van and Vigliotti MG. "A fully-abstract semantics of lambda-mu in the pi-calculus". In: *arXiv preprint arXiv:1409.3314* (2014).

[11] Barendregt HP et al. *The Lambda Calculus: its syntax and semantics, volume 103 of Studies in Logic*. 1984.

[12] Barendregt, H.P. "Lambda calculi with types". null. In: *S. Abramsky; D.M. Gabbai; T.S.E. Maibaum (eds.), Handbook of logic in computer science; vol. 2*. Oxford : Clarendon Press, 1992, pp. 117–309. URL: http://hdl.handle.net/2066/17231 (visited on 03/04/2021).

[13] Blazy S, Dargaye Z, and Leroy X. "Formal Verification of a C Compiler Front-End". In: *FM 2006: Formal Methods*. Ed. by Misra J, Nipkow T, and Sekerinski E. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 460–475. ISBN: 978-3-540-37216-5.

[14] Brady EC. "Practical Implementation of a Dependently Typed Functional Programming Language". PhD thesis. Durham University, 2005.

[15] Bruijn NG de. "Telescopic mappings in typed lambda calculus". In: *Information and Computation* 91.2 (1991), p. 194.

[16] Carr A. *The Natural Deduction Pack*. Online. 2018. URL: https://logicmanual.philosophy.ox.ac.uk/carr/NDpack.pdf.

[17] Chris Dornan SM. *alex: Alex is a tool for generating lexical analysers in Haskell*. Hackage. Version 3.2.6. 2020. URL: https://hackage.haskell.org/package/alex.

[18] Church A. "A Set of Postulates For the Foundation of Logic". eng. In: *Annals of mathematics* 34.4 (1933), pp. 839–864. ISSN: 0003-486X.

[19] Coquand T. "The paradox of trees in type theory". In: *BIT Numerical Mathematics* 32.1 (1992), pp. 10–14.

[20] Curien PL and Herbelin H. "The duality of computation". In: *ACM sigplan notices* 35.9 (2000), pp. 233–243.

[21] Daan Leijen Paolo Martin AL. *parsec: Monadic parser combinators*. Hackage. Version 3.1.14.0. 2021. URL: https://hackage.haskell.org/package/parsec.

[22] Daan Leijen Paolo Martin MK. *megaparsec: Monadic parser combinators*. Hackage. Version 9.0.1. 2020. URL: https://hackage.haskell.org/package/megaparsec.

[23] Dalen D van. *Logic and structure*. Vol. 3. Springer, 1994.

[24] Damas L. "Type assignment in programming languages". In: *KB thesis scanning project 2015* (1984).

[25] David R and Nour K. "Arithmetical proofs of strong normalization results for symmetric lambda calculi". In: *arXiv preprint arXiv:0905.0762* (2009).

[26] De Bruijn NG. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[27] De Groote P. "On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 1994, pp. 31–43.

[28] Downen P and Ariola ZM. "A tutorial on computational classical logic and the sequent calculus." In: *J. Funct. Program.* 28 (2018), e3.

[29] Dybjer P. "Inductive families". In: *Formal aspects of computing* 6.4 (1994), pp. 440–465.

[30] Felleisen M and Friedman DP. *Control Operators, the SECD-machine, and the [1]-calculus*. Indiana University, Computer Science Department: Indiana University, Computer Science Department, 1986.

[31] Gentzen G. "Investigations into Logical Deduction". In: *American Philosophical Quarterly* 1.4 (1964), pp. 288–306. ISSN: 00030481. URL: http://www.jstor.org/stable/20009142.

[32] Gill A and Marlow S. *happy: Happy is a parser generator for Haskell*. Hackage. Version 1.20.0. 2020. URL: https://hackage.haskell.org/package/happy.

[33] Griffin TG. "A formulae-as-type notion of control". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 47–58.

[34] Groote P de. "Strong normalization of classical natural deduction with disjunction". In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2001, pp. 182–196.

[35] Gundry A and McBride C. "A tutorial implementation of dynamic pattern unification". In: *Unpublished draft* (2013).

[36] Harper R. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[37] Harper R and Pollack R. "Type checking with universes". In: *Theoretical computer science* 89.1 (1991), pp. 107–136.

[38] Herbelin H. "A constructive proof of dependent choice, compatible with classical logic". In: *2012 27th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2012, pp. 365–374.

[39] Herbelin H. "On the degeneracy of Σ-types in presence of computational classical logic". In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2005, pp. 209–220.

[40] Hindley R. "An Abstract form of the church-rosser theorem. I". In: *Journal of Symbolic Logic* 34.4 (1969), pp. 545–560. DOI: 10.1017/S0022481200128439.

[41] Howard WA. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[42] Kliger A. *unbound-generics: Support for programming with names and binders using GHC Generics*. Hackage. Version 0.4.1. 2020. URL: https://hackage.haskell.org/package/unbound-generics.

[43] Lepigre R. "A classical realizability model for a semantical value restriction". In: *European Symposium on Programming*. Springer. 2016, pp. 476–502.

[44] Löh A, McBride C, and Swierstra W. "A tutorial implementation of a dependently typed lambda calculus". In: *Fundamenta informaticae* 102.2 (2010), pp. 177–207.

[45] Luo Z. "An extended calculus of constructions". PhD thesis. University of Edinburgh, 1990.

[46] Martin-Löf P and Sambin G. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

[47] Mazzoli F. *Bertus: Implementing Observational Equality*. Master's Project. 2013.

[48] McBride C. "Elimination with a motive". In: *International Workshop on Types for Proofs and Programs*. Springer. 2000, pp. 197–216.

[49] Miller D. "Unification under a mixed prefix". In: *Journal of symbolic computation* 14.4 (1992), pp. 321–358.

[50] Milner R. "A theory of type polymorphism in programming". In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[51] Miquey É. "A sequent calculus with dependent types for classical arithmetic". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018, pp. 720–729.

[52] Miquey É. "Classical realizability and side-effects". PhD thesis. Université Sorbonne Paris Cité-Université Paris Diderot (Paris 7 . . ., 2017.

[53] Miquey É, Montillet X, and Munch-Maccagnoni G. "Dependent Type Theory in Polarised Sequent Calculus". Draft. 2020.

[54] Miquey É, Montillet X, and Munch-Maccagnoni G. "Dependent Type Theory in Polarised Sequent Calculus". In: *TYPES 2020-26th International Conference on Types for Proofs and Programs*. abstract. 2020.

[55] Miquey É, Montillet X, and Munch-Maccagnoni G. *Dependent Type Theory in Polarised Sequent Calculus*. 2020. URL: https://perso.ens-lyon.fr/etienne.miquey/content/ldep-talk.pdf.

[56] MIT. *Scheme*. online. May 2021. URL: https://groups.csail.mit.edu/mac/projects/scheme/.

[57] Negri S, Von Plato J, and Ranta A. *Structural proof theory*. Cambridge University Press, 2008.

[58] Nordström B, Petersson K, and Smith JM. "Martin-Löf's type theory". In: *Handbook of logic in computer science* 5 (2000), pp. 1–37.

[59] Norell U. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer, 2007.

[60] Norell U and al. et. *The Agda Wiki*. online. Feb. 2021. URL: https://wiki.portal.chalmers.se/agda/Main/HomePage.

[61] Parigot M. "$\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. Berlin, Heidelberg, 1992, pp. 190–201.

[62] Parigot M. "Classical proofs as programs". In: *Kurt Gödel Colloquium on Computational Logic and Proof Theory*. Springer. 1993, pp. 263–276.

[63] Peyton Jones SL. *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[64] Pierce BC and Benjamin C. *Types and programming languages*. MIT press, 2002.

[65] Plotkin GD. "Call-by-name, call-by-value and the λ-calculus". In: *Theoretical computer science* 1.2 (1975), pp. 125–159.

[66] Robinson JA. "A Machine-Oriented Logic Based on the Resolution Principle". In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41.

[67] Schroer DE. *The Church-Rosser Theorem*. Vol. 2. Cornell Univ., June, 1965.

[68] Sjöberg V, Casinghino C, Ahn KY, Collins N, Eades III HD, Fu P, et al. "Irrelevance, heterogeneous equality, and call-by-value dependent type systems". In: *arXiv preprint arXiv:1202.2923* (2012).

[69] Sørensen MH and Urzyczyn P. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[70] Summers AJ. "Curry-howard term calculi for gentzen-style classical logics". PhD thesis. Citeseer, 2008.

[71] Swaen MDG. "The Logic of First Order Intuitionistic Type Theory with Weak Sigma- Elimination". In: *The Journal of Symbolic Logic* 56.2 (1991), pp. 467–483. ISSN: 00224812. URL: http://www.jstor.org/stable/2274694.

[72] Team TCD. *The Coq Reference Manual, Release 8.13.0*. 2021. URL: https://github.com/coq/coq/releases/download/V8.13.0/coq-8.13.0-reference-manual.pdf.

[73] Turing AM. "Computability and lambda-definability". In: *Journal of Symbolic Logic* 2.4 (1937), pp. 153–163. DOI: 10.2307/2268280.

[74] Turing AM. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[75] Univalent Foundations Program T. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[76] Wadler P. "Call-by-value is dual to call-by-name". In: *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. 2003, pp. 189–201.

[77] Wadler P. "Propositions as types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84.

[78] Weirich S. *Designing Dependently-Typed Programming Languages*. https://www.cs.uoregon.edu/research/summerschool/summer14/curriculum.html. Online lecture. 2014.

[79] Weirich S. *Pi-Forall*. https://github.com/sweirich/pi-forall. 2020.

[80] Weirich S, Yorgey BA, and Sheard T. "Binders Unbound". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 333–345. ISSN: 0362-1340. DOI: 10.1145/2034574.2034818. URL: https://doi.org/10.1145/2034574.2034818.

[81] Wells JB. "The Essence of Principal Typings". In: *Automata, Languages and Programming*. Ed. by Widmayer P, Eidenbenz S, Triguero F, Morales R, Conejo R, and Hennessy M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 913–925. ISBN: 978-3-540-45465-6.