Candid

A Dependently Typed Programming Language with Control Operators for Classical Logic

David Davies and Steffen van Bakel and Nicolas Wu

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK

Abstract

The Curry-Howard correspondence has shown us that mathematics and programming can be seen as the same activity. This correspondence is often approached using intuitionistic logic to give a theoretical grounding to programming languages. However, intuitionistic logic is not the only way to understand this correspondence, as classical logic has computational interpretations through control operators, whilst also allowing proofs involving classical concepts like Proof by Contradiction. This paper describes a dependently typed classical calculus and its implementation as a programming language, Candid.
When adding dependent types to a classical calculus, restrictions must be made on their interactions with the control operators to maintain consistency. We extend a dependent classical calculus with coproducts, inductive data and codata, and extend the restrictions for dependent types and control operators to these new structures. We then implement this calculus as the core of a dependently typed theorem prover for classical logic, able to express proofs of uniquely classical propositions as runnable programs.

keywords: Classical Logic, Dependent Types, Theorem Provers

1 Introduction

Proof assistants are programming languages that correspond with a formal logic, and come in different shapes and forms; Coq [12] has a particular focus on the theorem proving aspect where proofs can be written with intuitive tactics, whereas Agda [33] and Idris [7] are more deeply connected to functional programming languages 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 [27]. They all capitalise on the Curry-Howard correspondence [21], the fact that functions in a functional programming language correspond to proofs in intuitionistic logic [8, 9, 10], 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 [44].

Intuitionism inescapably limits 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 is the distinguishing feature of Classical Logic [15]. 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 from a proof of B; it is not enough to prove it cannot be the case that 'not (A or B)' is not true. This notion of constructive logic and mathematics is strongly tied with computability, and until fairly recently it was believed that a proof had a correspondence with a function only if the proof is constructive, and that classical logic did not have a computational counterpart.

This belief was challenged when [16] discovered that a control operator, similar to the call/cc function in Scheme, corresponded directly with an axiom of classical logic, namely that of 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, that of calculi such as the $\lambda\mu$ calculus that correspond with classical logic [36, 42, 13, 4]. It is natural to investigate the possibility of defining a theorem prover for Classical Logic based on one of these calculi, and our choice at this time is to depart with $\lambda\mu$.

In this paper we show that in terms of implementability, expressiveness, and elegance proof assistants based on Classical Logic have much to contribute. 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. Our attempts proved successful and we present Candid, a theorem prover bases on $\lambda\mu$, but enriched with dependent types, as an extension of $\mathrm{ECC}_{\mathrm{K}}$ [31] adding coproducts and dependent algebraic data types. After some startling examples of the capabilities of Candid in Sect. 2, we start our investigation in this paper in Sect. 3, where we focus on the system of classical natural deduction (CND) we explore here, one that uses the logical connectors *implication*, *negation*, *conjunction*, and *disjunction*, and discuss issues that arise with the definition of proof contraction. This is followed in Sect. 4, where we briefly repeat the definition of Parigot's $\lambda\mu$. We will present the underlying logic with focus for that system, discuss how it relates to CND, and how the problem of proof contraction in CND is solved.

As we will see in Sect. 5, the link between first order classical logic and computation is a bit more tricky. First order intuitionistic logic is known to be achieved by dependently typed systems (Sect. 5.1), but [19] showed that naively combining dependent types and control operators allows one to prove that all types have only one inhabitant (Sect. 5.2). Fortunately, not all is lost, as [20] also shows a way to restrict how dependent types and control operators can interact, which regains a logically consistent type theory (Sect. 5.3). A more general dependent classical system, $\mathrm{ECC}_{\mathrm{K}}$ (Sect. 5.4) is proposed by [31], as part of ongoing work in defining type-preserving CPS for dependently typed systems. $\mathrm{ECC}_{\mathrm{K}}$ is, roughly speaking, the Extended Calculus of Constructions [26] plus the control operators catch and throw. Although a powerful calculus, it is not strong enough to represent proofs in classical logic.

This paper introduces the calculus ECC_{μ}, which expands on $\mathrm{ECC}_{\mathrm{K}}$; first by adding coproducts with dependent elimination (Sect. 6), and then generalising coproducts and products to inductive data and codata (Sect. 8), all while making sure that these new constructs are able to interact safely with the control operators. ECC_{μ} is used as the core of a dependently typed theorem prover, Candid. We introduce the notions of weak head normal forms and a type checking algorithm (Sect. 7) for ECC_{μ} that provide theoretical foundations of the language, which is implemented as a functional programming language (Sect. 9).

The contributions of this paper are as follows:

- We define the dependently typed classical calculus ECC_{μ}, which extends $\mathrm{ECC}_{\mathrm{K}}$ by adding coproducts with dependent elimination (Sect. 6).
- We introduce a bidirectional typing algorithm for ECC_{μ}, including novel rules for typing μ and $[\cdot]$ terms (Sect. 7).
- We expand ECC_{μ} with inductive data and codata, maintaining consistency by generalising the restrictions of how control operators can interact with products and coproducts (Sect. 8).
- We outline the implementation of Candid: a dependently typed programming language with control operators, data and codata, based on ECC_{μ} that functions as a proof assistant for classical logic (Sect. 9).

Finally, we discuss related work (Sect. 10) and conclude (Sect. 11).

2 Working with Candid Proofs

This section gives some examples of uniquely classical propositions and their *computable* proofs.

To express double negation elimination, a new operator, μ, is added to the λ-calculus. Due to the Curry-Howard correspondence, μ can be understood from both a logical and a computational perspective. Where $\lambda x.M$ means to index an assumption by x, $\mu\alpha.M$ means to index a *negated* assumption by α; for example, $\mu\alpha.M : A$ means that α indexes the assumption $\neg A$; and then M refutes this assumption and allows to derive absurdity, $\bot$; the prefix μ then embodies the 'proof by contradiction' step. So essentially, departing from $\Gamma, x{:}\neg A \vdash M : \bot$, it is possible to derive both $\Gamma \vdash \lambda x.M : \neg\neg A$ and $\Gamma \vdash \mu x.M : A$, highlighting the difference between the intuitionistic logic of the pure λ-calculus, and the classical reasoning μ gives.

To form the contradiction needed in these proofs, there is the syntax $[\beta]M$, which means that $M : B$ directly contradicts $\beta : \neg B$. The new term constructs are kept together, and used to create terms of the form $\mu\alpha.[\beta]M$. In this theorem prover, $\mu\alpha.[\beta]M$ is written as `\\'a.['b]M` (where, using the `qsymbols` notation, `'a` is taken to mean α, `'b` β, and so on).

Example 2.1 (Law of Excluded Middle) The *law of the excluded middle* (lem) captures exactly the notion that in Classical Logic a proposition is either true or false. In standard theorem provers it is added as an axiom, but it is provable in Candid.

```
lem : (A : Type) -> (A + !A)
lem _ = \\'a.['a] inj2 (\x -> \\_.['a] inj1 x)
```

The proof reads: assume there is a proof `'a` showing $A + \neg A$ is not true. We seek to contradict this by proving $A + \neg A$, and comparing it with `['a]`. This is achieved by assuming `x` is a proof of A; then by `inj1 x` follows $(A + \neg A)$ which contradicts the assumption `'a` of $\neg(A + \neg A)$. Thus, by proof by contradiction (PbC) on `'a`, we have $(A + \neg A)$. Notice that lem holds for any type A; we are not restricted to, say, the impredicative type of propositions $\mathbb{P}$. This separates Candid from proof assistants like Agda and Coq, in which lem is not provable in general.

Example 2.2 (Double Negation Elimination) Also the strongest classical axiom [2], $\neg\neg A \to A$ (*DNE*) can be understood through as: if $\neg A$ is not true, A is. Again `\\` and `[·]` are used:

```
dne : (A : Type) -> !!A -> A
dne A nna = \\'a.[_] nna (\x -> \\_.['a] x)
```

This term can be read similarly to the previous example. We assume $\neg\neg A$, and then assume a proof `'a` of $\neg A$, which immediately causes a contradiction. However, the contradiction is with a normal λ-variable `nna` (not a special assumption `'g` from `\\'g`), so we compare this contradiction with the tautology $\neg\bot$ (represented by `[_]`), which gives a second (trivial) contradiction.

This can be used in many ways; in general, we could just apply it to intuitionistic proofs of $\neg\neg A$ to get A, in some sense performing a reverse double negation translation. For example, take an intuitionistic proof of $\neg\neg(A \vee \neg A)$:

```
nnlem : (A : Type) -> !!((A + !A)
nnlem _ x = x (inj2 (\y -> (x inj1 y)))
```

In intuitionistic logic, this is all that would be achievable. But in classical logic, it is possible to eliminate these negations, by applying double negation elimination to `nnlem`:

```
lem2 : (A : Type) -> (A + !A)
lem2 A = dne (A+!A) (nnlem A)
```

Candid can normalise this term (in the command line REPL):

```
> lem2
\\'a.['a] inj2 (\y -> \\_.['a] inj1 y)
```

Note that this is the first proof of LEM; we took a proof of a double negation translated proposition and turned it into a proof of the original proposition. REPL *evaluated* the proof of `lem2`; this illustrates that the system can normalise proofs in classical logic, yielding computational classical logic.

Example 2.3 (CLASSICAL EXISTENTIAL) Candid is able to classically reason about quantified statements as well. In intuitionistic logic, proofs of $\exists n.P(n)$ are constructive and provide the witness. In classical logic, this might not be the case: the proof could come from a refutation of its negation.

Candid is able to reason about such non-constructive existentials. Interestingly, the refutation will still involve creating a pair (that might consist of the continuation variables).

Our goal is to prove a tautology of classical logic: $\exists n.B(n) \leftrightarrow \neg(\forall n.\neg B(n))$. The implication left to right is provable in intuitionistic logic, with a proof given by `Qinv`:

```
Qinv : (A : Type)
       -> (B : A -> Type)
       -> (n:A)* (B n)
       -> !((n:A) -> !(B n))
Qinv A B ex f = f (proj1 ex) (proj2 ex)
```

What Candid is able to prove, unlike other proof assistants, is $\neg(\forall n.\neg B(n)) \rightarrow \exists n.B(n)$. To show how, we first define a lemma, `P` $:= \neg(\exists n.B(n)) \rightarrow \forall n.\neg B(n)$, that is proved by:

```
P : (A : Type)
    -> (B : A -> Type)
    -> !((n:A) * B n)
    -> ((n:A) -> !(B n))
P _ _ nex = \n b_n -> nex (n, b_n)
```

Note that the proof of `P` is intuitionistic and, thus, makes no use of the classical operator `\\·.[·]`.

The classical proof of $\neg(\forall n.\neg B(n)) \rightarrow \exists n.B(n)$ is expressed by `Q`, which uses lemma `P`:

```
Q : (A : Type)
    -> (B : A -> Type)
    -> !((n:A) -> !(B n))
    -> ((n:A) * B n)
Q A B nfa = \\nex.[_] nfa (P A B (\x -> \\_.[nex] x))
```

Again, this proof can be understood intuitively. We assume a proof nfa of $\neg(\forall n.\neg B(n))$. Then, for the sake of argument, let us assume $\exists n.B(n)$ is not true – this is expressed by `\\nex`. By the lemma `P` we know we can convert our proof `nex` of $\neg\exists n.B(n)$ into a proof of $\forall n.\neg B(n)$ – this contradicts nfa. Thus, by *PbC*, it must be the case that $\exists n.B(n)$ is true. Finally, to form the logical equivalence $\exists n.B(n) \leftrightarrow \neg(\forall n.\neg B(n))$, we can form a pair of the proofs of each implication, as $A \leftrightarrow B$ ('A if and only if B') is encoded by a pair of proofs of $A \rightarrow B$ and $A \leftarrow B$:

```
T : (A : Type)
    -> (B : (A -> Type))
    -> (((n:A) * (B n)) <-> !((n:A) -> !(B n)))
T A B = (Qinv A B, Q A B)
```

3 Classical Logic

Natural Deduction for Classical Logic, defined by [15] is a way of describing the structure of formal proofs in mathematics that follow the intuitive, human, lines of reasoning as much as possible. Natural deduction deals with statements of the shape $A_1,\ldots,A_n \vdash B$, where all the formulas on the left of the turnstyle are assumed to hold, and the one on the right is shown to follow from those assumption, using reasoning steps that are formalised through inference rules that introduce or eliminate connectors in the logical formulas. In that paper, Gentzen also presents the Sequent Calculus, which differs from Natural Deduction in that it derives sequences of the shape $A_1,\ldots,A_n \vdash B_1,\ldots,B_m$ with the intended meaning 'if all of the properties $A_1, \ldots, A_n$ hold, then at least one of the $B_1, \ldots, B_m$ does as well.'

In the Sequent Calculus, for each connector, there is a *left* and *right* introduction rule; there are no elimination rules for connectors, just a generic (cut)-rule that eliminates a formula.

To better be able to reason about the representation of the structure of proofs and the technicalities of proof contraction, it is necessary to represent the structure of proofs via term information from an appropriate calculus, and inhabit the inference rules with terms, such that proof contractions correspond to term reduction. This employs the Curry-Howard principle, which expresses a correspondence between terms and their types on one side, and proofs for propositions on the other. There are many systems already proposed for term calculi that represent proofs in classical logic. Four directions are that of $\lambda\mu$-calculus defined by [36], the $\nu\lambda\mu$-calculus by [42] that adds negation to $\lambda\mu$ (representing $(\neg I)$ through $\nu x.M$, and $(\neg E)$ through $[M]N$), the $\overline{\lambda}\mu\tilde{\mu}$-calculus defined by [13], and the calculus $\mathcal{X}$ defined by [4]. The latter two deal with representations of the Sequent Calculus, that is perhaps less suitable for a human-operated proof-tool. For reasons of simplicity, this paper starts with the exploration of the first.

The variant of Classical Natural Deduction that will be considered in this paper uses the logical connectors $\rightarrow$ (*implication*), $\neg$ (*negation*), $\wedge$ (*conjunction*) and $\vee$ (*disjunction*).

Definition 3.1 (Classical Natural Deduction) The formulas for Classical Natural Deduction are:

$$A,B ::= \varphi \mid A{\rightarrow}B \mid \neg A \mid A \wedge B \mid A \vee B$$

A context Γ is a multi-set of formulas, where $\Gamma, A = \Gamma \uplus \{A\}$ and $\vdash_{\text{NI}}$ is defined through the inference rules (using a sequent notation):

$$\dfrac{}{\Gamma,A \vdash A}\,(Ax) \quad \dfrac{\Gamma,A \vdash B}{\Gamma \vdash A{\rightarrow}B}\,({\rightarrow}I) \quad \dfrac{\Gamma \vdash A{\rightarrow}B \quad \Gamma \vdash A}{\Gamma \vdash B}\,({\rightarrow}E) \quad \dfrac{\Gamma,A \vdash \bot}{\Gamma \vdash \neg A}\,(\neg I) \quad \dfrac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}\,(\neg E) \quad \dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\,(EFQ)$$

$$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge I) \quad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \; \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,(\wedge E) \quad \dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \; \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,(\vee I) \quad \dfrac{\Gamma \vdash A \vee B \quad \Gamma,A \vdash C \quad \Gamma,B \vdash C}{\Gamma \vdash C}\,(\vee E)$$

$$\dfrac{\Gamma,\neg A \vdash \bot}{\Gamma \vdash A}\,(PbC) \quad \dfrac{\Gamma \vdash A\{a/x\}}{\Gamma \vdash \forall x.A}\,(\forall I) \quad \dfrac{\Gamma \vdash \forall x.A}{\Gamma \vdash A\{a/x\}}\,(\forall E) \quad \dfrac{\Gamma \vdash A\{a/x\}}{\Gamma \vdash \exists x.A}\,(\exists I) \quad \dfrac{\Gamma \vdash \exists x.A \quad \Gamma, A\{a/x\} \vdash B}{\Gamma \vdash B}\,(\exists E)$$

Notice that, in this presentation, $\bot$ is not a formula. Omitting rule (DNE) would give a definition for intuitionistic logic. Moreover, in $\vdash_{\text{NI}}$, rule (EFQ) is a derivable by weakening and (PbC).

Negation comes of course with introduction and elimination rules, but plays a more intricate role in Classical Logic, in that the *law of excluded middle* or something similar holds. There are many rules that express this to a different degree, like:

$$\dfrac{\Gamma,\neg A \vdash \bot}{\Gamma \vdash A}\,(PbC) \quad \dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,(DNE) \quad \dfrac{}{\vdash A \vee \neg A}\,(LEM) \quad \dfrac{}{\vdash ((A{\rightarrow}B){\rightarrow}A){\rightarrow}A}\,(PL) \quad \dfrac{}{\vdash (\neg A{\rightarrow}A){\rightarrow}A}\,(RAA)$$

(called *'proof by contradiction'*, *'double negation elimination'*, *'law of excluded middle'*, *'Peirce's law'*, and *reductio ad absurdum*, respectively.) These rules have different expressive power, and adding one rather than another changes the set of provable formula (see [1]).

For the Sequent Calculus, Gentzen defines a notion of (proof) contraction that removes occurrences of (cut), and shows that this is terminating: for every proof that shows $\Gamma \vdash \Delta$, there exists a (cut)-free proof that shows the same result. He does not show that result for Natural Deduction, which would eliminate all introduction-elimination pairs, and we can guess that that is because this notion of contraction is harder to define.

The main difficulty is that in Natural Deduction, not all proof-constructions follow the introduction-elimination pattern of the inference rules; above, in particular, that is the rule (PbC).

For those that do, proof contraction consists of elimination an introduction step followed by an elimination step for the same logical connector; for implication that looks like in the diagram (where $\Rightarrow$ stands for proof contraction). Notice that, in the rule $(\to I)$, the formula A ceases to be an assumption, and that, in the composed proof on the right, A is no longer an assumption needed to reach the conclusion, since it has been shown to hold by D_2. This is not always possible for negation.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{\Gamma,A\vdash A}\,(Ax)\\ \boxed{D_1}\\ \Gamma,A\vdash B\end{array}}{\Gamma\vdash A\to B}\,(\to I)\qquad \begin{array}{c}\boxed{D_2}\\ \Gamma\vdash A\end{array}}{\Gamma\vdash B}\,(\to E) \quad\Rightarrow\quad \begin{array}{c}\boxed{D_2}\\ \Gamma\vdash A\\ \boxed{D_1}\\ \Gamma\vdash B\end{array}$$

Example 3.2 Take the following proof:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\boxed{D_1}\\ \Gamma,\neg(A\to B)\vdash\neg\neg(A\to B)\end{array}\qquad \dfrac{}{\neg(A\to B)\vdash\neg(A\to B)}\,(Ax)}{\Gamma,\neg(A\to B)\vdash\bot}\,(\neg E)}{\Gamma\vdash A\to B}\,(PbC)\qquad \begin{array}{c}\boxed{D_2}\\ \Gamma\vdash A\end{array}}{\Gamma\vdash B}\,(\to E)$$

It is *a priori* not obvious how to contract this proof; it is normal practice for the sub-derivations to be the building stones for the proof for $\Gamma \vdash B$ without the (PbC)-$(\to E)$ pair, but it is not immediately clear how to do that.

Notice that there is no sub-derivation above the step (PbC) that has A as an assumption (so does not contain $\Gamma, A \vdash A$ as the result of rule (Ax)), nor one that derives $\Gamma \vdash A \to B$.

Below will be shown how $\lambda\mu$ deals with this problem.

This paper will present Candid, a kernel programming language equipped with a notion of dependent types that will allow for the inhabitation of proofs for derivable statements in $\vdash_{\text{NI}}$ as a variant of $\lambda\mu$ (see next section), extended with constructs to represent pairing, choice, and dependent types. Negation will be represented by adding $\bot$ as a psuedo-formula in that it will only be used to represent negation, and only 'under the hood'; so $(\to I)$ and $(\to E)$ will be used for $(\neg I)$ and $(\neg E)$. The system is complete, in that all judgements provable in $\vdash_{\text{NI}}$ can be proven in Candid, but it is not the case that all proofs in $\vdash_{\text{NI}}$ can be inhabited directly in Candid.

4 $\lambda\mu$-Calculus

Parigot's $\lambda\mu$-calculus is a proof-term syntax for classical logic, expressed in Natural Deduction, defined as an extension of the Curry type assignment system for the λ-calculus. With $\lambda\mu$ Parigot created a multi-conclusion typing system which corresponds to classical logic with *focus*; there derivable statements have the shape $\Gamma \vdash A \mid \Delta$, where A is the main conclusion of the statement, expressed as *active*, Γ is the set of assumptions and Δ is the set of alternative conclusions, or have the shape $\Gamma \vdash \bot \mid \Delta$ if there is no formula under focus.

$\lambda\mu$'s underlying logic corresponds to the following:

Definition 4.1 (A CLASSICAL LOGIC WITH FOCUS) The formulas for this system are:

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

and a context Γ is a set of formulas, where $\Gamma, A = \Gamma \cup \{A\}$; the inference rules are defined by:

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

$$(Act): \frac{\Gamma \vdash \bot \mid A, \Delta}{\Gamma \vdash A \mid \Delta} \qquad (Pass): \frac{\Gamma \vdash A \mid A, \Delta}{\Gamma \vdash \bot \mid \Delta}$$

$\Gamma \vdash_{\mathrm{F}} M : A \mid \Delta$ is used for judgements derivable in this system.

Notice that negation is not part of the type language, so does not occur in Γ nor in Δ.

$$\frac{\Gamma, \neg\Delta, \neg A \vdash \bot}{\Gamma \vdash A}\,(PbC) \qquad \frac{\frac{}{\Gamma, \neg\Delta, \neg A \vdash \neg A}\,(Ax) \quad \Gamma, \neg\Delta, \neg A \vdash A}{\Gamma, \neg\Delta, \neg A \vdash \bot}\,(\neg E)$$

The intention of this system is to express classical logic, and for this it encapsulates the rule (PbC). To see this, we need first to emphasise that it is the intention that the formulas in Δ can be seen as negated. In fact, for any statement $\Gamma \vdash_{\mathrm{F}} A \mid \Delta$ can be seen as $\Gamma, \neg\Delta \vdash_{\mathrm{NI}} A$ (where $\neg\Delta$ lists the negated versions of all types in Δ). With that view, the rules (Act) and $(Pass)$ corresponds to allowing the shown variants of rule (PbC) and $(\neg E)$ but in a version of Natural Deduction where formulas have at most a negation at the front. Note that it therefore solves the problem of Example 3.2 by not allowing the rule (PbC) to be applied to assumptions on the right in $(\neg E)$: A cannot be a negated type, so the judgements in the right-hand proof cannot occur swapped.

The variant of $\lambda\mu$ considered in this paper, as defined by [37] and that gives a Curry-Howard interpretation to the above inference rules will now be presented.

Definition 4.2 (SYNTAX OF $\lambda\mu$) The *$\lambda\mu$-terms* considered here are defined by the grammar:

$$\begin{array}{rcll} M,N & ::= & V \mid MN \mid \mu\alpha.[\beta]M & \\ V & ::= & x \mid \lambda x.M & (values) \end{array}$$

Recognising both λ and μ as binders, the notion of free and bound names and variables is defined as usual, and Barendregt's convention is extended to keep free and bound names and variables distinct, using (silent) α-conversion whenever necessary.

The $x \in M$ ($\alpha \in M$) if x (α) occurs in M, either free of bound, and call a term *closed* if it has no free names or variables. The pseudo-terms of the shape $[\alpha]M$ are called *commands*, written *Cmd*, and are treated as terms for reasons of brevity, whenever convenient.

4.1 Reduction

As with Implicative Intuitionistic Logic, the reduction rules for the terms that represent the proofs correspond to proof contractions, but in $\vdash_{\mathrm{F}}$; the difference is that the reduction rules for the λ-calculus are the *logical* reductions, *i.e.* deal with the removal of a introduction-elimination pair for a type construct. In addition to these, Parigot expresses also the *structural* rules that change the focus of a proof, where elimination essentially deals with negation and takes place for a type constructor that appears in one of the alternative conclusions. In $\lambda\mu$, reduction of terms is expressed via implicit substitution, and as usual, $M\{N/x\}$ stands for the (instantaneous) substitution of all occurrences of x in M by N. Two kinds of structural substitution are defined: the first is the standard one, where $M\{N{\cdot}\gamma/\alpha\}$ stands for the term obtained from M in which every command of the form $[\alpha]P$ is replaced by $[\gamma]PN$ (γ is a fresh name). The second will be of use for CBV reduction; here $\{N{\cdot}\gamma/\alpha\}M$ stands for the term obtained from M in which every $[\alpha]P$ is replaced by $[\gamma]NP$.

Definition 4.3 (STRUCTURAL SUBSTITUTION) *Right-structural substitution*, $M\{N{\cdot}\gamma/\alpha\}$, and *left-*

structural substitution, $\{N{\cdot}\gamma/\alpha\}\,M$, are defined inductively over pseudo terms. The main cases are:

$$\begin{array}{ll} [\alpha]M\{N{\cdot}\gamma/\alpha\} \triangleq [\gamma](M\{N{\cdot}\gamma/\alpha\}N) & \{N{\cdot}\gamma/\alpha\}[\alpha]M \triangleq [\gamma]N(\{N{\cdot}\gamma/\alpha\}\,M) \\ [\beta]M\{N{\cdot}\gamma/\alpha\} \triangleq [\beta](M\{N{\cdot}\gamma/\alpha\})\ (\beta \neq \alpha) & \{N{\cdot}\gamma/\alpha\}[\beta]M \triangleq [\beta]\{N{\cdot}\gamma/\alpha\}\,M\ (\beta \neq \alpha) \end{array}$$

The following notions of reduction are defined on $\lambda\mu$. For the third, call by value, different variants exists in the literature; the one adopted here is from [35].

Definition 4.4 ($\lambda\mu$ REDUCTION) *i*) The reduction rules of $\lambda\mu$ are:

$$\begin{array}{rrcl} \textit{logical}\ (\beta): & (\lambda x.M)N & \to & M\{N/x\} \\ \textit{structural}\ (\mu): & (\mu\alpha.\mathit{Cmd})N & \to & \mu\gamma.\mathit{Cmd}\{N{\cdot}\gamma/\alpha\} \quad (\gamma\ \textit{fresh}) \\ \textit{renaming}\ (\mathit{Rename}): & [\beta]\mu\gamma.\mathit{Cmd} & \to & \mathit{Cmd}\{\beta/\gamma\} \end{array}$$

Evaluation contexts are defined as terms with a single hole $\lceil\,\rfloor$ by:

$$\mathsf{C} ::= \lceil\,\rfloor \mid \mathsf{C}M \mid M\mathsf{C} \mid \lambda x.\mathsf{C} \mid \mu\alpha.[\beta]\mathsf{C}$$

(Free, unconstrained) reduction $\to_{\beta\mu}$ on $\lambda\mu$-terms is defined through $\mathsf{C}\lceil M\rfloor \to_{\beta\mu} \mathsf{C}\lceil N\rfloor$ if $M \to N$ using either the β, μ, *Erase*, or *Rename*-reductions rule.

ii) CBN *evaluation contexts* are defined as:

$$\mathsf{C}_{\mathbf{N}} ::= \lceil\,\rfloor \mid \mathsf{C}_{\mathbf{N}}M \mid \mu\alpha.[\beta]\mathsf{C}_{\mathbf{N}}$$

CBN reduction $\to^{\mathbf{N}}_{\beta\mu}$ is defined through: $\mathsf{C}_{\mathbf{N}}\lceil M\rfloor \to^{\mathbf{N}}_{\beta\mu} \mathsf{C}_{\mathbf{N}}\lceil N\rfloor$ if $M \to N$ using either the β, μ, *Erase*, or *Rename*-reduction rule.

iii) CBV *evaluation contexts* are defined through:

$$\mathsf{C}_{\mathbf{V}} ::= \lceil\,\rfloor \mid \mathsf{C}_{\mathbf{V}}M \mid V\mathsf{C}_{\mathbf{V}} \mid \mu\alpha.[\beta]\mathsf{C}_{\mathbf{V}}$$

CBV reduction $\to^{\mathbf{V}}_{\beta\mu}$ is defined through: $\mathsf{C}_{\mathbf{V}}\lceil M\rfloor \to^{\mathbf{V}}_{\beta\mu} \mathsf{C}_{\mathbf{V}}\lceil N\rfloor$ if $M{\to}N$ using either μ, *Erase*, *Rename*, or:

$$\begin{array}{rrcl} (\beta_{\mathbf{V}}): & (\lambda x.M)V & \to^{\mathbf{V}}_{\beta\mu} & M\{V/x\} \\ (\mu_{\mathbf{V}}): & V(\mu\alpha.\mathit{Cmd}) & \to^{\mathbf{V}}_{\beta\mu} & \mu\gamma.\{V{\cdot}\gamma/\alpha\}\mathit{Cmd} \quad (\gamma\ \textit{fresh}) \end{array}$$

Remark that, for rule $(\mu_{\mathbf{V}})$, $\mu\alpha.[\beta]N$ is not a value. Also, unlike for the λ-calculus, CBV reduction is not a sub-reduction system of $\to_{\beta\mu}$: the rule $(\mu_{\mathbf{V}})$ (and left-structural substitution) are not part of $\to_{\beta\mu}$. Both CBN and CBV constitute *reduction strategies* in that they pick exactly one free $\beta\mu$-redex to contract; notice that a term might be in either CBN or CBV-normal form (*i.e.* reduction has stopped), but not need be that for $\to_{\beta\mu}$.

It is possible to define more reduction rules, but Parigot refrained from that since he aimed at defining a confluent reduction system. It is possible to add the CBV-rules to $\lambda\mu$, and define

$$\begin{array}{rrcl} (\mu_L): & (\mu\alpha.\mathit{Cmd})N & \to_{\beta\mu} & \mu\gamma.\mathit{Cmd}\{N{\cdot}\gamma/\alpha\} \quad (\gamma\ \textit{fresh}) \\ (\mu_R): & M(\mu\alpha.\mathit{Cmd}) & \to_{\beta\mu} & \mu\gamma.\{M{\cdot}\gamma/\alpha\}\mathit{Cmd} \quad (\gamma\ \textit{fresh}) \end{array}$$

but then reduction would no longer be confluent: the critical pair $(\mu\alpha.\mathit{Cmd}_1)(\mu\beta.\mathit{Cmd}_2)$ can be contracted in two ways, with perhaps different outcomes.

4.2 Type assignment

Type assignment for $\lambda\mu$ is defined below; there is a *main*, or *active*, conclusion, labelled by a term, and the *alternative* conclusions are labelled by names α, β, *etc*. Judgements in $\lambda\mu$ are of the shape $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, where Δ consists of pairs of Greek characters (the *names*) and types; the left-hand context Γ, as for the λ-calculus, contains pairs of Roman characters and types.

Definition 4.5 (TYPING RULES FOR $\lambda\mu$) *i*) Let φ range over a countable (infinite) set of type-variables. The set of *Curry types* is defined by the grammar:

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

ii) A *context* (of term variables) Γ is a partial mapping from term variables to types, denoted as a finite set of *statements* $x{:}A$, such that the *subjects* of the statements (x) are distinct. Γ_1,Γ_2 is the *compatible* union of Γ_1 and Γ_2 (if $x{:}A_1 \in \Gamma_1$ and $x{:}A_2 \in \Gamma_2$, then $A_1 = A_2$), and $\Gamma,x{:}A$ stands for $\Gamma,\{x{:}A\}$, $x \notin \Gamma$ if there exists no A such that $x{:}A \in \Gamma$, and $\Gamma \setminus x$ for $\Gamma \setminus \{x{:}A\}$.

iii) A *context of names* Δ (or *co-context*) is a partial mapping from *names* to types, denoted as a finite set of *statements* $\alpha{:}A$, such that the *subjects* of the statements (α) are distinct. Notions Δ_1,Δ_2, as well as $\Delta,\alpha{:}A$ and $\alpha \notin \Delta$ are defined as for Γ.

iv) The type assignment rules for $\lambda\mu$, adapted to our notation, are:

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

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

$\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$ is used for statements derivable in this system.

v) Barendregt's convention on free and bound variables and names is extended to judgements (for all the notions of type assignment defined here), so in $\Gamma,x{:}A \vdash_{\lambda\mu} M : B \mid \alpha{:}C,\Delta$, both x and α cannot appear bound in M.

Notice that, by the extension of Barendregt's convention in Definition 4.5, Γ' and Δ' cannot contain statements for the bound names and variables in M.

It is worthwhile mentioning that $\bot$ plays no role in the presentation of $\lambda\mu$, and only pops up in the literature when trying to inhabit double negation elimination (see Sect. 4.3).

$$\dfrac{\dfrac{\Gamma \vdash B \mid A,B,\Delta}{\Gamma \vdash \bot \mid A,B,\Delta}\,(Pass)}{\Gamma \vdash A \mid B,\Delta}\,(Act) \qquad \dfrac{\dfrac{\Gamma \vdash A \mid A,\Delta}{\Gamma \vdash \bot \mid A,\Delta}\,(Pass)}{\Gamma \vdash A \mid \Delta}\,(Act)$$

Notice that, if all term information is erased from the inference rules, the rules from $\vdash_{\text{F}}$ appear, but for the variants of (μ); these can be inferred, however, so they are admissible.

The intuition behind the structural rule is given by [17]: *"in a $\lambda\mu$-term $\mu\alpha.M$ of type $A\to B$, only the subterms named by α are* really *of type $A\to B$ (...); hence, when such a μ-abstraction is applied to an argument, this argument must be passed over to the sub-terms named by α."* Remark that this is accurate, but hides the fact that the naming construction $[\alpha]M$ is actually a (hidden) instance of rule $(\neg E)$, so 'naming' is actually an application.

4.3 Double negation elimination and $\lambda\mu$

Double negation elimination is shown in $\vdash_{\text{NI}}$ by the proof on the left; this can also be shown in $\vdash_{\text{F}}$, as in the proof on the right ($\bot$ is added to express negation, and $\Gamma = (C\to\bot)\to\bot$):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg C,\neg C \vdash \neg\neg C}\,(Ax) \quad \dfrac{}{\neg\neg C,\neg C \vdash \neg C}\,(Ax)}{\neg\neg C,\neg C \vdash \bot}\,(\neg E)}{\neg\neg C \vdash C}\,(PbC)}{\vdash \neg\neg C \Rightarrow C}\,(\to I) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (C\to\bot)\to\bot \mid C}\,(Ax) \quad \dfrac{\dfrac{\dfrac{}{\Gamma,C \vdash C \mid C}\,(Ax)}{\Gamma,C \vdash \bot \mid C}\,(Pass)}{\Gamma \vdash C\to\bot \mid C}\,(\to I)}{\Gamma \vdash \bot \mid C}\,(\to E)}{\Gamma \vdash C \mid}\,(Act)}{\vdash ((C\to\bot)\to\bot)\to C \mid \bot}\,(\to I)$$

Notice that the rules ($Pass$) and (Act) are not paired, while they are in $\lambda\mu$, and that the assumption $\neg C \vdash_{\text{NI}} \neg C$ gets replaced by the proof for $\vdash_{\text{F}} C\to\bot \mid C$.

Using this transformation, [36] shows that 'double negation elimination' can be represented in $\lambda\mu$ through the term $\lambda y.\mu\alpha.[\gamma]y(\lambda x.\mu\delta.[\alpha]x)$. As suggested above, first $\bot$ is added as a pseudo-type to express negation $\neg A$ through $A\to\bot$, as well as contradiction.

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

This corresponds to the proof in $\vdash_{\text{F}}$ above, but for the fact that extra calls to (*Pass*) and (*Act*) are added inside the calls to (μ), as well as additional names of type $\bot$; notice that this term is not closed as it has a free name γ.

Parigot essentially replaces here an instance of the (Ax) rule for $\Gamma,\neg C \vdash_{\text{NI}} \neg C$ by a derivation for $\Gamma \vdash_{\lambda\mu} \lambda y.\mu\delta.[\alpha]y : C{\to}\bot \mid \alpha{:}C,\Delta$. It is this what allows for the successful encoding of $\vdash_{\text{NI}}$ in $\lambda\mu$.

This kind of transformation will play an important role later in the paper.

It is important to point out that the use of γ in the previous example creates an anomaly. Although it is a logical tautology, the $\lambda\mu$-term that is the witness for $((C{\to}\bot){\to}\bot){\to}C$ is *not* a closed term so the proof has an uncanceled assumption. Moreover, terms can have type $\bot$ without being typed with the equivalent of rule $(\neg I)$, but using $(\to E)$.

Several attempts have been made to address this. Parigot not only adds $\bot$ to the language of types in a side remark, but also allows for statements like $\gamma{:}\bot$ to be used without adding them explicitly to the co-context, so does not consider them 'real' assumptions. [1] rectify that by defining an extension of $\lambda\mu$, adding a special syntax construct $[\mathsf{tp}]M$, where tp acts as a 'continuation constant' and represent the outermost context of the term. In their system, the witness to $((C{\to}\bot){\to}\bot){\to}C$ is the closed term $\lambda y.\mu\alpha.[\mathsf{tp}]y(\lambda x.\mu\delta.[\alpha]x)$. It can be argued that it would be better to add negation as a type constructor to $\lambda\mu$, as done in [42], where DNE gets represented by $\lambda y.\mu x.yx$, arguably a more simple proof.

Another solution would be to detach, syntactically, passivation from activation. That is the approach in de Groote and Saurin's $\Lambda\mu$-calculus [17, 39]; there the witness would be $\lambda y.\mu\alpha.y(\lambda x.[\alpha]x)$ which directly inhabits the proof in $\vdash_{\text{F}}$ above. That variant of $\lambda\mu$ better expresses the logic of $\vdash_{\text{F}}$, but one problem with $\Lambda\mu$ is that is not clear if (denotational) semantics can be defined for it, which is possible for $\lambda\mu$ [41, 3]. This is directly related to the fact that a μ-abstraction can now be applied to a term of type $\bot$ that is an application, rather than a term typed (implicitly) with rule $(\neg E)$.

5 Dependent Types and Control

Systems with dependent types allow for reasoning about quantified propositions – those in first order logic. A theorem prover for first order classical logic would then certainly need dependent types. In this section, after a quick review of ITT, we will show that a naive combination of dependent types with the μ operator leads to an inconsistent logic. We will then discuss a calculus for arithmetic that safely allow the two to co-exist in a type system by restricting their interactions, and start to explore how this calculus can be expanded to more general domains.

5.1 Dependent Types

Intuitionistic Type Theory (ITT) is a proof system for first order intuitionistic logic, first introduced by [27]. ITT introduces *dependent types* to the λ-calculus, which correspond with quantification in first order logic. In a dependent type system, types are able to depend on terms. A full presentation of the syntax of a standard ITT can be found in Figure 1.

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

Figure 1. Syntax for ITT

Dependent functions are functions whose codomain (return type) depends on the argument supplied to the domain. For example, a function $\lambda x.M$ is typed by $(x{:}A)\to B$, the dependent function type. Here, B is a *family* of types, indexed by values of type A; one could think of B itself as being a 'function' that takes values x of type A and returns a type, $B(x)$. Supplying an argument N to $\lambda x.M$ determines the return type, $(\lambda x.M)N : B\{N/x\}$. These dependent functions relate to 'for all' quantification in logic: $\lambda x.M : (x{:}A)\to B$ represents a proof that, for all x of type A, $B(x)$ holds.

Dependent pairs are pairs (M,N) with type $(x{:}A)\times B$, where the type family B is indexed by the left value, M, of the pair. Dependent pairs correspond with an existential proposition. In constructive logic, a proof of an existential formula $\exists(x{:}A).B(x)$ is formed by providing an example of such an x (called the *witness*, W), and then showing $B(W)$. Thus, the proof of an existential is a pair of a witness, W, and a proof P that the witness satisfies B, so $B(W)$ holds; in ITT this is written $(W,P) : (x{:}A)\times B$.

5.2 The Problem of Control

Given these two systems discussed above enable to reason about classical propositional logic and intuitionistic first order logic, one might want to see if they can be combined to create a system for reasoning with classical first order logic, that is, a system that allows for reasoning about $\forall,\exists$-quantified statements, and also proof by contradiction.

Unfortunately, dependent types do not sit well with classical calculi. As [19] discovered, a naive mix of dependent types and control leads to a 'degeneracy in the domain of discourse', meaning that all types can be shown to have only one inhabitant. Thus any two terms (under the same type) are judgementally equal.

An example of such an offending term, permitting a proof of $0=1$, is given by [29];

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

The projections of P show that $\pi_1(P)\to^* 0$ and $\pi_2(\mu\alpha.[\alpha](0,\mu\delta.[\alpha](1,\mathsf{refl})))\to^* \mathsf{refl}$.

$$\begin{array}{llllll}
\pi_1(\mu\alpha.[\alpha](0,\mu\delta.[\alpha](1,\mathsf{refl}))) & \to_{\mu_{\pi_1}} & \mu\alpha.[\alpha]\pi_1(0,\mu\delta.[\alpha]\pi_1(1,\mathsf{refl})) & \to_{\pi_1} & \mu\alpha.[\alpha]0 \to_{\mu_\eta} 0 \\
\pi_2(\mu\alpha.[\alpha](0,\mu\delta.[\alpha](1,\mathsf{refl}))) & \to_{\mu_{\pi_2}} & \mu\alpha.[\alpha]\pi_2(0,\mu\delta.[\alpha]\pi_2(1,\mathsf{refl})) & \to_{\pi_2} & \\
\mu\alpha.[\alpha]\mu\delta.[\alpha]\pi_2(1,\mathsf{refl}) & \to_{\pi_2} & \mu\alpha.[\alpha]\mu\delta.[\alpha]\mathsf{refl} & \to_{Rename} & \\
\mu\alpha.[\alpha]\mathsf{refl} & \to_{\mu_\eta} & \mathsf{refl} & &
\end{array}$$

Following the type assignment for $\pi_2(P)$ leads to the derivation on the right. However, noting that $\pi_1(P)$ reduces to 0, it follows that $(x=1)\{\pi_1(P)/x\}\equiv(\pi_1(P)=1)\equiv(0=1)$, and thus the type of $\pi_2(P)$ is $0=1$.

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma\vdash P:\exists(x{:}\mathbb{N}).x=1\mid\Delta\end{array}}{\Gamma\vdash\pi_2(P):(x=1)\{\pi_1(P)/x\}\mid\Delta}$$

As described by [29], the issue comes down to P behaving differently in different contexts. In the left projection, P gives the (incorrect) witness 0 to the proposition. In the right projection, the evaluation reaches the $\mu\delta.[\alpha]$, which causes it to 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. So P uses a different witness depending on its context.

5.3 Restricting Control and Dependent Types

As a step towards a general system with both dependent types and control, [20] defined the calculus dPA$^{\omega}$, a classical proof system for arithmetic; here, the operands of dependent eliminations are restricted to a subset of terms that are called *negative elimination free* (NEF).

As the name suggests, NEF terms are those that cannot contain a negation elimination $(\neg E)$. Thus, a NEF term cannot contain subterms of the form $\mu\alpha.[\beta]N$ as the subterm $[\beta]N$ corresponds with $(\neg E)$; or an application MN, as this could correspond with $(\neg E)$ when $M : A \to \bot$ and $N : A$. The exception to this is when these subterms appear under a lambda abstraction $\lambda x.P$, as a μ operator in a subterm is unable to capture a context outside of this abstraction; therefore lambda abstractions are always in NEF. NEF terms, then, are those that behave more like intuitionistic values [32], and are not able to backtrack when evaluated [28], avoiding the inconsistency caused by the above term P.

For example, the rules for dependent projections in dPA$^{\omega}$ are:

$$(\exists E_1^d) : \frac{\Gamma \vdash M : (x{:}B) \times A}{\Gamma \vdash \pi_1(M) : B} \; (M \in \textsc{nef}) \qquad (\exists E_2^d) : \frac{\Gamma \vdash M : (x{:}B) \times A}{\Gamma \vdash \pi_2(M) : A\{\pi_1(M)/x\}} \; (M \in \textsc{nef})$$

Immediately, these rules disallow recreating the proof P of $0 = 1$. The second projection of P, $\pi_2(P)$, is not typeable by $(\exists E_2^d)$, as P is not NEF. If the pair type is not dependent, the projection out of non-NEF pairs is still possible; there are separate rules for non-dependent eliminations:

$$(\exists E_1) : \frac{\Gamma \vdash M : (x{:}B) \times A}{\Gamma \vdash \pi_1(M) : B} \; (x \notin fv(A)) \qquad (\exists E_2) : \frac{\Gamma \vdash M : (x{:}B) \times A}{\Gamma \vdash \pi_2(M) : A} \; (x \notin fv(A))$$

In general, the NEF restrictions only apply to dependent types, and the non-dependent eliminations can be typed without regard for if the operands are NEF. In fact, dPA$^{\omega}$ with the NEF restriction is enough to give a consistent, normalising proof system [28, 20][1].

Definition 5.1 (REDUCTIONS) Reductions in this calculus largely follow a CBV strategy [20, 28]. The reductions for application and let expressions are:

$$\begin{aligned} (\lambda x.M)N &\to \text{let } x = M \text{ in } N \\ \text{let } x = V \text{ in } M &\to M\{V/x\} \end{aligned}$$

This strategy ensures that, in a dependent elimination, the reduction will only occur with a value operand (which implies it is NEF). The only exception to the CBV strategy is for the coinductive operator, `cofix`, which follows a lazy (call-by-need) reduction. Lazy reduction is needed as coinductive structures can be potentially infinite, so only the needed terms are evaluated.

5.4 The Calculus ECC$_K$

ECC$_K$ [31] is a classical extension to the Extended Calculus of Constructions (ECC) [26]. The classical reasoning is achieved by adding control operators, and restricting type assignments for dependent eliminations to NEF terms. ECC$_K$ is formally understood through its translation into L$_{\text{dep}}$, a classical sequent calculus with dependent types; the results for both calculi are part of ongoing work.

[1] More precisely, the normalisation property is shown for dLPA$^{\omega}$, a sequent calculus corresponding with dPA$^{\omega}$, although no formal translation between the two is made.

Valid Contexts $$(\cdot):\frac{}{\emptyset\vdash\cdot\mid\emptyset}\qquad(ax):\frac{\Gamma\vdash A:\mathcal{U}_i\mid\Delta}{\Gamma,x{:}A\vdash x:A\mid\Delta}\qquad(\alpha x):\frac{\Gamma\vdash A:\mathcal{U}_i\mid\Delta}{\Gamma\vdash\cdot\mid\alpha{:}A,\Delta}$$

Introduction/Formation

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

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

NEF $$(\textsc{nef}I):\frac{\Gamma\vdash M:A\mid\Delta}{\Gamma\vdash_{\textsc{nef}}M:A\mid\Delta}\ (M\in\textsc{nef})\qquad(\textsc{nef}E):\frac{\Gamma\vdash_{\textsc{nef}}M:A\mid\Delta}{\Gamma\vdash M:A\mid\Delta}$$

Control $$(\mathsf{catch}):\frac{\Gamma\vdash M:A\mid\alpha{:}A,\Delta}{\Gamma\vdash\mathsf{catch}_\alpha\ M:A\mid\Delta}\qquad(\mathsf{throw}_{)}:\frac{\Gamma\vdash M:A\mid\alpha{:}A,\Delta}{\Gamma\vdash\mathsf{throw}_\alpha\ M:B\mid\alpha{:}A,\Delta}$$

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

$\Gamma\vdash A:\mathcal{U}\mid\Delta$ is used when there exists i such that $\Gamma\vdash A:\mathcal{U}_i\mid\Delta$.

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

Equality $$(\mathsf{refl}):\frac{\Gamma\vdash A:\mathcal{U}\mid\Delta\quad\Gamma\vdash M:A\mid\Delta}{\Gamma\vdash\mathsf{refl}:M=_A M\mid\Delta}\qquad(=):\frac{\Gamma\vdash A:\mathcal{U}_i\mid\Delta\quad\Gamma\vdash M:A\mid\Delta\quad\Gamma\vdash N:A\mid\Delta}{\Gamma\vdash M=_A N:\mathcal{U}_i\mid\Delta}$$

$$(\mathsf{subst}):\frac{\Gamma,x{:}A\vdash B:\mathcal{U}\mid\Delta\quad\Gamma\vdash N:B\{P/x\}\mid\Delta\quad\Gamma\vdash M:P=Q\mid\Delta}{\Gamma\vdash\mathsf{subst}\ M\ N:B\{Q/x\}\mid\Delta}$$

Non-Dependent Elimination

$$(\to E):\frac{\Gamma\vdash M:A\to B\mid\Delta\quad\Gamma\vdash N:A\mid\Delta}{\Gamma\vdash MN:B\mid\Delta}\qquad(\mathsf{let}):\frac{\Gamma\vdash M:A\mid\Delta\quad\Gamma,x{:}A\vdash N:B\mid\Delta}{\Gamma\vdash\mathsf{let}\ x=M\ \mathsf{in}\ N:B\mid\Delta}\ (x\notin fv(B))$$

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

Dependent Elimination

$$(\to E^d):\frac{\Gamma\vdash M:(x{:}A)\to B\mid\Delta\quad\Gamma\vdash_{\textsc{nef}}N:A\mid\Delta}{\Gamma\vdash MN:B\{N/x\}\mid\Delta}\qquad(\mathsf{let}^d):\frac{\Gamma\vdash_{\textsc{nef}}M:A\mid\Delta\quad\Gamma,x{:}A\vdash N:B\mid\Delta}{\Gamma\vdash\mathsf{let}\ x=M\ \mathsf{in}\ N:B\{M/x\}\mid\Delta}$$

$$(\times E^d_1):\frac{\Gamma\vdash_{\textsc{nef}}M:(x{:}A)\times B\mid\Delta}{\Gamma\vdash\pi_1(M):A\mid\Delta}\qquad(\times E^d_2):\frac{\Gamma\vdash_{\textsc{nef}}M:(x{:}A)\times B\mid\Delta}{\Gamma\vdash\pi_2(M):B\{\pi_1(M)/x\}\mid\Delta}$$

Figure 2. $\mathrm{ECC_K}$ Type Assignments

Definition 5.2 ($\mathrm{ECC_K}$ Terms) The terms of $\mathrm{ECC_K}$ are defined as those of ITT, without coproducts ($A+B$, $\mathsf{inj}_i(M)$, case analysis) or the constants $\bot,\langle\rangle$ and 1, by adding the two control operators:

$$M,N,A,B\ ::=\ \ldots\mid\mathsf{catch}_\alpha\ M\mid\mathsf{throw}_\alpha\ M$$

In terms of $\lambda\mu$, $\mathsf{catch}_\alpha\ M$ and $\mathsf{throw}_\alpha\ M$ can be understood by $\mu\alpha.[\alpha]M$ and $\mu\_.[\alpha]M$ (where '_' is the Haskell notation for a name that does nor occur in $fn(M)$), respectively.

Definition 5.3 (Reduction and type assignment) The reductions (Figure 3b) are, like for dPA$^\omega$, CBV; reductions will only substitute with NEF terms, which helps maintain subject reduction.

The type assignment ruless for $\mathrm{ECC_K}$ [31] are presented in Figure 2. The rules infer judgements of the shape $\Gamma\vdash M:A\mid\Delta$; although there is little to distinguish the categories of terms and types, $M,N,P,Q,\ldots$ will be used for the 'terms' that appear on the left of the statement,

$$
\begin{aligned}
M_{\textsc{nef}}, N_{\textsc{nef}}, A_{\textsc{nef}}, B_{\textsc{nef}} &::= \\
& x \mid \mathcal{U}_i \mid \mathbb{P} \mid (x{:}A_{\textsc{nef}}) \to B_{\textsc{nef}} \\
& \mid \lambda x : A.M \mid \mathsf{let}\ x = M_{\textsc{nef}}\ \mathsf{in}\ n_{\textsc{nef}} \\
& \mid (x{:}A_{\textsc{nef}}) \times B_{\textsc{nef}} \mid (M_{\textsc{nef}}, n_{\textsc{nef}}) \\
& \mid \pi_i(M_{\textsc{nef}}) \mid M_{\textsc{nef}} = n_{\textsc{nef}}
\end{aligned}
$$

(a) ECC_K NEF Terms

$$
\begin{aligned}
(\lambda x.M)N &\to \mathsf{let}\ x = N\ \mathsf{in}\ M \\
\mathsf{let}\ x = V\ \mathsf{in}\ M &\to M\{V/x\} \\
\pi_i(\mathsf{let}\ x = M\ \mathsf{in}\ N) &\to \mathsf{let}\ x = M\ \mathsf{in}\ \pi_i(N) \\
\pi_i(v_1, v_2) &\to v_i \\
\mathsf{subst\ refl}\ M &\to M
\end{aligned}
$$

(b) ECC_K CBV Reductions

Figure 3. ECC_K

and $A, B, C, D, \ldots$ for the 'types' appearing on the right; lower case characters will be used for variables. This notational convention is especially useful when presenting inference rules.

Notice that the type system is very similar to that of ECC, except that the dependent eliminations are restricted to NEF terms.

6 ECC_μ: ECC with Control

ECC_K is an important step towards a general classical calculus, but, being part of ongoing work, its definition is not quite complete. In this section, we present the calculus ECC_μ, a direct extension of ECC_K. We first introduce coproducts with dependent elimination, which will be greatly useful in gaining intuition for how to add inductive data (Section 8), and define when they are NEF. We will then present a more complete set of CBV reductions for ECC_μ, which include the μ reductions, as well as notions of values, evaluation contexts, and normal forms. We then prove subject reduction for the calculus, and also sketch a proof of the consistency of ECC_μ.

Definition 6.1 (Terms of ECC_μ) The constants $\bot, \langle\rangle, 1$ and coproducts are added to the definition of syntax in Figure 4e. Note that we use the control operators μ and $[\cdot]$ from $\lambda\mu$ instead of the catch and throw operators of ECC_K. The normal forms (wrt to the reductions in Figure 5), and NEF terms are presented in Figures 4a and 4f, respectively. The NEF terms add the constants and coproducts to those for ECC_K; determining when coproduct terms are NEF comes from dPA^ω. Note that M need not be NEF for $\lambda x{:}A.M$ to be NEF.

When dealing with terms of the form $[\beta]M$, unless otherwise stated, the discussion will also apply to terms of the form $[\mathsf{tp}]M$.

The CBV reduction strategy is taken from dPA^ω; this is necessary as it ensures that substitutions from reductions will only occur with values (and thus NEF terms), which will allow to prove subject reduction. μ-reduction is generalised to be able to control any CBV context; this evaluation is deterministic under the CBV strategy.

Definition 6.2 (Values, Reduction, and Type Assignment) The values of the calculus (Figure 4c) generalise those of dPA^ω [20]. For determining when types are values, the intuition given by [40] is used; data constructors are values only when their arguments are values, and data types are always values. Then function, pair and coproduct types can be viewed as data types, and thus they are always values.

The reduction rules for ECC_μ are give in Figure 5. The evaluation contexts (Figure 4d) again come from generalising those in dPA^ω.

The type assignment rules for the terms shared by ECC_K and ECC_μ are the same as in Figure 2; those for the new terms in ECC_μ are presented in Figure 6. The rules for the control operators are the same as found in [2]. For the constants and coproducts, the rules can be found in [43], though, of course, for $(+E^d)$ we add the appropriate NEF restriction discussed above.

$$\begin{array}{lll}
N ::= & x \mid \bot \mid 1 \mid \langle\rangle \mid \mathcal{U}_i & \\
\mid & (x{:}N_1)\to N_2 \mid (x{:}N_1)\times N_2 \mid N_1 + N_2 \mid N_1 = N_2 & \\
\mid & \lambda x{:}A.N \mid \mathsf{refl} & \\
\mid & (N_1, N_2) & (N_i \neq \mu\delta.N') \\
\mid & \mathsf{inj}_i(N) & (N \neq \mu\delta.N') \\
\mid & \mu\alpha[\beta]N & (N \neq \mu\delta.N') \\
\mid & \mathsf{case}\ N \rhd z.N\ \mathsf{of}\ (x_1.N_1 \mid x_2.N_2) & \\
 & & (N \neq \mathsf{inj}_i(N'), \mu\alpha.N') \\
\mid & \pi_i(N) & (N \neq (N_1,N_2), \mu\alpha.N') \\
\mid & xN_1 \cdots N_k & (N_i \neq \mu\alpha.N') \\
\mid & \mathsf{subst}\ N_1\ N_2 & (N_1 \neq \mathsf{refl}, \mu\alpha.M)
\end{array}$$

(a) ECC$_\mu$ Normal Forms

$$\begin{array}{lll}
M_{\text{NEF}}, N_{\text{NEF}}, A_{\text{NEF}}, B_{\text{NEF}} & ::= & x \mid \bot \mid 1 \mid \langle\rangle \mid \mathcal{U}_i \\
\mid (x{:}A_{\text{NEF}})\to B_{\text{NEF}} & \mid & \lambda x{:}A.M \\
\mid \mathsf{let}\ x = M_{\text{NEF}}\ \mathsf{in}\ N_{\text{NEF}} & \mid & (x{:}A_{\text{NEF}})\times B_{\text{NEF}} \\
\mid (M_{\text{NEF}}, n_{\text{NEF}}) & \mid & \pi_i(M_{\text{NEF}}) \\
\mid M_{\text{NEF}} = N_{\text{NEF}} & \mid & \mathsf{refl} \\
\mid \mathsf{subst}\ M_{\text{NEF}}\ n_{\text{NEF}} & & \\
\mid A_{\text{NEF}} + B_{\text{NEF}} & \mid & \mathsf{inj}_i(M_{\text{NEF}}) \\
\mid \mathsf{case}\ M_{\text{NEF}} \rhd z.A\ \mathsf{of}\ (x.N_{1\text{NEF}} \mid y.N_{2\text{NEF}}) & &
\end{array}$$

(b) ECC$_\mu$ NEF Terms

$$\begin{array}{ll}
V ::= & x \mid \bot \mid 1 \mid \langle\rangle \mid \mathcal{U}_i \\
\mid & (x{:}A)\to B \mid (x{:}A)\times B \mid A + B \mid M = N \\
\mid & \lambda x{:}A.M \mid (V_1, V_2) \mid \mathsf{inj}_i(V) \mid \mathsf{refl}
\end{array}$$

(c) ECC$_\mu$ Values

$$\begin{array}{ll}
\mathcal{K} ::= & \bullet \mid \mathcal{K}M \mid v\mathcal{K} \\
\mid & \mathsf{inj}_i(\mathcal{K}) \mid (\mathcal{K}, M) \mid (V, \mathcal{K}) \\
\mid & \mathsf{case}\ \mathcal{K} \rhd z.A\ \mathsf{of}\ (x_1.N_1 \mid x_2.N_2) \\
\mid & \mathsf{let}\ x = \mathcal{K}\ \mathsf{in}\ M \mid \pi_i(\mathcal{K}) \mid \mathsf{subst}\ \mathcal{K}\ M
\end{array}$$

(d) ECC$_\mu$ CBV Evaluation Contexts

$$\begin{array}{lll}
M,N,\ A,B ::= \cdots & & \\
\mid & \bot & \text{Empty Type} \\
\mid & 1 & \text{Unit Type} \\
\mid & \langle\rangle & \text{Unit Element} \\
\mid & A + B & \text{Coproduct Type} \\
\mid & \mathsf{inj}_i(M) & \text{Injection } (i = 1,2) \\
\mid & \mathsf{case}\ M \rhd z.A\ \mathsf{of}\ (x.N_1 \mid y.N_2) & \\
 & \text{Coproduct Dependent Eliminator} &
\end{array}$$

(e) ECC$_\mu$ Terms

$$\begin{array}{ll}
M_{\text{NEF}}\ , N_{\text{NEF}} ::= & \cdots \mid \bot \mid 1 \mid \langle\rangle \\
\mid & \mathsf{refl} \\
\mid & \mathsf{subst}\ M_{\text{NEF}}\ N_{\text{NEF}} \\
\mid & A_{\text{NEF}} + B_{\text{NEF}} \\
\mid & \mathsf{inj}_i(M_{\text{NEF}}) \\
\mid & \mathsf{case}\ M_{\text{NEF}} \rhd z.A\ \mathsf{of}\ (x.N_{1\,\text{NEF}} \mid y.N_{2\,\text{NEF}})
\end{array}$$

(f) ECC$_\mu$ NEF Terms

Figure 4. ECC$_\mu$ Fragments and Notions

Crucially, under CBV in the term $(\lambda x.M)(\mu\alpha.N)$ the μ-reduction is prioritised over β-reduction. This is similar to Definition 4.4 ((iii)).

To justify the typing rule for dependent elimination of coproducts, observe the reduction of a case analysis:

$$\mathsf{case}\ \mathsf{inj}_i(M) \rhd z.A\ \mathsf{of}\ (x_1.N_1 \mid x_2.N_2) : A\{M/z\} \ \to\ \mathsf{let}\ x_i = M\ \mathsf{in}\ N_i : A\{\mathsf{inj}_i(M)/x_i\}$$

From the $\texttt{let}^d$ rule, it follows that $M \in$ NEF; therefore, in general, P must be NEF in $\mathsf{case}\ P \rhd z.A\ \mathsf{of}\ (x_1.N_1 \mid x_2.N_2)$. This also justifies the pattern matched methods $x_i.N_i$, as it is possible to determine when $\mathsf{let}\ x_i = M\ \mathsf{in}\ N_i \in$ NEF. If, instead, case analysis methods were functions i.e. $\mathsf{case}\ P \rhd z.A\ \mathsf{of}\ (N_1|N_2)$, then `case` would reduce to an application, which cannot be NEF – this would mean NEF terms are not closed under reduction.

It is possible to show closure of NEF terms under substitution and reductions.

Lemma 6.3 (NEF SUBSTITUTION / REDUCTION CLOSURE) ***Proof in A.1.1.***

i) $M,N \in$ NEF $\Rightarrow M\{N/x\} \in$ NEF.

ii) $M \in$ NEF *and* $M \to^* N \Rightarrow N \in$ NEF.

The closure of NEF-reduction is essential; NEF terms cannot contain a subterm that will backtrack, and will not reduce to a term that will backtrack. When expanding the calculus, this is a goal for when to define new syntax to be NEF; specifically, when a new term construct has a reduction into previously defined terms, the new construct is constricted to being NEF only when all of its single-step reductions are NEF. As a corollary, we get the same result for $M \to^* N$. A similar result for $M =_\beta N$ does not hold. Consider the example, if $P,Q \in$ NEF then

$$\begin{array}{rcll}
(\lambda x.M)N & \to & \text{let } x = N \text{ in } M, (M \neq \mu\alpha.M') & \\
\text{let } x = V \text{ in } M & \to & M\{V/x\} & \\
\mathcal{K}\{\text{let } x = M \text{ in } N\} & \to & \text{let } x = M \text{ in } \mathcal{K}\{N\} & \\
\text{case inj}_i(M) \rhd z.A \text{ of } (x_1.N_1 \mid x_2.N_2) & & & \\
& \to & \text{let } x_i = M \text{ in } N_i & \\
\pi_i(v_1, v_2) & \to & v_i & \\
\text{subst refl } M & \to & M & \\
\mathcal{K}\{\mu\alpha.M\} & \to & \mu\alpha.M\{[\alpha]\mathcal{K}\{\bullet\}/[\alpha]\bullet\} & \\
\mu\alpha.[\alpha]M & \to & M & (\alpha \notin fn(M)) \\
[\beta]\mu\delta.M & \to & M\{\beta/\delta\} &
\end{array}$$

(a) CBV Reductions

$$\begin{array}{rcl}
(\mu\alpha.M)N & \to & \mu\alpha.M\{[\alpha] \bullet N/[\alpha]\bullet\} \\
v(\mu\alpha.M) & \to & \mu\alpha.M\{[\alpha]v \bullet /[\alpha]\bullet\} \\
\pi_i(\mu\alpha.M) & \to & \mu\alpha.M\{[\alpha]\pi_i(\bullet)/[\alpha]\bullet\} \\
(v, \mu\alpha.M) & \to & \mu\alpha.M\{[\alpha](v, \bullet)/[\alpha]\bullet\} \\
(\mu\alpha.M, N) & \to & \mu\alpha.M\{[\alpha](\bullet, N)/[\alpha]\bullet\} \\
\text{let } x = \mu\alpha.M \text{ in } N & \to & \mu\alpha.M\{[\alpha]\text{let } x = \bullet \text{ in } N/[\alpha]\bullet\} \\
\text{inj}_i(\mu\alpha.M) & \to & \mu\alpha.M\{[\alpha]\text{inj}_i(\bullet)/[\alpha]\bullet\} \\
\text{case } \mu\alpha.M \rhd z.A \text{ of } (x_1.N_1 \mid x_2.N_2) \to & & \\
\multicolumn{3}{r}{\mu\alpha.M\{[\alpha]\text{case } \bullet \rhd z.A \text{ of } (x_1.N_1 \mid x_2.N_2)/[\alpha]\bullet\}} \\
\text{subst } (\mu\alpha.M)\ N & \to & \mu\alpha.M\{[\alpha]\text{subst } (\bullet)\ N/[\alpha]\bullet\}
\end{array}$$

(b) (μ)-reductions

Figure 5. ECC$_\mu$ Reductions

Constants $(\bot) : \dfrac{}{\Gamma \vdash \bot : \mathcal{U}_i \mid \Delta}$ $\quad (unit) : \dfrac{}{\Gamma \vdash \langle\rangle : 1 \mid \Delta}$ $\quad (1) : \dfrac{}{\Gamma \vdash 1 : \mathcal{U}_i \mid \Delta}$

Control $(\mu) : \dfrac{\Gamma \vdash M : \bot \mid \alpha{:}A, \Delta}{\Gamma \vdash \mu\alpha.M : A \mid \Delta}$ $\quad (name) : \dfrac{\Gamma \vdash M : A \mid \alpha{:}A, \Delta}{\Gamma \vdash [\alpha]M : \bot \mid \alpha{:}A, \Delta}$ $\quad (\text{tp}) : \dfrac{\Gamma \vdash M : \bot \mid \Delta}{\Gamma \vdash [\text{tp}]M : \bot \mid \Delta}$

Coproduct Introduction/Formation

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

Coproduct Non-Dependent Elimination

$$(+E) : \frac{\Gamma \vdash M : A{+}B \mid \Delta \quad \Gamma \vdash C : \mathcal{U} \mid \Delta \quad \Gamma, x{:}A \vdash N_1 : C \mid \Delta \quad \Gamma, y{:}B \vdash N_2 : C \mid \Delta}{\Gamma \vdash \text{case } M \rhd z.C \text{ of } (x.N_1 \mid y.N_2) : C \mid \Delta} \ (z \notin fv(C))$$

Coproduct Dependent Elimination

$$(+E^d) : \frac{\Gamma \vdash_{\text{NEF}} M : A{+}B \mid \Delta \quad \Gamma, z{:}A{+}B \vdash C : \mathcal{U} \mid \Delta \quad \Gamma, x{:}A \vdash N_1 : C_1 \mid \Delta \quad \Gamma, y{:}B \vdash N_2 : C_2 \mid \Delta}{\Gamma \vdash \text{case } M \rhd z.C \text{ of } (x.N_1 \mid y.N_2) : C[M/z] \mid \Delta}$$

with $C_1 = C\{\text{inj}_1(x)/z\}$ and $C_2 = C\{\text{inj}_2(y)/z\}$.

Figure 6. ECC$_\mu$ Type Assignments

$(\lambda x.P)Q \notin$ NEF; this reduces to let $x = Q$ in $P \in$ NEF. Thus they are β-equal, but not both in NEF.

This same idea will also help to determine when to allow for dependent elimination. When reducing new syntax to terms for which it is already known that it is not possible to allow dependent elimination, the requirements carry back to the new syntax.

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

Lemma 6.4 (TERM SUBSTITUTION) ***Proof in A.1.2.*** *If there exists 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 6.5 (SUBJECT REDUCTION) ***Proof in A.1.3*** *If $\Gamma \vdash M : A \mid \Delta$ and $M \to N$, then $\Gamma \vdash N : A \mid \Delta$.*

The calculus is consistent, with a proof sketch via an encoding into ECC$_K$.

Claim 6.6 (ECC$_\mu$ CONSISTENCY) **Proof Sketch in A.1.4** There is no term M such that $\emptyset \vdash M : \bot \mid \emptyset$

7 Typing Algorithm

To use ECC$_\mu$ as the core of a theorem prover, a typing algorithm is needed. In this section, we achieve this by defining a bidirectional algorithm. Part of the algorithm requires evaluating types to weak head normal form, so this notion will be introduced for ECC$_\mu$. At the end of this section, we discuss weakening the NEF restrictions to terms that reduce to NEF, providing a more expressive user-level language.

Extensive use is made of terms that are in *weak head normal form* (WHNF). The key idea behind terms in WHNF is that, normally using weak-head or lazy reduction they have been evaluated to the outermost data/type constructor or lambda abstraction [22], and that this 'head' of the term is not reducible. [5] have extended the notion of WHNF to $\lambda\mu$: for a term $\mu\alpha.[\beta]M$ to be WHNF, reductions on the head are not allowed. If $\alpha = \beta$, and $\alpha \notin fn(M)$, then the rule (μ_η) could be applied; so either $(\alpha \neq \beta$ or $\alpha \in fn(H)$ needs to hold in order for the term to be in WHNF. The other case to consider is the renaming reduction; when $M = \mu\gamma.[\delta]M'$, it is possible to reduce the term by $\mu\alpha.[\beta]\mu\gamma.[\delta]M' \to \mu\alpha.[\delta]M'[\beta/\gamma]$; thus, $M \neq \mu\gamma.[\delta]M'$. Moreover, the subterm itself, M, needs to be in WHNF.

The WHNF definition also respects 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.

Definition 7.1 (WEAK HEAD NORMAL FORM TERMS [5, 33])

$$\begin{array}{rlrlr}
H ::= & x \mid \lambda x.M \mid \mathsf{refl} \mid \langle\rangle \mid 1 \mid \bot \mid \mathcal{U}_i & & \mid \pi_i(H) & (H \neq (M,N) \text{ or } \mu\alpha.M) \\
& \mid (x{:}A)\to B \mid (x{:}A)\times B \mid A + B & & \mid \mathsf{case}\ H \rhd z.C\ \mathsf{of}\ (x_1.N_1 \mid x_2.N_2) & \\
& \mid (M,N) & (M,N \neq \mu\alpha.M') & & (H \neq \mathsf{inj}_i(M) \text{ or } \mu\alpha.M) \\
& \mid \mathsf{inj}_i(M) & (M \neq \mu\alpha.M') & \mid \mu\alpha.[\beta]H & (\alpha \neq \beta, \text{ or } \alpha \in fn(H); H \neq \mu\gamma.[\delta]M) \\
& \mid HM_1 \cdots M_n & (H \neq \lambda x.M \text{ or } \mu\alpha.M) & \mid \mathsf{subst}\ H\ M & (H \neq \mathsf{refl} \text{ or } \mu\alpha.N)
\end{array}$$

7.1 Bidirectional Typing

The approach to typing is to define a *bidirectional algorithm*, where there are two types of judgements [33]:

$$\begin{array}{ll}
\Gamma \vdash M \Rightarrow A \mid \Delta \rhd N & (\textit{Type Inference}) \\
\Gamma \vdash M \Leftarrow A \mid \Delta \rhd N & (\textit{Type Checking})
\end{array}$$

This can be read: infer the type A for M, with output N; and check the type A against the term M, with output N. During the typing algorithm, the terms are sometimes (partially) evaluated to check their weak-head normal form or if they are NEF; this evaluated term is given by the output N.

The bidirectional algorithm itself is largely standard, and is similar to that of [33]. The rules for coproducts are based on those in [38]. The new rules are highlighted; the full presentation (including subtyping rules) can be seen in Appendix A.3.1. The rules for elimination are split into their dependent and non-dependent versions, with the appropriate NEF checks, otherwise they are much the same as those in [33] and [38].

Soundness of the algorithm (wrt the type system) comes from the nature of bidirectional algorithms as it is directly derived from the type system. Completeness of this algorithm is not achieved; in general, terms can only be typed when 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: a function is defined by declaring its type and then giving an inhabiting term.

To type a μ-term, the assumption $\alpha{:}A$ is added to the co-context. This means the type A is needed before typing starts; so the (μ) rule must be treated as a type *checking* rule. As for the $(name)$ rule, it is in fact quite similar to the $(\to E)$ rule[2]. As the type of α is already known (otherwise it is not bound in the term and tp), we only need to check that M matches that type.

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

$$(name) : \frac{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t}{\Gamma \vdash [\alpha]M \Rightarrow \bot \mid \alpha{:}A, \Delta \rhd [\alpha]t}$$

$$(tp) : \frac{\Gamma \vdash M \Leftarrow \bot \mid \Delta \rhd t}{\Gamma \vdash [\mathsf{tp}]M \Rightarrow \bot \mid \Delta \rhd [\mathsf{tp}]t}$$

7.2 NEF Restrictions

The system is extended with the NEF *Rules*, which allow the conversion between $\vdash_{\text{NEF}}$ and $\vdash$. These rules are written with $\Box$, which can be replaced with either $\Rightarrow$ or $\Leftarrow$, allowing to check if 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.

From the user's perspective, however, this can lead to a very restrictive language, as, in the term $M(NP)$, M is not able to have a dependent type, as applications (NP) cannot be NEF– even if (NP) evaluates to some term $Q \in \text{NEF}$. This motivates the need to consider the class of terms RNEF; terms that *reduce* to NEF terms.

$$(\text{NEF}I) : \frac{\Gamma \vdash M \Box A \mid \Delta \rhd t}{\Gamma \vdash_{\text{NEF}} M \Box A \mid \Delta \rhd t} \ (M \in \text{NEF})$$

$$(\text{NEF}E) : \frac{\Gamma \vdash_{\text{NEF}} M \Box A \mid \Delta \rhd t}{\Gamma \vdash M \Box A \mid \Delta \rhd t}$$

A similar idea, is explored by [24], where they restrict dependent types to a class of terms called 'semantic values' (those that are equivalent to syntactic values) a value restriction for their dependent typings. The user is then able to write programs, unaware of the value restriction, as the typing algorithm attempts to find values equivalent to the user terms where needed; the type system uses explicit equivalency proofs for these value substitutions.

Definition 7.2 (RNEF) $M \in \text{RNEF}$ (NEF-reducible) when there exists a term $N \in \text{NEF}$ such that $M \to^* N$.

It is possible to add the rule displayed to the right, which greatly increases the expressiveness of the user-level language, but at the costof greatly slow down the type-checking algorithm, as all user-defined functions would have to be effectively inlined at each use.

$$(\text{NEF}I) : \frac{\Gamma \vdash N \Box A \mid \Delta \rhd t}{\Gamma \vdash_{\text{NEF}} M \Box A \mid \Delta \rhd t} \ (M \to^* N \in \text{NEF})$$

What is certainly worth exploring, then, is if there is a way to determine for a given function, f, the exact requirements on its arguments to be RNEF, and store these requirements alongside its definition. Then, when f is called, the type-checker would only need to consult these requirements, instead of evaluating f.

8 Dependent Algebraic Data Types

Most theorem provers and dependently typed languages boast the capability for user-defined (co)data types. In this section, we add inductive families and codata to ECC_μ by generalising the results for coproducts and dependent pairs. We first add the syntax for data to the calculus, and how to assign types to data, making the appropriate NEF restrictions. We will then do the same for codata. At the end of the section, we will present the reductions for data and codata. For codata, this requires a specific introduction of lazy evaluation and lazy evaluation contexts,

[2] 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 [42]

as we cannot eagerly evaluate an infinite structure. Finally, we will define the (weak head) normal forms and values of data and codata, to help include them in the typing algorithm.

The resulting calculus forms the basis of our theorem prover for classical logic, Candid.

Below the notation $\overrightarrow{o_i}$ is used to represent a vector/sequence of objects $o_1 \cdots o_k$; often the subsript $\cdot_i$ will be omitted, as in $\vec{o}$.

Definition 8.1 *i*) The notation $\overrightarrow{(a_i{:}A_i)}$ (or $\overrightarrow{(a:A)}$ if the range of the index is not important) is used as an abbreviation of the (finite) sequence of type assignments $(a_1{:}A_1)\cdots(a_n{:}A_n)$.

ii) $M\,\overrightarrow{N}$ is the sequence of terms formed by adding M to the start of the sequence $\overrightarrow{N}$; $(a{:}A)\cdot\overrightarrow{(b{:}B)}$ is the sequence formed by adding $(a{:}A)$ to the start of the sequence $\overrightarrow{(b{:}B)}$, and $\overrightarrow{(a{:}A)}\cdot\overrightarrow{(b{:}B)} = (a_1{:}A_1)\cdots(a_n{:}A_n)\cdot(b_1{:}B_1)\cdots(b_m{:}B_m)$.

iii) (Sequence Formation)

$$\frac{\Gamma \vdash A_1 : \mathcal{U} \mid \Delta \quad \cdots \quad \Gamma, a_1{:}A_1, \ldots, a_{n-1}{:}A_{n-1} \vdash A_n : \mathcal{U} \mid \Delta}{\Gamma \vdash \overrightarrow{(a_i{:}A_i) : \mathcal{U}} \mid \Delta}$$

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

iv) (Sequence Instance)

$$\frac{}{\Gamma \vdash \cdot : \cdot \mid \Delta} \qquad \frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \vdash \overrightarrow{N_i} : (\overrightarrow{b_i{:}B_i}\ \{M/x\}) \mid \Delta \quad (1 \le i \le n)}{\Gamma \vdash M\,\overrightarrow{N_i} : (x{:}A)\cdot\overrightarrow{(b_i{:}B_i)} \mid \Delta}$$

We write $\vec{x} : \overrightarrow{(a_i{:}A_i)}$ for the sequence of (dependent) assignments: $(x_1 : (a_1{:}A_1))\cdots(x_k : (a_k{:}A_k))$.

Below the notation $\overrightarrow{\{\Gamma \vdash N : A \mid \Delta\}}$ will be used for $\Gamma \vdash N_i : A_i \mid \Delta\ (1 \le i \le n)$.

8.1 Data

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

Definition 8.2 (Inductive Families) *i*) An inductive family D, with parameters p of type E_p and indices i or type F_i, is defined (in a style similar to [33]) by:

$$\textbf{data}\ D\,\overrightarrow{(p{:}E_p)} : \overrightarrow{(i{:}F_i)} \to \mathcal{U}\ \textbf{where}\ \left\{\mathsf{constr}_j : \overrightarrow{(a{:}A)}\ \overrightarrow{(b{:}B)} \to D\,\vec{p}\ \vec{S}_j\right\}_{j=1}^{k}$$

where:

- $\vec{p}$ are the *parameters*, $\vec{i}$ the *indices*
- $\vec{S}_j$ are the indices corresponding to the j-th constructor
- $\overrightarrow{(a{:}A)}$ are the non-recursive arguments (not containing D)
- $\overrightarrow{(b_j{:}B_j)}$ are the recursive arguments. Each B_j must be *positive* in D, i.e. is of the form $C_1 \to \cdots \to C_k \to D\,\vec{p}\ \vec{i}$, and the type D does not appear in any C_j [6].

Notice that this definition adds each constructor to the syntax of terms, and adds it type to an environment, or signature.

ii) The term syntax is given by:

$$\begin{aligned} M,N,A,B ::= {} & \cdots \mid D\,\overrightarrow{M}\ \overrightarrow{N} \mid \mathsf{constr}_i \\ & \mid \mathsf{elim}\ M \rhd \vec{z}.A\ \mathsf{by}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) \\ & \mid \mathsf{case}\ M \rhd \vec{z}.A\ \mathsf{of}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) \end{aligned}$$

iii) Defining the data constructors and eliminators NEF is similar to that for coproducts:

$$\begin{aligned} M,N,A,B ::= {} & \cdots \mid D\,\overrightarrow{M}_{\textsc{nef}}\,\overrightarrow{N}_{\textsc{nef}} \mid \mathsf{constr}_i\,\overrightarrow{M}_{\textsc{nef}} \\ & \mid \mathsf{elim}\ M_{\textsc{nef}} \rhd \vec{z}.A\ \mathsf{by}\ (\vec{x}_1.N_{1\textsc{nef}} \mid \cdots \mid \vec{x}_k.N_{k\textsc{nef}}) \\ & \mid \mathsf{case}\ M_{\textsc{nef}} \rhd \vec{z}.A\ \mathsf{of}\ (\vec{x}_1.N_{1\textsc{nef}} \mid \cdots \mid \vec{x}_k.N_{k\textsc{nef}}) \end{aligned}$$

In part (ii) the elim construct is for inductive eliminations that will recurse on the data type; case is for non-recursive elimination. This distinction is the same as that given by [6]. For elim $M \rhd \vec{z}.C$ by $(\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k)$, M is called the *target*, $\vec{z}.C$ the *motive*, and $\vec{x}_i.N_i$ the *methods*, and similar for the case construct.

Definition 8.3 To avoid cluttering the notation with too many indices, we will drop the universe levels (i.e. we will write $\mathcal{U}$ instead of $\mathcal{U}_i$), although they are implicitly present.

i) Inductive Family Definition Check [14, 6] Definitions are checked by ensuring the parameters and indices are types, and then checking the types of constructors given the parameter variables (but not the indices).

$$(\mathbf{data}): \quad \frac{\Gamma \vdash \overrightarrow{(p{:}E_p)} \cdot \overrightarrow{(i{:}F_i)} : \mathcal{U} \mid \Delta}{\Gamma \vdash D\ \overrightarrow{(p{:}E_p)}\ \overrightarrow{(i{:}F_i)} : \mathcal{U} \mid \Delta}$$

$$(\mathbf{where}): \quad \left\{ \frac{\overrightarrow{\left\{\Gamma, \overrightarrow{(p{:}E_p)} \vdash \overrightarrow{(a{:}A)} \cdot \overrightarrow{(b{:}B)} : \mathcal{U} \mid \Delta\right\}} \quad \overrightarrow{\left\{\Gamma, \overrightarrow{(p{:}E_p)}, \overrightarrow{(a{:}A)}, \overrightarrow{(b{:}B)} \vdash S : F \mid \Delta\right\}}}{\Gamma \vdash \mathsf{constr}_j : \overrightarrow{(a{:}A)}\ \overrightarrow{(b{:}B)} \to D\ \vec{P}\ \vec{S} \mid \Delta} \right\}_{j=1}^{k}$$

Here the type derived for the constructor must be the one stored in the environment.

ii) Inductive Family Formation/Introduction [14, 6]

To ensure an instantiation of a data type is valid, the parameters and indices must be of the correct types, and the parameter types P indeed types (i.e. members of a universe $\mathcal{U}$).

$$(data): \quad \frac{\overrightarrow{\{\Gamma \vdash P : (p{:}E) \mid \Delta\}} \quad \overrightarrow{\left\{\Gamma \vdash Q : (i{:}F)\{\overrightarrow{P/p}\} \mid \Delta\right\}}}{\Gamma \vdash D\ \vec{P}\ \vec{Q} : \mathcal{U} \mid \Delta}$$

Only constructors in *canonical form* [6] are considered, that is, those that are fully applied. This makes it easier to identify NEF terms.

$$(\mathsf{constr}_i): \quad \frac{\overrightarrow{\{\Gamma \vdash P : (p{:}E_p) \mid \Delta\}} \quad \overrightarrow{\left\{\Gamma \vdash M : (a{:}A)\{\overrightarrow{P/p}\} \mid \Delta\right\}} \quad \overrightarrow{\left\{\Gamma \vdash N : (b{:}B)\{\overrightarrow{P/p}, \overrightarrow{M/a}\} \mid \Delta\right\}}}{\Gamma \vdash \mathsf{constr}_j\ \vec{P}\ \vec{M}\ \vec{N} : D\ \vec{P}\ \vec{S} \mid \Delta}$$

iii) Inductive Family Dependent Elimination (elim)

$$(\mathsf{elim}^d): \quad \frac{\begin{array}{r} \overrightarrow{\left\{\Gamma, \vec{x_j} : \overrightarrow{(P{:}E)} \cdot \overrightarrow{(a{:}A)} \cdot \overrightarrow{(b{:}B)} \cdot \overrightarrow{(v{:}V)} \vdash_{\text{NEF}} N_j : C(\vec{P}\ \vec{Q}\ (\mathsf{constr}_j\, \vec{x}^{\,p}\, \vec{x}^{\,a}\, \vec{x}^{\,b})) \mid \Delta\right\}}_{j=1}^{k} \\ \Gamma \vdash_{\text{NEF}} M : D\ \vec{P}\ \vec{Q} \mid \Delta \quad \Gamma, \vec{z} : \overrightarrow{(P{:}E)} \cdot \overrightarrow{(Q{:}F)} \cdot (D\ \vec{P}\ \vec{Q}) \vdash C : \mathcal{U} \mid \Delta \qquad \vdots \end{array}}{\Gamma \vdash \mathsf{elim}\ M \rhd \vec{z}.C\ \mathsf{by}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) : C(\vec{P}\ \vec{Q}\ M) \mid \Delta}$$

where:
- $C(\vec{P}\ \vec{Q}\ M)$ stands for $C\{(\vec{P}\ \vec{Q}\ M)/\vec{z}\}$.
- $\vec{x}^{\,y}$ stands for the subsequence of $\vec{x}$ corresponding to the variables typed by $\overrightarrow{(y{:}Y)}$.
- each B is of the form $(k_1{:}K_1) \to \cdots \to (k_l{:}K_l) \to D\ \vec{P}\ \vec{S}$.
- each V is of the form $(u_1{:}U_1) \to \cdots \to (u_l{:}U_l) \to C(\vec{P}\ \vec{S}\ (b_j\, u_1 \cdots u_l))$, and is of the same length as the type B (i.e. the index l matches).

iv) Inductive Family Dependent Elimination (case)

$$(\mathsf{case}^d): \quad \frac{\begin{array}{l} \Gamma, \vec{z} : \overrightarrow{(p{:}E)} \cdot \overrightarrow{(q{:}F)} \cdot (D\ \vec{P}\ \vec{Q}) \vdash C : \mathcal{U} \mid \Delta \\ \Gamma \vdash_{\text{NEF}} M : D\ \vec{P}\ \vec{Q} \mid \Delta \quad \vdots \quad \overrightarrow{\left\{\Gamma, \vec{x_j} : \overrightarrow{(p{:}E)} \cdot \overrightarrow{(a{:}A)} \cdot \overrightarrow{(b{:}B)} \vdash N_j : C(\vec{P}\ \vec{Q}\ (\mathsf{constr}_j\, \vec{x}^{\,p}\, \vec{x}^{\,a}\, \vec{x}^{\,b})) \mid \Delta\right\}}_{j=1}^{k} \end{array}}{\Gamma \vdash \mathsf{case}\ M \rhd \vec{z}.C\ \mathsf{of}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) : C(\vec{P}\ \vec{Q}\ M) \mid \Delta}$$

v) Inductive Family Non-Dependent Elimination.

$$(\mathsf{elim}): \quad \frac{\begin{array}{l} \Gamma \vdash C : \mathcal{U} \mid \Delta\ (z_i \not\in fv(C)) \\ \Gamma \vdash M : D\ \vec{P}\ \vec{Q} \mid \Delta \quad \vdots \quad \overrightarrow{\left\{\Gamma, \vec{x_j} : \overrightarrow{(p{:}E)} \cdot \overrightarrow{(a{:}A)} \cdot \overrightarrow{(b{:}B)} \cdot \overrightarrow{(v{:}V_i)} \vdash N_j : C \mid \Delta\right\}}_{j=1}^{k} \end{array}}{\Gamma \vdash \mathsf{elim}\ M \rhd \vec{z}.C\ \mathsf{by}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) : C \mid \Delta}$$

$$(\mathsf{case}): \frac{\Gamma \vdash M : D\,\vec{P}\,\vec{Q} \mid \Delta \quad \Gamma \vdash C : \mathcal{U} \mid \Delta\ (z_i \notin fv(C)) \quad \overrightarrow{\left\{\Gamma, \vec{x_j} : \overrightarrow{(p{:}E_p)} \cdot \overrightarrow{(a{:}A)} \cdot \overrightarrow{(b{:}B)} \vdash N_j : C \mid \Delta\right\}_{j=1}^{k}}}{\Gamma \vdash \mathsf{case}\ M \rhd \vec{z}.C\ \mathsf{of}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) : C \mid \Delta}$$

The reasoning behind the NEF restrictions of both the target and all the methods in part (iii) comes from the reduction for elim in Figure 7. In particular, the rule (let^d) needs the subterms assigned to x_i to be NEF. This means each $\vec{w}_j.(\mathsf{elim}\ b_j\,\vec{w}_j \rhd \vec{z}.C\ \mathsf{by}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k))$ must be NEF, and thus, as $\vec{w}_j$ could have length 0, each $n_i \in$ NEF. Also $a,b \in$ NEF, which generalises to the target being NEF.

The reasoning behind the NEF restriction for the target in the rule (case^d) again comes from observing the reductions in Figure 7. From the let^d rule, we see we need $a,b \in$ NEF, thus, more generally, we need the target to be NEF.

Just like with coproducts, the non-dependent elimination needs no NEF restrictions. This can be seen from the reductions (in Section 8.3) for both elim and case, in which they reduce to let expressions where the type is not dependent in the bound variable. Thus, by the (let) rule, we know we don't need any of the terms to be NEF.

8.2 Codata

As suggested by [33], codata can be seen as a generalisation of dependent pairs. A codata type R with parameters $\overrightarrow{(p{:}E_p)}$ is defined by;

$$\mathbf{codata}\,R\,\overrightarrow{(p{:}E_p)} : \mathcal{U} \quad \mathbf{where}\ \{\mathsf{proj}_i : A_i\}_{i=1}^{k}$$

where A_i is strictly positive w.r.t. R, and $fv(A_i) \subseteq \vec{p}$, and $\mathsf{proj}_1, \cdots, \mathsf{proj}_{i-1}$ can appear in A_i. We extend the syntax similarly to how we did for data types:

$$M, N, A, B ::= \cdots \mid R\,\vec{p} \mid \mathsf{proj}_i(M) \mid \mathsf{build}(N_1 \mid \cdots \mid N_k)$$

We define when the terms are NEF, which can be obtained by generalising those of product types:

$$M_{\text{NEF}}, N_{\text{NEF}}, A_{\text{NEF}}, B_{\text{NEF}} ::= \cdots \mid R\,\vec{p}_{\text{NEF}} \mid \mathsf{proj}_i(M_{\text{NEF}}) \mid \mathsf{build}(N_{\text{NEF}1} \mid \cdots \mid N_{\text{NEF}k}).$$

Definition 8.4 (Type Assignment for Codata) *i) (Codata Declaration)*

$$(\mathbf{codata}): \frac{\Gamma \vdash \overrightarrow{(p{:}E_p)} :: \mathcal{U} \mid \Delta}{\Gamma \vdash R\,\overrightarrow{(p{:}E_p)} : \mathcal{U} \mid \Delta} \qquad (\mathbf{where}): \left\{ \frac{\Gamma, \overrightarrow{(p{:}E_p)}, \mathsf{proj}_1{:}A_1, \cdots, \mathsf{proj}_{i-1}{:}A_{i-1} \vdash A_i : \mathcal{U} \mid \Delta}{\Gamma, \overrightarrow{(p{:}E_p)}, \mathsf{proj}_1{:}A_1, \cdots, \mathsf{proj}_{i-1}{:}A_{i-1} \vdash \mathsf{proj}_i : A_i \mid \Delta} \right\}_{i=1}^{k}$$

ii) (Codata Formation/Introduction) The formation of codata 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. Given a valid codata declaration $R\,\overrightarrow{(p{:}E_p)}{:}\mathcal{U}$ in the context Γ, instances of the type and introductions are checked, where $\vec{P}$ is a term vector (of the same length as $\vec{p}$), by;

$$(codata): \frac{\Gamma \vdash \overrightarrow{P : (p{:}E_p)} \mid \Delta}{\Gamma \vdash R\,\vec{P} : \mathcal{U} \mid \Delta}$$

$$(build_R): \frac{\Gamma \vdash R\,\vec{P} : \mathcal{U} \mid \Delta \quad \Gamma \vdash N_1 : A_1\{\overrightarrow{P/p}\} \mid \Delta \quad \cdots \quad \Gamma \vdash N_k : A_k\{\overrightarrow{P/p}, \overrightarrow{N/p}\} \mid \Delta}{\Gamma \vdash \mathsf{build}(N_1 \mid \cdots \mid N_k) : R\,\vec{P} \mid \Delta}$$

iii) (Codata Projection) The projections are a generalisation of the pair projections. Given codata $R\,\overrightarrow{(p{:}E_p)} : \mathcal{U}$ in the context Γ, and $\vec{P}$ a term vector of the same length as $\vec{p}$, we type the projections by the following rules:

$$(\mathsf{proj}_i): \frac{\Gamma \vdash M : R\,\vec{P} \mid \Delta \quad \Gamma \vdash R\,\vec{P} : \mathcal{U} \mid \Delta}{\Gamma \vdash \mathsf{proj}_i(M) : A_i\{\overrightarrow{P/p}\} \mid \Delta}\ (p_j \notin fv(A_i)\ \text{for}\ j = 1, \cdots, (i-1))$$

$$(\mathsf{proj}_i^d): \frac{\Gamma \vdash_{\text{NEF}} M : R\,\vec{P} \mid \Delta \qquad \Gamma \vdash R\,\vec{P} : \mathcal{U} \mid \Delta}{\Gamma \vdash \mathsf{proj}_i(M) : A_i\{\overrightarrow{P/p}, \mathsf{proj}_1(M)/\mathsf{proj}_1, \cdots, \mathsf{proj}_{i-1}(M)/\mathsf{proj}_{i-1}\} \mid \Delta}$$

$$\begin{array}{rcl}
\mathsf{constr}_i\,\overrightarrow{V}\;\mu\alpha.M\,\overrightarrow{N} & \to & \mu\alpha.M\{[\alpha]\mathsf{constr}_i\,\overrightarrow{V}\;\bullet\;N/[\alpha]\bullet\} \\
\mathsf{case}\;(\mathsf{constr}_i\,\vec{a}\;\vec{b}\,)\rhd\vec{z}.C\;\mathsf{of}\;(\vec{x}_1.N_1\mid\cdots\mid\vec{x}_k.N_k) & \to & \mathsf{let}\;\vec{x}_i=\vec{a}\;\vec{b}\;\mathsf{in}\;N_i \\
\mathsf{elim}\;(\mathsf{constr}_i\,\vec{a}\;\vec{b}\,)\rhd\vec{z}.C\;\mathsf{by}\;(\vec{x}_1.N_1\mid\cdots\mid\vec{x}_k.N_k) & \to &
\end{array}$$

$$\mathsf{let}\,\vec{x}\;=\;\left\{\begin{array}{l}
\vec{a}\;\vec{b} \\
\vec{w}_1.(\mathsf{elim}\;b_1\,\vec{w}_1\rhd\vec{z}.C\;\mathsf{by}\;(\vec{x}_1.N_1\mid\cdots\mid\vec{x}_k.N_k)) \\
\qquad\cdots \\
\vec{w}_l.(\mathsf{elim}\;b_l\,\vec{w}_l\rhd\vec{z}.C\;\mathsf{by}\;(\vec{x}_1.N_1\mid\cdots\mid\vec{x}_k.N_k))
\end{array}\right\}\mathsf{in}\,N_i$$

Figure 7. Reductions for Inductive Families

$$\begin{array}{rcl}
\mathcal{K}\{\mathsf{build}(N_1\mid\cdots\mid N_k)\} & \to & \mathsf{let}\;x=\mathsf{build}(N_1\mid\cdots\mid N_k)\;\mathsf{in}\;\mathcal{K}\{x\} \\
\mathsf{let}\;x=\mathsf{build}(N_1\mid\cdots\mid N_k)\;\mathsf{in}\;\mathcal{L}\{\mathsf{proj}_i(x)\} & \to & \mathsf{let}\;x=N_i\;\mathsf{in}\;\mathcal{L}\{x\} \\
\mathsf{build}(\overrightarrow{V}\mid\mu\alpha.M\mid\overrightarrow{N}\,) & \to & \mu\alpha.M\{[\alpha]\mathsf{build}(\overrightarrow{V}\mid\bullet\mid\overrightarrow{N}\,)/[\alpha]\bullet\} \\
\mathsf{let}\;x=\mathsf{build}(N_1\mid\cdots\mid N_k)\;\mathsf{in}\;\mu\alpha.[\beta]M & \to & \mu\alpha.[\beta]\mathsf{let}\;x=\mathsf{build}(N_1\mid\cdots\mid N_k)\;\mathsf{in}\;M
\end{array}$$

Figure 8. Lazy Reductions for Codata

8.3 Reductions for (Co)Inductive Types

Viewing a constructor constr_i fully applied to its arguments as function application, CBV contexts are expanded by:

$$\begin{array}{rcl}
\mathcal{K} & ::= & \cdots\mid\mathsf{constr}_i\,\vec{v}\;\mathcal{K}\,\overrightarrow{M}\mid\mathsf{proj}_i(\mathcal{K}) \\
& \mid & \mathsf{case}\;\mathcal{K}\rhd\vec{z}.A\;\mathsf{of}\;(\vec{x}_1.N_1|\cdots|\vec{x}_k.N_k) \\
& \mid & \mathsf{elim}\;\mathcal{K}\rhd\vec{z}.A\;\mathsf{by}\;(\vec{x}_1.N_1|\cdots|\vec{x}_k.N_k)
\end{array}$$

The CBV contexts for the constructors constr_i and destructors case and elim come from [40]. The reductions of inductive families are defined in Figure 7. $\vec{x}=\overrightarrow{M}\;\overrightarrow{N}$ represents that the sequence of variables $\vec{x}$ is of the same length as the sequence $\overrightarrow{M}\;\overrightarrow{N}$, and that each x_i is bound to the corresponding term on the right hand side.

A CBV strategy will not work for coinductive structures; they represent (potentially) infinite objects, so cannot be evaluated to completion. Thus, specifically for codata, CBN or lazy reduction is used. Following [20], this also requires the introduction of specific contexts for lazy evaluation, labeled $\mathcal{L}$.

Definition 8.5 (Lazy Evaluation Contexts) $\mathcal{L}::=\bullet\mid\mathcal{L}\{\mathcal{K}\}\mid\mathsf{let}\;x=\mathsf{build}(N_1\mid\cdots\mid N_k)\;\mathsf{in}\;\mathcal{L}$

The reduction rules for codata are given in Figure 8, and are similar to the lazy reduction rules of the cofix operator of dPA$^\omega$ [20]. The way in which these rules achieve lazy evaluation is well explained by [28]: the first rule highlights that, when a coinductive structure is reached in a CBV context, its computation is delayed by abstracting it; the second rule precisely corresponds to when the coinductive structure is linked to x, whose value is needed, so a single evaluation step is performed. The third reduction shows that control operators are able to capture the context of a build statement. The last reduction describes how control operators interact with coinductive structures under let expressions.

Finally, to be able to understand CBV reduction, and to enable implementation of a bidirectional typing algorithm, WHNFs and values for (co)data terms are defined.

Definition 8.6 ((Weak Head) Normal Forms, Values) *i*) Normal Forms.

$$\begin{array}{rll}
N ::= & \cdots\mid D\,\overrightarrow{N}\mid R\,\overrightarrow{N}\mid\mathsf{constr}_i\,\overrightarrow{N} & \\
& \mid\mathsf{build}(N_1\mid\cdots\mid N_k) & (N_i\neq\mu\alpha.M) \\
& \mid\mathsf{proj}_i(N) & (N\neq\mathsf{build}(N_1\mid\cdots\mid N_k),\mu\alpha.M) \\
& \mid\mathsf{case}\;N\rhd z.C\;\mathsf{of}\;(\vec{x}_1.N_1\mid\cdots\mid\vec{x}_k.N_k) & (N\neq\mathsf{constr}_i\,\overrightarrow{N},\mu\alpha.M) \\
& \mid\mathsf{elim}\;N\rhd z.C\;\mathsf{by}\;(\vec{x}_1.N_1\mid\cdots\mid\vec{x}_k.N_k) & (N\neq\mathsf{constr}_i\,\overrightarrow{N},\mu\alpha.M)
\end{array}$$

ii) Weak Head Normal Forms (WHNF).

$$\begin{aligned} h, H ::= & \cdots \mid D\,\overrightarrow{M} \mid R\,\overrightarrow{M} \mid \mathsf{constr}_i\,\overrightarrow{M} \mid \mathsf{build}(M_1 \mid \cdots \mid M_k) \\ & \mid \mathsf{proj}_i(H) & (H \neq \mathsf{build}(N_1 \mid \cdots \mid N_k)) \\ & \mid \mathsf{case}\ H \rhd z.C\ \mathsf{of}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) & (H \neq \mathsf{constr}_i\,\overrightarrow{N}) \\ & \mid \mathsf{elim}\ H \rhd z.C\ \mathsf{by}\ (\vec{x}_1.N_1 \mid \cdots \mid \vec{x}_k.N_k) & (H \neq \mathsf{constr}_i\,\overrightarrow{N}) \end{aligned}$$

iii) Values. $V ::= \cdots \mid D\,\overrightarrow{M} \mid R\,\overrightarrow{M} \mid \mathsf{constr}_i\,\overrightarrow{V} \mid \mathsf{build}(V_1 \mid \cdots \mid V_k)$

9 Implementation

In this section, we discuss the technical details of Candid, a theorem prover that uses ECC$_\mu$ as its core calculus. A full presentation of the syntax can be found in A.4.1, and is reminiscent of Agda's syntax. We will also discuss how users can interact with the type system through a REPL.

(*Term Representation*): The variables are implemented via the `unbound-generics` [23] library, which is a re-implementation of the unbound library outlined in [45]. This gives variables a locally nameless representation, where bound variables have De Bruijn Indices [11], and free variables are nameful. Evaluation is achieved by substitution: for usual substitution, we were able to use `unbound-generics`'s `Subst` class; for the structural substitutions, we developed a similar generic class `StrSubst`, letting us define structural substitutions for each μ-reduction.

(*Type Checking*): We implemented the bidirectional algorithm (Section 7) for ECC$_\mu$. The pipeline for type-checking follows the usual steps: we first lex and parse the given file into the internal data structure for definitions and terms. The function and data definitions are then type checked in sequential order (with respect to the order they were defined in the file), adding the function types, data/codata types and constructors/projectors to a global context when they successfully type check, and exit with an error message if not.

As described in the bidirectional algorithm, we need to make use of an evaluator for both reducing to WHNF, and attempting to reduce a term to NEF. This distinguishes the algorithm from other similar bidirectional systems, as it means the evaluator is used in two different ways; one needs to have to goal of WHNF, that won't bother to reduce subterms when not needed to, the other needs to be more eager and able to exit once the term has reached a NEF form. This was achieved by abstracting the reduction rules as rewrites, and letting the two subevaluators (for WHNF and NEF) call these rewrites and also each other if they need to.

User-level Language

The surface language is very close to that of ECC$_\mu$, although there are simplifications for the user. Functions can be defined with arguments, where the definition `f x y = M` is desugared into `f = \x -> \y -> M`. Multiple variables can be bound at a single lambda or telescope, where `\x y -> M` and `(A B : T)` are desugared into `\x -> \y -> M` and `(A : T)(B : T)`. There are also some shorthands for logical operators, where `!A` and `¬A` are both desugared into `A -> Bot` (where `Bot` represents the type $\bot$), and `A <-> B` to `(A -> B) * (B -> A)`.

(*(Co)Data*): Defining data and codata is very similar to how one would in Agda:

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

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

Note that constructors use telescopes rather than function types in their definition. This is due to the fact that constructors can only be used in canonical form.

Data and record types are handled via pattern matched case and co-pattern matched 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)
```

(*REPL*): Users are able to interact with the type checker through *holes*, indexed by a number. The user can than query the REPL, which reports back the goal and scope at each hole:

```
foo : (A B : Type) -> A -> B -> A
foo T1 T2 x = \y -> ?1

Hole 1:
    Goal: T1
    Scope: {x:T1, y:T2};
```

The type holes let the user engage in a dialog with the type system [34], and enables a hole-driven design workflow, where the type signature of the function guides its construction.

(*Future Features*): To ensure soundness of programs, theorem provers like Agda employ strict positivity checks on (co)data constructor/projector types, and heuristics to ensure program termination. These are both able to be added to the calculus; strict positivity can be achieved by a syntactic check, and a common termination heuristic is that of *structural recursion*, in which recursive functions must always call a subexpression of a given argument in at least one of the recursive arguments. The universe hierarchy is also not currently implemented, so a user is able to write `Type:Type`. Allowing the typical ambiguity in the surface language has well documented implementation methods [18], that are applicable to this language.

Dependently typed languages like Agda enjoy implicit arguments, where the type checker can fill in arguments the user doesn't supply and (co)pattern matched function definitions, which allow functions to have multiple definitions based on the (co)patterns of the arguments/return value. Methods to implementation of both of these features are well documented; algorithms for handling implicit arguments and pattern matching are both explained well in [33].

10 Related Work

(*Calculi with* NEF *Restrictions*): Our work on ECC_μ is based on the calculi dPA^ω, of [20], and $\mathrm{ECC}_\mathrm{K}$of [32]. dPA^ω is a classical proof system for Peano Arithmetic, that is able to prove both the axioms of countable and dependent choice. Countable choice is achieved through a clever use of a coinductive fixpoint, and the intuition that a for-all quantification over a countable domain can be represented by a countable conjunction. For example, a predicate P defined over the natural numbers can be represented by $P(0) \wedge P(1) \wedge P(2) \cdots$. This infinite conjunction is constructed by then building a stream with the coinductive fixpoint.

ECC_K is Luo's Extended Calculus of Constructions equipped with `catch` and `throw` control operators. This calculus is defined via a translation into $\mathrm{L}_{\mathrm{dep}}$; a dependent, classical sequent calculus, capable of CPS translations for dependent types. L_{dep} combines the previous work of the three authors, including, interestingly, a sequent calculus version of dPA^{ω} [30], that was used to show the strong normalisation of dPA^{ω}.

(*PML*$_2$): [24] defines a type system with a similar goal; combining dependent types with control. This system is implemented in the language PML_2 [25]. The key difference when compared to our own system is that Leipgre uses a 'semantic value restriction' instead of the NEF restriction. A term is a semantic value when it can be shown to be observationally equivalent to a syntactic value; the typing rules for semantic values then need a proof of this equivalence. Roughly speaking, when compared with our typing rules, the $\vdash_{\text{NEF}} M$ requirement on M is the same as finding a value V that is equivalent to M, and then making the judgement on M with the equivalency proof in the context; $(M \equiv V) \vdash M$. Due to the basis in equivalency proofs the semantic value restriction is not decideable [29], and is much looser than the NEF requirement. In fact, all NEF terms can be shown to be equivalent to a value, so the semantic value restriction can be understood as a proper superset of NEF terms [29].

PML_2 itself is a dependently typed CBV ML-like language with control operators. This is the most closely related proof assistant to our own, as it is has dependent functions and pairs compatible with classical logic. As it is based on the semantic value model described above, allowing dependent type checking means the system must uses equational reasoning on non-value terms, which is comparable with our method for checking if a non-NEF term is RNEF by evaluating it. As the set of NEF terms is a proper superset of syntactic values, our RNEF evaluator invoked less often than PML_2's equational reasoning in the type checking process.

11 Conclusion

In this paper, we introduced classical logic, how it differs with intuitionistic logic, and its computational content, through the $\lambda\mu$ calculus. Then, we reviewed dependent types and how they relate to first order intuitionistic logic, and that care must be taken when combining them with control operators. We explored the calculi dPA^{ω} and ECC_K, that achieve this safe combination. We presented our calculus ECC_{μ}, which extends ECC_K with dependent coproducts, inductive data and codata, which allows for classical reasoning and dependent types to safely interact. We defined a bidirectional typing algorithm for ECC_{μ}, which successfully turns the type system into a decideable type checking algorithm. Finally, we presented our programming language Candid, based on this calculus, which is a proof assistant for computational classical logic.

We conjecture that we can expand the set of provable propositions in ECC_{μ} by weakening the NEF restriction on terms. By making negation an explicit type (for example, in the $\nu\lambda\mu$-calculus [42]), rather than encoded by '$\to \perp$', this would allow for a less restrictive definition of NEF terms that relates specifically to negation elimination. This would allow ECC_{μ} to prove even more tautologies of classical logic.

The notion of judgemental equality in ECC_{μ} is currently not subscribed to any particular school of equality, like Observational Type Theory (OTT) or Homotopy Type Theory (HoTT). Thus, the notion of equality in ECC_{μ} has great potential to be expanded upon. For example, HoTT has very powerful notions of equality, given by the univalence axiom. This axiom implies that LEM is not true for some types [43], and thus seemingly incompatible with classical logic. It is worth investigating if this is still the case in the presence of the NEF restrictions, and if in fact these characterise the types for which LEM is provable.

We hope this work forms part of the first steps towards a new style of theorem provers that uncover the computational content of classical logic, and the new reasoning power it brings.

References

[1] Z.M. Ariola and H. Herbelin. Minimal Classical Logic and Control Operators. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, and G.J. Woeginger, editors, *Proceedings of Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 871–885. Springer Verlag, 2003.

[2] Z.M. Ariola, H. Herbelin, and A. Sabry. A proof-theoretic foundation of abortive continuations. *High. Order Symb. Comput.*, 20(4):403–429, 2007.

[3] S. van Bakel, F. Barbanera, and U. de'Liguoro. Intersection Types for the $\lambda\mu$-calculus. *Logical Methods in Computer Science*, 141(1), 2018.

[4] S. van Bakel and P. Lescanne. Computation with Classical Sequents. *Mathematical Structures in Computer Science*, 18:555–609, 2008.

[5] S. van Bakel and M.G. Vigliotti. A fully-abstract semantics of lambda-mu in the pi-calculus. In Paulo Oliva, editor, *Proceedings Fifth International Workshop on Classical Logic and Computation, CL&C 2014, Vienna, Austria, July 13, 2014*, volume 164 of *EPTCS*, pages 33–47, 2014.

[6] E.C. Brady. *Practical implementation of a dependently typed functional programming language*. PhD thesis, Durham University, UK, 2005.

[7] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, September 2013.

[8] L.E.J. Brouwer. *Over de Grondslagen der Wiskunde*. PhD thesis, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, 1907.

[9] L.E.J. Brouwer. De onbetrouwbaarheid der logische principes. *Tijdschrift voor Wijsbegeerte*, 2:152–158, 1908.

[10] L.E.J. Brouwer. Unreliability of the Logical Principles. In A. Heyting, editor, *Collected Works 1. Philosophy and Foundations of Mathematics*. North-Holland, Amsterdam, 1975.

[11] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[12] The Coq Development Team. *The Coq Reference Manual, Release 8.13.0*, 2021.

[13] P.-L. Curien and H. Herbelin. The Duality of Computation. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 233–243. ACM, 2000.

[14] P. Dybjer. Inductive Families. *Formal Aspects Comput.*, 6(4):440–465, 1994.

[15] G. Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39(2):176–210 and 405–431, 1935.

[16] Timothy Griffin. A Formulae-as-Types Notion of Control. In F.E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 47–58. ACM Press, 1990.

[17] P. de Groote. On the Relation between the Lambda-Mu-Calculus and the Syntactic Theory of Sequential Control. In F. Pfenning, editor, *Logic Programming and Automated Reasoning, 5th International Conference, LPAR'94, Kiev, Ukraine, July 16-22, 1994, Proceedings*, volume 822 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 1994.

[18] R. Harper and R. Pollack. Type Checking with Universes. *Theor. Comput. Sci.*, 89(1):107–136, 1991.

[19] H. Herbelin. On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2005.

[20] H. Herbelin. A Constructive Proof of Dependent Choice, Compatible with Classical Logic. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 365–374. IEEE Computer Society, 2012.

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

[22] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[23] A. Kliger. unbound-generics: Support for programming with names and binders using GHC Generics. Hackage, 2020.

[24] R. Lepigre. A Classical Realizability Model for a Semantical Value Restriction. In P. Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.

[25] R. Lepigre. PML2: Integrated Program Verification in ML. In *TYPES*, volume 104 of *LIPIcs*, pages

4:1–4:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[26] Z. Luo. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, UK, 1990.

[27] P. Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

[28] É. Miquey. *Réalisabilité classique et effets de bords*. PhD thesis, University of the Republic, Montevideo, Uruguay, 2017.

[29] É. Miquey. A sequent calculus with dependent types for classical arithmetic. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 720–729. ACM, 2018.

[30] É. Miquey. A constructive proof of dependent choice in classical arithmetic via memoization. *CoRR*, abs/1903.07616, 2019.

[31] É. Miquey, X. Montillet, and G. Munch-Maccagnoni. Dependent Type Theory in Polarised Sequent Calculus. Draft, 2020.

[32] É. Miquey, X. Montillet, and G. Munch-Maccagnoni. Dependent Type Theory in Polarised Sequent Calculus. In *TYPES 2020-26th International Conference on Types for Proofs and Programs*, 2020. abstract.

[33] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[34] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

[35] C.-H.L. Ong and C.A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings of the 24th Annual ACM Symposium on Principles Of Programming Languages*, pages 215–227, 1997.

[36] M. Parigot. Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In A. Voronkov, editor, *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

[37] M. Parigot. Classical Proofs as Programs. In *Kurt Gödel Colloquium*, pages 263–276, 1993. Presented at TYPES Workshop, at Băstad, June 1992.

[38] B.C. Pierce. *Types and programming languages*. MIT Press, 2002.

[39] A. Saurin. On the Relations between the Syntactic Theories of $\lambda\mu$-Calculi. In M. Kaminski and S. Martini, editors, *Computer Science Logic, 22nd International Workshop (CSL'08), Bertinoro, Italy*, volume 5213 of *Lecture Notes in Computer Science*, pages 154–168. Springer Verlag, September 16-19 2008.

[40] V. Sjöberg, C. Casinghino, K.Y. Ahn, N. Collins, H.D. Eades III, P. Fu, G. Kimmell, T. Sheard, A. Stump, and S. Weirich. Irrelevance, Heterogeneous Equality, and Call-by-value Dependent Type Systems. In J. Chapman and P.B. Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012*, volume 76 of *EPTCS*, pages 112–162, 2012.

[41] Th. Streicher and B. Reus. Classical logic: Continuation Semantics and Abstract Machines. *Journal of Functional Programming*, 11(6):543–572, 1998.

[42] A.J. Summers. *Curry-Howard Term Calculi for Gentzen-Style Classical Logics*. PhD thesis, Imperial College London, UK, 2008.

[43] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[44] P. Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

[45] S. Weirich, B.A. Yorgey, and T. Sheard. Binders unbound. In M.M.T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 333–345. ACM, 2011.

Appendix A Appendix

Appendix A.1 Proofs

Appendix A.1.1 Lemma 6.3: nef-Substitution and Reduction Closure in ECC$_\mu$

Proof: ($\mathbf{i}$) By induction on the structure of NEF terms. $x \notin fv(M) \Rightarrow M\{N/x\} = m \in \text{NEF}$. This covers terms $y, \langle\rangle, 1, \mathcal{U}_i, 0, \mathsf{refl}$. From now on, we assume $x \in fv(M)$.

(*Base Case*): Assume $n \in$ NEF; then $x\{N/x\} = n \in$ NEF.

(*Inductive Cases*): Assume $p\{N/x\}, q\{N/x\}, M\{N/x\}, A\{N/x\}, B\{N/x\} \in$ NEF. Then:

- $((y{:}A) \to B)\{N/x\} = (y{:}A\{N/x\}) \to B\{N/x\} \in \text{NEF}$
- $(\lambda y.m)\{N/x\} = \lambda y.(M\{N/x\}) \in \text{NEF}$
- $(\mathsf{let}\ y = p\ \mathsf{in}\ q)\{N/x\} = (\mathsf{let}\ y = p\{N/x\}\ \mathsf{in}\ q\{N/x\}) \in \text{NEF}$
- $((y{:}A) \times B)\{N/x\} = (y{:}A\{N/x\}) \times B\{N/x\} \in \text{NEF}$
- $\pi_i(M)\{N/x\} = \pi_i(M\{N/x\}) \in \text{NEF}$
- $(P,Q)\{N/x\} = (P\{N/x\}, Q\{N/x\}) \in \text{NEF}$
- $(A+B)\{N/x\} = (A\{N/x\} + B\{N/x\} \in \text{NEF})$
- $\mathsf{inj}_i(M)\{N/x\} = \mathsf{inj}_i(M\{N/x\}) \in \text{NEF}$
- $(\mathsf{case}\ M\ \mathsf{of}\ (y_1.p, y_2.q))\{N/x\} = \mathsf{case}\ M\{N/x\}\ \mathsf{of}\ (y_1.p\{N/x\}, y_2.q\{N/x\}) \in \text{NEF}$
- $(p =_A q)\{N/x\} = (p\{N/x\} =_A q\{N/x\}) \in \text{NEF}$
- $(\mathsf{subst}\ p\ q)\{N/x\} = \mathsf{subst}\ p\{N/x\}\ q\{N/x\} \in \text{NEF}$

(($\mathbf{ii}$)): 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.

- $\mathsf{let}\ x = \mathsf{inj}_i(p)\ \mathsf{in}\ q \in \text{NEF} \Rightarrow p, q \in \text{NEF} \Rightarrow \mathsf{let}\ y = p\ \mathsf{in}\ q[\mathsf{inj}_i(y)/a] \in \text{NEF}$
- $\pi_i(\mathsf{let}\ x = p\ \mathsf{in}\ q) \in \text{NEF} \Rightarrow p, q \in \text{NEF} \Rightarrow \mathsf{let}\ x = p\ \mathsf{in}\ \pi_i(q) \in \text{NEF}$
- $\mathsf{let}\ x = p\ \mathsf{in}\ q \in \text{NEF} \Rightarrow p, q \in \text{NEF} \Rightarrow p\{N/x\} \in \text{NEF}$
- $\mathsf{case}\ \mathsf{inj}_i(M)\ \mathsf{of}\ (x_1.n_1 | x_2.n_2) \in \text{NEF} \Rightarrow m, p, q \in \text{NEF} \Rightarrow \mathsf{let}\ x = M\ \mathsf{in}\ n_i \in \text{NEF}$
- $\pi_i(m_1, m_2) \in \text{NEF} \Rightarrow m_1, m_2 \in \text{NEF}$
- $\mathsf{subst}\ \mathsf{refl}\ M \in \text{NEF} \Rightarrow m \in \text{NEF}$
- $\mathcal{K}\{\mathsf{let}\ x = M\ \mathsf{in}\ n\} \in$ NEF means that all terms appearing in the context $\mathcal{K}$ are NEF[3], and that $m, n \in \text{NEF} \Rightarrow \mathsf{let}\ x = M\ \mathsf{in}\ \mathcal{K}\{n\} \in \text{NEF}$.

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.

Appendix A.1.2 Lemma 6.4: nef-Term Substitution

Proof by induction on the structure of M. Assume $\Gamma, x{:}C \vdash M : A \mid \Delta$ and $\Gamma \vdash_{\text{NEF}} N : C \mid \Delta$. We write Γ' and Δ' for $\Gamma\{N/x\}$ and $\Delta\{N/x\}$, respectively.

- x: $\quad \Gamma, x{:}C \vdash x{:}C \mid \Delta$ and $\Gamma \vdash C : \mathcal{U}_i \mid \Delta \qquad (Ax)$

  $\Rightarrow \Gamma' \vdash x\{N/x\} : C\{N/x\} \mid \Delta'$,

  $\Gamma' \vdash C\{N/x\} : \mathcal{U}_i \mid \Delta' \qquad$ Induction

  $\Rightarrow \Gamma' \vdash x\{N/x\} : C \mid \Delta'$ and $\Gamma' \vdash C : \mathcal{U}_i \mid \Delta'$ ($x \notin fv(C)$, by (Ax))

  $\Rightarrow \Gamma' \vdash N : C \mid \Delta' \qquad$ Defn

[3] This can be proved with a very simple induction on the definition of $\mathcal{K}$, needing only consider when it is of the form $\mathsf{inj}_i(\mathcal{K}'), (\mathcal{K}', m), (v, \mathcal{K}), \mathsf{case}\ \mathcal{K}'\ \mathsf{of}\ (x_1.n_1 | x_2.n_2), pi_i(\mathcal{K}'), \mathsf{subst}\ \mathcal{K}'\ M, \mathsf{let}\ x = \mathcal{K}'\ \mathsf{in}\ M$.

- y: $\quad \Gamma, y:C \vdash y:C \mid \Delta$ and $\Gamma \vdash C : \mathcal{U}_i \mid \Delta$ $\qquad (Ax)$

$$\begin{array}{lll}
\Rightarrow & \Gamma' \vdash y\{N/x\} : C\{N/x\} \mid \Delta', & \\
 & \Gamma' \vdash C\{N/x\} : \mathcal{U}_i \mid \Delta' & \text{Induction} \\
\Rightarrow & \Gamma' \vdash y\{N/x\} : C\{N/x\} \mid \Delta' & (Ax) \\
\Rightarrow & \Gamma' \vdash y : C\{N/x\} \mid \Delta' & \text{Defn}
\end{array}$$

- $(y{:}M) \to N$ $\quad A = \mathcal{U}_i, \quad \Gamma, x{:}C \vdash m : \mathcal{U}_i \mid \Delta$ and, $\Gamma, x{:}C, y : m \vdash n : \mathcal{U}_i \mid \Delta$ $\qquad (\Pi)$

$$\begin{array}{lll}
\Rightarrow & \Gamma' \vdash M\{N/x\} : \mathcal{U}_i \mid \Delta', & \\
 & \Gamma', y : M\{N/x\} \vdash n\{N/x\} : \mathcal{U}_i \mid \Delta' & \text{Induction} \\
\Rightarrow & \Gamma' \vdash (y{:}M\{N/x\}) \to n\{N/x\} : \mathcal{U}_i \mid \Delta' & (\Pi) \\
\Rightarrow & \Gamma' \vdash ((y{:}M) \to N)\{N/x\} : \mathcal{U}_i \mid \Delta' & \text{Defn}
\end{array}$$

- $\lambda y.M$ $\quad A = (y{:}E) \to F$ and $\Gamma, x{:}C, y : E \vdash m : F \mid \Delta$ $\qquad (\to I)$

$$\begin{array}{lll}
\Rightarrow & \Gamma', y : E\{N/x\} \vdash M\{N/x\} : F\{N/x\} \mid \Delta' & \text{Induction} \\
\Rightarrow & \Gamma' \vdash \lambda y.(M\{N/x\}) : (y{:}E\{N/x\}) \to F\{N/x\} \mid \Delta' & (\to I) \\
\Rightarrow & \Gamma' \vdash (\lambda y.M)\{N/x\} : ((y{:}E) \to F)\{N/x\} \mid \Delta' & \text{Defn}
\end{array}$$

- let $y = P$ in $Q : A$

Non-dependent: $\Gamma, x{:}C \vdash P : B \mid \Delta$ and $\Gamma, x{:}C, y{:}B \vdash Q : A \mid \Delta$ $\qquad (\texttt{let})$

$$\begin{array}{lll}
\Rightarrow & \Gamma' \vdash P\{N/x\} : B\{N/x\} \mid \Delta', & \\
 & \Gamma', y{:}B\{N/x\} \vdash Q\{N/x\} : A\{N/x\} \mid \Delta' & \text{Induction} \\
\Rightarrow & \Gamma' \vdash \text{let } y = P\{N/x\} \text{ in } Q\{N/x\} : A\{N/x\} \mid \Delta' & (\texttt{let}) \\
\Rightarrow & \Gamma' \vdash (\text{let } y = P \text{ in } Q)\{N/x\} : A\{N/x\} \mid \Delta' & \text{Defn}
\end{array}$$

Dependent: $A = A'[P/y]\Gamma, x{:}C \vdash P : B \mid \Delta,$

$$\begin{array}{lll}
 & \Gamma, x{:}C, y{:}B \vdash_{\textsc{nef}} Q : A' \mid \Delta & (\texttt{let}^d) \\
\Rightarrow & \Gamma' \vdash P\{N/x\} : B \mid \Delta', & \\
 & \Gamma', y{:}B \vdash_{\textsc{nef}} Q\{N/x\} : A'\{N/x\} \mid \Delta' & \text{Induction} \\
\Rightarrow & \Gamma' \vdash \text{let } y = P\{N/x\} \text{ in } Q\{N/x\} : A'\{N/x\}\{(M\{N/x\})/y\} \mid \Delta' & (\texttt{let}^d) \\
\Rightarrow & \Gamma' \vdash (\text{let } y = P \text{ in } Q)\{N/x\} : A'[P/y]\{N/x\} \mid \Delta' & \text{Defn}
\end{array}$$

- PQ Non-dependent: $\Gamma, x{:}C \vdash P : B \to A \mid \Delta$ and $\Gamma, x{:}C \vdash Q : B \mid \Delta$ $\qquad (\to E)$

$$\begin{array}{lll}
\Rightarrow & \Gamma' \vdash P\{N/x\} : (B \to A)\{N/x\} \mid \Delta', & \\
 & \Gamma' \vdash Q\{N/x\} : B\{N/x\} \mid \Delta' & \text{Induction} \\
\Rightarrow & \Gamma' \vdash P\{N/x\} : (B\{N/x\} \to A\{N/x\}) \mid \Delta' & \text{Defn} \\
\Rightarrow & \Gamma' \vdash P\{N/x\}Q\{N/x\} : A\{N/x\} \mid \Delta' & (\to E) \\
\Rightarrow & \Gamma' \vdash (PQ)\{N/x\} : A\{N/x\} \mid \Delta' & \text{Defn}
\end{array}$$

Dependent: $A = A'[n/y], \Gamma, x{:}C \vdash P : (y{:}B) \to A' \mid \Delta,$

$$\begin{array}{lll}
 & \Gamma, x{:}C \vdash_{\textsc{nef}} Q : B \mid \Delta & (\to E^d) \\
\Rightarrow & \Gamma' \vdash P\{N/x\} : ((y{:}B) \to A')\{N/x\} \mid \Delta', & \\
 & \Gamma' \vdash_{\textsc{nef}} Q\{N/x\} : B\{N/x\} \mid \Delta' & \text{Induction} \\
\Rightarrow & \Gamma' \vdash P\{N/x\} : (y{:}B\{N/x\}) \to A'\{N/x\} \mid \Delta' & \text{Defn} \\
\Rightarrow & \Gamma' \vdash P\{N/x\}Q\{N/x\} : A'\{N/x\}[(Q\{N/x\})/y] \mid \Delta' & (\to E^d) \\
\Rightarrow & \Gamma' \vdash (PQ)\{N/x\} : A'[Q/y]\{N/x\} \mid \Delta' & \text{Defn}
\end{array}$$

- $(y{:}P) \times Q$ $\quad A = \mathcal{U}_i, \quad \Gamma, x{:}C \vdash P : \mathcal{U}_i \mid \Delta$ and $\Gamma, x{:}C, y : m \vdash Q : \mathcal{U}_i \mid \Delta$ $\qquad (\Sigma)$

$$\begin{array}{lll}
\Rightarrow & \Gamma' \vdash P\{N/x\} : \mathcal{U}_i \mid \Delta', & \\
 & \Gamma, y : P\{N/x\} \vdash Q\{N/x\} : \mathcal{U}_i \mid \Delta & \text{Induction} \\
\Rightarrow & \Gamma' \vdash (y{:}P\{N/x\}) \times Q\{N/x\} : \mathcal{U}_i \mid \Delta' & (\Sigma) \\
\Rightarrow & \Gamma' \vdash ((y{:}P) \times Q)\{N/x\} : \mathcal{U}_i \mid \Delta' & \text{Defn}
\end{array}$$

- (P,Q) $\quad A=(y{:}A_1)\times A_2,\quad \Gamma,x{:}C\vdash P:A_1\mid\Delta$ and $\Gamma,x{:}C\vdash Q:A_2[P/y]\mid\Delta$ $\quad(\times I)$
  $\Rightarrow \Gamma'\vdash P\{N/x\}:A_1\{N/x\}\mid\Delta',$
  $\Gamma'\vdash Q\{N/x\}:(A_2[(P\{N/x\})/y])\{N/x\}\mid\Delta'$ Induction
  $\Rightarrow \Gamma'\vdash Q\{N/x\}:A_2\{N/x\}[(M\{N/x\})/y]\mid\Delta'$ Defn
  $\Rightarrow \Gamma'\vdash (P\{N/x\},Q\{N/x\}):(y{:}A_1\{N/x\})\times A_2\{N/x\}\mid\Delta'$ $(\times I)$
  $\Rightarrow \Gamma'\vdash (P,Q)\{N/x\}:((y{:}A_1)\times A_2)\{N/x\}\mid\Delta'$ Defn
- $\pi_1(M)$ Non-dependent: $\Gamma,x{:}C\vdash m:A\times B\mid\Delta$ $(\times E_1)$
  $\Rightarrow \Gamma'\vdash M\{N/x\}:(A\times B)\{N/x\}\mid\Delta'$ Induction
  $\Rightarrow \Gamma'\vdash M\{N/x\}:A\{N/x\}\times B\{N/x\}\mid\Delta'$ Defn
  $\Rightarrow \Gamma'\vdash \pi_1(M\{N/x\}:A\{N/x\})\mid\Delta'$ $(\times E_1)$
  $\Rightarrow \Gamma'\vdash (\pi_1(M))\{N/x\}:A\{N/x\}\mid\Delta'$ Defn

  Dependent: $\Gamma,x{:}C\vdash m:(y{:}A)\times B\mid\Delta$ $(\times E_1^d)$
  $\Rightarrow \Gamma'\vdash M\{N/x\}:((y{:}A)\times B)\{N/x\}\mid\Delta'$ Induction
  $\Rightarrow \Gamma'\vdash M\{N/x\}:((y{:}A\{N/x\})\times B\{N/x\})\mid\Delta'$ Defn
  $\Rightarrow \Gamma'\vdash \pi_1(M\{N/x\}:A\{N/x\})\mid\Delta'$ $(\times E_1)$
  $\Rightarrow \Gamma'\vdash (\pi_1(m))\{N/x\}:A\{N/x\}\mid\Delta'$ Defn
- $\pi_2(M)$ Non-dependent: $\Gamma,x{:}C\vdash m:B\times A\mid\Delta$ $(\times E_2)$
  $\Rightarrow \Gamma'\vdash M\{N/x\}:(B\times A)\{N/x\}\mid\Delta'$ Induction
  $\Rightarrow \Gamma'\vdash M\{N/x\}:B\{N/x\}\times A\{N/x\}\mid\Delta'$ Defn
  $\Rightarrow \Gamma'\vdash \pi_2(M\{N/x\}:A\{N/x\})\mid\Delta'$ $(\times E_2)$
  $\Rightarrow \Gamma'\vdash (\pi_2(M))\{N/x\}:A\{N/x\}\mid\Delta'$ Defn

  Dependent: $A=A'[\pi_1(m)/y],\quad \Gamma,x{:}C\vdash m:(y{:}B)\times A'\mid\Delta$ $(\times E_2^d)$
  $\Rightarrow \Gamma'\vdash M\{N/x\}:((y{:}B)\times A')\{N/x\}\mid\Delta'$ Induction
  $\Rightarrow \Gamma'\vdash M\{N/x\}:(y{:}B\{N/x\})\times A'\{N/x\}\mid\Delta'$ Defn
  $\Rightarrow \Gamma'\vdash \pi_2(M\{N/x\}:A'[\pi_1(M\{N/x\})/y])\mid\Delta'$ $(\times E_2)$
  $\Rightarrow \Gamma'\vdash (\pi_2(m))\{N/x\}:A'[\pi_1(m)/y]\{N/x\}\mid\Delta'$ Defn
- $D+E$ $\quad A=\mathcal{U}_i,\quad \Gamma,x{:}C\vdash D:\mathcal{U}_i\mid\Delta$ and $\Gamma,x{:}C\vdash E:\mathcal{U}_i\mid\Delta$ $(+F)$
  $\Rightarrow \Gamma'\vdash D\{N/x\}:\mathcal{U}_i\mid\Delta'$ and $\Gamma\vdash E\{N/x\}\mid\Delta$ Induction
  $\Rightarrow \Gamma'\vdash D\{N/x\}+E\{N/x\}:\mathcal{U}_i\mid\Delta'$ $(+F)$
  $\Rightarrow \Gamma'\vdash (D+E)\{N/x\}:\mathcal{U}_i\mid\Delta'$ Defn
- $\mathsf{inj}_i(M)$: We show for $i=1$; it is almost exactly the same for $i=2$ (as there is no dependency).
  $A=A_1+A_2,\quad \Gamma,x{:}C\vdash M:A_1\mid\Delta$ and $\Gamma,x{:}C\vdash A_i:\mathcal{U}_i\mid\Delta$ $(+I)$
  $\Rightarrow \Gamma'\vdash M\{N/x\}:A_1\{N/x\}\mid\Delta',$
  $\Rightarrow \Gamma'\vdash A_i\{N/x\}:\mathcal{U}_i\mid\Delta'$ Induction
  $\Rightarrow \Gamma'\vdash \mathsf{inj}_1(M\{N/x\}):A_1\{N/x\}+A_2\{N/x\}\mid\Delta'$ $(+I)$
  $\Rightarrow \Gamma'\vdash (\mathsf{inj}_1(M))\{N/x\}:(A_1+A_2)\{N/x\}\mid\Delta'$ Defn
- case $M\triangleright z.P$ of $(x_1.N_1|x_2.N_2)$

Non-dependent: $M = A, \quad \Gamma, x{:}C \vdash m : B_1 + B_2 \mid \Delta,$
$\Gamma, x{:}C, x_i{:}B_i \vdash N_i : A \mid \Delta,$ $(+E)$
$\Gamma, x{:}C \vdash A : \mathcal{U}_i \mid \Delta$

$\Rightarrow$ $\Gamma' \vdash M\{N/x\} : (B_1 + B_2)\{N/x\} \mid \Delta',$
$\Gamma', x_i{:}B_i\{N/x\} \vdash N_i\{N/x\} : (A\{N/x\}) \mid \Delta',$ (Induction)
$\Gamma' \vdash A\{N/x\} : \mathcal{U}_i \mid \Delta'$

$\Rightarrow$ $\Gamma' \vdash M\{N/x\} : B_1\{N/x\} + B_2\{N/x\} \mid \Delta'$ (Defn)

$\Rightarrow$ $\Gamma\{N/x\} \vdash$
$\mathsf{case}\ M\{N/x\} \rhd z.(A\{N/x\})\ \mathsf{of}\ (x_1.(N_1\{N/x\}) \mid x_2.(N_2\{N/x\})) : A\{N/x\}$
$\mid \Delta\{N/x\}$ $(+E)$

$\Rightarrow$ $\Gamma' \vdash (\mathsf{case}\ M \rhd z.A'\ \mathsf{of}\ (x_1.N_1 | x_2.N_2))\{N/x\} : A\{N/x\} \mid \Delta'$ Defn

Dependent: $A = A'[m/z], \quad \Gamma, x{:}C \vdash m : B_1 + B_2 \mid \Delta,$
$\Gamma, x{:}C, x_i{:}B_i \vdash N_i : A[\mathsf{inj}_i(x_i)/x] \mid \Delta,$ $(+E^d)$
$\Gamma x{:}C, z : B_1 + B_2 \vdash M : \mathcal{U}_i \mid \Delta$

$\Rightarrow$ $\Gamma' \vdash M\{N/x\} : (B_1 + B_2)\{N/x\} \mid \Delta',$
$\Gamma', x_i{:}B_i\{N/x\} \vdash N_i\{N/x\} : (A'[\mathsf{inj}_i(x_i)/z]\{N/x\}) \mid \Delta',$ (Induction)
$\Gamma', z : (B_1 + B_2)\{N/x\} \vdash A'\{N/x\} : \mathcal{U}_i \mid \Delta'$

$\Rightarrow$ $\Gamma' \vdash M\{N/x\} : B_1\{N/x\} + B_2\{N/x\} \mid \Delta',$
$\Gamma', x_i{:}B_i\{N/x\} \vdash N_i\{N/x\} : A'\{N/x\}[\mathsf{inj}_i(x_i)/z] \mid \Delta',$ (Defn)
$\Gamma', z : B_1\{N/x\} + B_2\{N/x\} \vdash A'\{N/x\} : \mathcal{U}_i \mid \Delta'$

$\Rightarrow$ $\Gamma\{N/x\} \vdash$
$\mathsf{case}\ M\{N/x\} \rhd z.(A'\{N/x\})\ \mathsf{of}\ (x_1.(N_1\{N/x\}) | x_2.(N_2\{N/x\})) : A'\{N/x\}[m/z]$
$\mid \Delta\{N/x\}$ $(+E^d)$

$\Rightarrow$ $\Gamma' \vdash (\mathsf{case}\ M \rhd z.A'\ \mathsf{of}\ (x_1.N_1 | x_2.N_2))\{N/x\} : A'[m/z]\{N/x\} \mid \Delta'$ Defn

- $m =_B n$ $\quad A = \mathcal{U}_i, \quad \Gamma, x{:}C \vdash m : B \mid \Delta$ and $\Gamma, x{:}C \vdash n : B \mid \Delta$ $(=)$
  $\Rightarrow \Gamma' \vdash M\{N/x\} : \{N/x\}B \mid \Delta',$
  $\Gamma \vdash n\{N/x\} : B\{N/x\} \mid \Delta$ Induction
  $\Rightarrow \Gamma' \vdash M\{N/x\} =_B n\{N/x\} \mid \Delta'$ $(=)$
  $\Rightarrow \Gamma' \vdash (m =_B n)\{N/x\} \mid \Delta'$ Defn
- refl $\quad A = (m =_B m), \quad \Gamma, x{:}C \vdash B : \mathcal{U}_i \mid \Delta$ and $\Gamma, x{:}C \vdash m : B \mid \Delta$ (refl)
  $\Rightarrow \Gamma' \vdash B\{N/x\} : \mathcal{U}_i \mid \Delta' \Gamma' \vdash M\{N/x\} : B\{N/x\} \mid \Delta'$ Induction
  $\Rightarrow \Gamma' \vdash \mathsf{refl}_M\{N/x\} : M\{N/x\} =_{B\{N/x\}} M\{N/x\} \mid \Delta'$ (refl)
  $\Rightarrow \Gamma' \vdash \mathsf{refl}_M\{N/x\} : (m =_B m)\{N/x\} \mid \Delta'$ Defn
- subst $M\ N$ $\quad \Gamma, x{:}C, z : B \vdash A : \mathcal{U}_i \mid \Delta, \Gamma, x{:}C \vdash n : A[p/z] \mid \Delta, \Gamma, x{:}C \vdash m : p = q \mid \Delta$ (subst)
  $\Rightarrow \Gamma', z : B\{N/x\} \vdash A\{N/x\} : \mathcal{U}_i \mid \Delta',$
  $\Gamma' \vdash n\{N/x\} : A[p/z]\{N/x\} \mid \Delta',$ Induction
  $\Gamma' \vdash M\{N/x\} : (p =_B q)\{N/x\} \mid \Delta'$
  $\Rightarrow \Gamma' \vdash n\{N/x\} : A\{N/x\}[(p\{N/x\})/z] \mid \Delta',$
  $\Gamma' \vdash M\{N/x\} : p\{N/x\} =_{B\{N/x\}} q\{N/x\} \mid \Delta'$ Defn
  $\Rightarrow \Gamma' \vdash \mathsf{subst}\ M\{N/x\}\ n\{N/x\} : A\{N/x\}[(q\{N/x\})/z] \mid \Delta'$ (subst)
  $\Rightarrow \Gamma' \vdash (\mathsf{subst}\ M\ N)\{N/x\} : A[q/z]\{N/x\} \mid \Delta'$ Defn

- $\mu\alpha.m \quad \Gamma,x{:}C \vdash m : \bot \mid \alpha : A,\Delta$ (μ)
  - $\Rightarrow \Gamma \vdash M\{N/x\} \mid \alpha : A\{N/x\},\Delta$ Induction
  - $\Rightarrow \Gamma \vdash \mu\alpha.(M\{N/x\}) : A\{N/x\} \mid \Delta$ (subst)
  - $\Rightarrow \Gamma \vdash (\mu\alpha.m)\{N/x\} : A\{N/x\} \mid \Delta$ Defn
- $[\alpha]m \quad A = \bot \quad \Gamma,x{:}C \vdash m : B \mid \Delta$ $(name)$
  - $\Rightarrow \Gamma' \vdash M\{N/x\} : B\{N/x\} \mid \Delta'$ Induction
  - $\Rightarrow \Gamma' \vdash [\alpha](M\{N/x\}) : \bot \mid \alpha : B\{N/x\},\Delta'$ $(name)$
  - $\Rightarrow \Gamma' \vdash ([\alpha]m)\{N/x\} : \bot\{N/x\} \mid \alpha : B\{N/x\},\Delta'$ Defn
- $\langle\rangle$: by (unit), $\Gamma' \vdash \langle\rangle : 1 \mid \Delta' \Rightarrow \Gamma' \vdash \langle\rangle\{N/x\} : 1\{N/x\} \mid \Delta'$
- 1: by (1), $\Gamma' \vdash 1 : \mathcal{U}_i \mid \Delta' \Rightarrow \Gamma' \vdash 1\{N/x\} : \mathcal{U}_i\{N/x\} \mid \Delta'$
- 0: by (0), $\Gamma' \vdash 0 : \mathcal{U}_i \mid \Delta' \Rightarrow \Gamma' \vdash 0\{N/x\} : \mathcal{U}_i\{N/x\} \mid \Delta'$
- $\mathcal{U}_i$: by $(\mathcal{U}_i)$, $\Gamma' \vdash \mathcal{U}_i : \mathcal{U}_{i+1} \mid \Delta' \Rightarrow \Gamma' \vdash \mathcal{U}_i\{N/x\} : \mathcal{U}_{i+1}\{N/x\} \mid \Delta'$

Appendix A.1.3 Lemma 6.5: Subject Reduction

Proof by induction on reductions. For each reduction $M \to N$, assume $\Gamma \vdash M : A \mid \Delta$.

- $(\lambda x.m)n \to \mathsf{let}\ x = N \ \mathsf{in}\ M$

  Non-dependent: $\Gamma \vdash \lambda x.M : B \to A \mid \Delta$ and $\Gamma \vdash N : B \mid \Delta$ $(\to E)$
  - $\Rightarrow \Gamma,x{:}B \vdash M : A \mid \Delta$ $(\to I)$
  - $\Rightarrow \Gamma \vdash \mathsf{let}\ x = N \ \mathsf{in}\ M : A \mid \Delta$ $(\mathtt{let})$

  Dependent: $A = A'\{N/x\}\Gamma \vdash \lambda x.M : (x{:}B) \to A' \mid \Delta$ and $\Gamma \vdash_{\text{NEF}} N : B \mid \Delta$ $(\to E^d)$
  - $\Rightarrow \Gamma,x{:}B \vdash M : A' \mid \Delta$ $(\to I)$
  - $\Rightarrow \Gamma \vdash \mathsf{let}\ x = N \ \mathsf{in}\ M : A'\{N/x\} \mid \Delta$ $(\mathtt{let}^d)$
- $\mathsf{let}\ x = V \ \mathsf{in}\ M \to M\{V/x\}$

  Non-dependent: $\Gamma,x{:}B \vdash M : A \mid \Delta, \quad \Gamma \vdash V : B \mid \Delta$ $(\mathtt{let})$
  - $\Rightarrow \Gamma \vdash M\{V/x\} : A \mid \Delta$ Lemma ??

  Dependent: $A = A'\{V/x\}\Gamma,x{:}B \vdash M : A \mid \Delta, \quad \Gamma \vdash_{\text{NEF}} V : B \mid \Delta$ $(\mathtt{let}^d)$
  - $\Rightarrow \Gamma \vdash M\{V/x\} : A'\{V/x\} \mid \Delta$ Lemma ??
- $\mathcal{K}\{\mathsf{let}\ x = M \ \mathsf{in}\ N\} \to \mathsf{let}\ x = M \ \mathsf{in}\ \mathcal{K}\{N\}$

  Non-dependent: $\Gamma \vdash \mathcal{K}\{\mathsf{let}\ x = M \ \mathsf{in}\ N\} : A \mid \Delta$

  Assume that there is a type B (a subterm of A) such that:

  $\Gamma \vdash \mathsf{let}\ x = M \ \mathsf{in}\ N : B \mid \Delta$.
  - $\Rightarrow$ The hole in $\mathcal{K}$ has type B, and $\Gamma \vdash N : B \mid \Delta$ $(\mathtt{let})$
  - $\Rightarrow \Gamma \vdash \mathcal{K}\{N\} : A \mid \Delta$, as n has the same type as $\bullet$
  - $\Rightarrow \Gamma \vdash \mathsf{let}\ x = M \ \mathsf{in}\ \mathcal{K}\{N\} : A \mid \Delta$ $(\mathtt{let})$

  Dependent: $\Gamma \vdash \mathcal{K}\{\mathsf{let}\ x = M \ \mathsf{in}\ N\} : A \mid \Delta$
  - $\Rightarrow A = A'\{M/x\}$, as the type assignment for $\mathcal{K}\{\mathsf{let}\ x = M \ \mathsf{in}\ N\}$ will at some point use the $\mathtt{let}^d$ rule, which will bind x in a subterm of A.

    Assume that there is a type B (a subterm of A) such that:

    $\Gamma \vdash \mathsf{let}\ x = M \ \mathsf{in}\ N : B \mid \Delta$.
  - $\Rightarrow B = B'\{M/x\}, \quad \Gamma \vdash_{\text{NEF}} M : C \mid \Delta, \quad \Gamma,x{:}C \vdash N : B' \mid \Delta$ $(\mathtt{let}^d)$
  - $\Rightarrow$ The hole in $\mathcal{K}$ has type B', with $x \in fv(B')$
  - $\Rightarrow \Gamma,x{:}C \vdash \mathcal{K}\{N\} : A' \mid \Delta$, as n has the same type as $\bullet$
  - $\Rightarrow \Gamma \vdash \mathsf{let}\ x = M \ \mathsf{in}\ \mathcal{K}\{N\} : A'\{M/x\} \mid \Delta$ $(\mathtt{let}^d)$
- $\mathsf{case}\ \mathsf{inj}_i(M) \triangleright z.C \ \mathsf{of}\ (x_1.N_1|x_2.N_2) \to \mathsf{let}\ x_i = M \ \mathsf{in}\ N_i$

Non-dependent: $A = C\{\mathsf{inj}_i(M)/z\}, \Gamma \vdash \mathsf{inj}_i(M) : B_1 + B_2 \mid \Delta, \quad \Gamma \vdash C : \mathcal{U}_i \mid \Delta,$
$\Gamma, x_i : B_i \vdash N_i : C \mid \Delta$ $(+E)$
$\Rightarrow \Gamma \vdash M : B_i \mid \Delta$ $(+I_i)$
$\Rightarrow \Gamma \vdash \mathsf{let}\ x_i = M\ \mathsf{in}\ N_i : C \mid \Delta$ $(\mathtt{let})$
$\Rightarrow \Gamma \vdash \mathsf{let}\ x_i = M\ \mathsf{in}\ N_i : C \mid \Delta$ (Defn)

Dependent: $A = C\{\mathsf{inj}_i(M)/z\}, \Gamma \vdash_{\text{NEF}} \mathsf{inj}_i(M) : B_1 + B_2 \mid \Delta,$
$\Gamma, z : B_1 + B_2 \vdash C : \mathcal{U}_i \mid \Delta,$ $(+E^d)$
$\Gamma, x_i : B_i \vdash N_i : C\{\mathsf{inj}_i(x_i)/z\} \mid \Delta$
$\Rightarrow \Gamma \vdash_{\text{NEF}} M : B_i \mid \Delta$ $(+I_i)$
$\Rightarrow \Gamma \vdash \mathsf{let}\ x_i = M\ \mathsf{in}\ N_i : (C\{\mathsf{inj}_i(x_i)/z\})\{M/x_i\} \mid \Delta$ $(\mathtt{let}^d)$
$\Rightarrow \Gamma \vdash \mathsf{let}\ x_i = M\ \mathsf{in}\ N_i : C\{\mathsf{inj}_i(M)/z\} \mid \Delta$ (Defn)

- $\pi_1(M_1, M_2) \to M_1$
  Non-dependent: $\Gamma \vdash (M_1, M_2) : A \times B \mid \Delta$ $(\times E_1)$
  $\Rightarrow \Gamma \vdash M_1 : A \mid \Delta$ $(\times I)$
  Dependent: $\Gamma \vdash_{\text{NEF}} (M_1, M_2) : (x{:}A) \times B \mid \Delta$ $(\times E_1^d)$
  $\Rightarrow \Gamma \vdash_{\text{NEF}} M_1 : A \mid \Delta$ $(\times I)$
- $\pi_2(M_1, M_2) \to M_2$
  Non-dependent: $\Gamma \vdash (M_1, M_2) : B \times A \mid \Delta$ $(\times E_2)$
  $\Rightarrow \Gamma \vdash M_2 : A \mid \Delta$ $(\times I)$
  Dependent: $A = A'[\pi_1(M_1, M_2)/x], \quad \Gamma \vdash_{\text{NEF}} (M_1, M_2) : (x{:}B) \times A' \mid \Delta$ $(\times E_2^d)$
  $\Rightarrow \Gamma \vdash_{\text{NEF}} M_2 : A'[\pi_1(M_1, M_2)/x] \mid \Delta$ $(\times I)$
- $\mathsf{subst}\ \mathsf{refl}\ M \to M$
  $A = B\{Q/x\}, \quad \Gamma \vdash \mathsf{refl} : P = Q \mid \Delta, \quad \Gamma \vdash M : B\{P/x\} \mid \Delta$ (subst)
  $\Rightarrow \Gamma \vdash \mathsf{refl} : P = P \mid \Delta$, so Q is syntactically equal to P (refl)
  $\Rightarrow B\{Q/x\} = B\{P/x\} = A$
  $\Rightarrow \Gamma \vdash M : A \mid \Delta$
- $V(\mu\alpha.M) \to \mu\alpha.M\{[\alpha]V \bullet / [\alpha]\bullet\}$
  $\Gamma \vdash V : B \to A \mid \Delta, \quad \Gamma \vdash \mu\alpha.M : B \mid \Delta$ $(\to E)$
  $\Rightarrow \Gamma \vdash M : \bot \mid \alpha : B, \Delta$ (μ)
  $\Rightarrow \Gamma \vdash N : B \mid \Delta$, for each N such that $[\alpha]N$ is a subterm of M (*name*)
  $\Rightarrow \Gamma \vdash VN : A \mid \Delta$ $(\to E)$
  $\Rightarrow \Gamma \vdash [\alpha]VN : \bot \mid \alpha : A, \Delta$ (*name*)
  $\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha]V \bullet / [\alpha]\bullet\} : A \mid \Delta$
- $(\mu\alpha.M)N \to \mu\alpha.M\{[\alpha] \bullet N / [\alpha]\bullet\}$
  $\Gamma \vdash N : B \mid \Delta, \quad \Gamma \vdash \mu\alpha.M : B \to A \mid \Delta$ $(\to E)$
  $\Rightarrow \Gamma \vdash M : \bot \mid \alpha : B \to A, \Delta$ (μ)
  $\Rightarrow \Gamma \vdash P : B \to A \mid \Delta$, for each P such that $[\alpha]P$ is a subterm of M (*name*)
  $\Rightarrow \Gamma \vdash PN : A \mid \Delta$ $(\to E)$
  $\Rightarrow \Gamma \vdash [\alpha]PN : \bot \mid \alpha : A, \Delta$ (*name*)
  $\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha] \bullet N / [\alpha]\bullet\} : A \mid \Delta$
- $\mathsf{let}\ x = \mu\alpha.M\ \mathsf{in}\ N \to \mu\alpha.M\{[\alpha]\mathsf{let}\ x = \bullet\ \mathsf{in}\ N / [\alpha]\bullet\}$

$\Gamma, x{:}B \vdash N : A \mid \Delta, \quad \Gamma \vdash \mu\alpha.M : B \mid \Delta$ (let)
$\Rightarrow \Gamma \vdash M : \bot \mid \alpha : B, \Delta$ (μ)
$\Rightarrow \Gamma \vdash P : B \mid \Delta$, for each P such that $[\alpha]P$ is a subterm of M (*name*)
$\Rightarrow \Gamma \vdash \text{let } x = P \text{ in } N : A \mid \Delta$ (let)
$\Rightarrow \Gamma \vdash [\alpha]\text{let } x = P \text{ in } N : \bot \mid \alpha : A, \Delta$ (*name*)
$\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha] \bullet N / [\alpha]\bullet\} : A \mid \Delta$

- $\mu\alpha.[\alpha]M \to m \qquad (\alpha \notin fn(M))$
  $\Gamma \vdash [\alpha]M : \bot \mid \alpha : A, \Delta$ (μ)
  $\Rightarrow \Gamma \vdash M : A \mid \Delta$ (*name*)
- $[\beta]\mu\delta.m \to M\{\beta/\delta\}$
  So $A : \bot$ by (*name*), and; $\Gamma \vdash \mu\delta.M : B \mid \beta : B, \Delta$ (*name*) We also need that $M[\beta/\delta] : A \Rightarrow$
  $\Rightarrow \Gamma \vdash M : \bot \mid \beta : B, \delta : B, \Delta$ (μ)

$M : A$. As $\beta : B$ and $\delta : B$, then substituting β for δ will not change the type of a term. Indeed, for any named term $[\delta]N$:

$\Gamma \vdash [\delta]N : \bot \mid \delta : B, \beta : B, \Delta$
$\Rightarrow \Gamma \vdash n : B \mid \delta : B, \beta : B, \Delta$ (*name*)
$\Rightarrow \Gamma \vdash [\beta]N : \bot \mid \delta : B, \beta : B, \Delta$ (*name*)

- $\pi_i(\mu\alpha.M) \to \mu\alpha.M[[\alpha]\pi_i(\bullet)/[\alpha]\bullet]$
  $A = A_i \qquad \Gamma \vdash \mu\alpha.M : A_1 \times A_2 \mid \Delta$ ($\times E$)
  $\Rightarrow \Gamma \vdash M : \bot \mid \alpha : A_1 \times A_2, \Delta$ (μ)
  $\Rightarrow \Gamma \vdash N : A_1 \times A_2 \mid \Delta$, for each n such that $[\alpha]N$ is a subterm of m (*name*)
  $\Rightarrow \Gamma \vdash \pi_i(N) : A_i \mid \Delta$ ($\times E$)
  $\Rightarrow \Gamma \vdash [\alpha]\pi_i(N) : \bot \mid \alpha : A_i, \Delta$ (*name*)
  $\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha]V \bullet / [\alpha]\bullet\} : A_i \mid \Delta$
- $\text{inj}_i(\mu\alpha.M) \to \mu\alpha.M\{[\alpha]\text{inj}_i(\bullet)/[\alpha]\bullet\}$
  $A = A_1 + A_2 \qquad \Gamma \vdash \mu\alpha.M : A_i \mid \Delta$ ($+I$)
  $\Rightarrow \Gamma \vdash M : \bot \mid \alpha : A_i, \Delta$ (μ)
  $\Rightarrow \Gamma \vdash N : A_i \mid \Delta$, for each n such that $[\alpha]N$ is a subterm of m (*name*)
  $\Rightarrow \Gamma \vdash \text{inj}_i(N) : A_1 + A_2 \mid \Delta$ ($+I$)
  $\Rightarrow \Gamma \vdash [\alpha]\text{inj}_i(N) : \bot \mid \alpha : A_1 + A_2, \Delta$ (*name*)
  $\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha]V \bullet / [\alpha]\bullet\} : A_1 + A_2 \mid \Delta$
- $(V, \mu\alpha.M) \to \mu\alpha.M[[\alpha](V, \bullet)/[\alpha]\bullet]$
  $A = A_1 \times A_2, \quad \Gamma \vdash V : A_1 \mid \Delta, \quad \Gamma \vdash \mu\alpha.M : A_2 \mid \Delta$ ($\times I$)
  $\Rightarrow \Gamma \vdash M : \bot \mid \alpha : A_2, \Delta$ (μ)
  $\Rightarrow \Gamma \vdash N : A_2 \mid \Delta$, for each N such that $[\alpha]N$ is a subterm of M (*name*)
  $\Rightarrow \Gamma \vdash (V, N) : A_1 \times A_2 \mid \Delta$ ($\times I$)
  $\Rightarrow \Gamma \vdash [\alpha](V, N) : \bot \mid \alpha : A_1 \times A_2, \Delta$ (*name*)
  $\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha](V, \bullet)/[\alpha]\bullet\} : A_1 \times A_2 \mid \Delta$
- $(\mu\alpha.M, N) \to \mu\alpha.M\{[\alpha](\bullet, N)/[\alpha]\bullet\}$
  $A = A_1 \times A_2, \quad \Gamma \vdash \mu\alpha.M : A_1 \mid \Delta, \quad \Gamma \vdash N : A_2 \mid \Delta$ ($\times I$)
  $\Rightarrow \Gamma \vdash M : \bot \mid \alpha : A_1, \Delta$ (μ)
  $\Rightarrow \Gamma \vdash P : A_1 \mid \Delta$, for each P such that $[\alpha]P$ is a subterm of M (*name*)
  $\Rightarrow \Gamma \vdash (P, N) : A_1 \times A_2 \mid \Delta$ ($\times I$)
  $\Rightarrow \Gamma \vdash [\alpha](P, N) : \bot \mid \alpha : A_1 \times A_2, \Delta$ (*name*)
  $\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha](\bullet, N)/[\alpha]\bullet\} : A_1 \times A_2 \mid \Delta$
- $\text{case } \mu\alpha.M \rhd z.A \text{ of } (x_1.N_1 | x_2.N_2) \to \mu\alpha.M[[\alpha]\text{case } \bullet \rhd z.A \text{ of } (x_1.N_1 | x_2.N_2)/[\alpha]\bullet]$

$$\begin{array}{lr}
\quad \Gamma \vdash \mu\alpha.M : A_1 + A_2 \mid \Delta, \quad \Gamma, x_i : A_i \vdash N_i : A \mid \Delta, \quad \Gamma \vdash A : \mathcal{U}_i \mid \Delta & (+E) \\
\Rightarrow \Gamma \vdash M : \bot \mid \alpha : A_1 + A_2, \Delta & (\mu) \\
\Rightarrow \Gamma \vdash P : A_1 + A_2 \mid \Delta, \text{ for each } P \text{ such that } [\alpha]P \text{ is a subterm of } M & (\mathit{name}) \\
\Rightarrow \Gamma \vdash \mathsf{case}\ P \rhd z.A\ \mathsf{of}\ (x_1.N_1 | x_2.N_2) : A \mid \Delta & (+E) \\
\Rightarrow \Gamma \vdash [\alpha]\mathsf{case}\ P \rhd z.A\ \mathsf{of}\ (x_1.N_1 | x_2.N_2) : \bot \mid \alpha : A, \Delta & (\mathit{name}) \\
\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha]\mathsf{case}\ \bullet \rhd z.A\ \mathsf{of}\ (x_1.N_1 | x_2.N_2) / [\alpha] \bullet \} : A \mid \Delta &
\end{array}$$

- $\mathsf{subst}\ (\mu\alpha.M)\ N \to \mu\alpha.M\{[\alpha]\mathsf{subst}\ (\bullet)\ N/[\alpha]\bullet\}$

$$\begin{array}{lr}
\quad A = B\{Q/x\}, \quad \Gamma \vdash \mu\alpha.M : P = Q \mid \Delta, \quad \Gamma \vdash N : B\{P/x\} \mid \Delta & (\mathsf{subst}) \\
\Rightarrow \Gamma \vdash M : \bot \mid \alpha : P = Q, \Delta & (\mathit{name}) \\
\Rightarrow \Gamma \vdash L : P = Q \mid \Delta, \text{ for each } L \text{ such that } [\alpha]L \text{ is a subterm of } M & (\mathit{name}) \\
\Rightarrow \Gamma \vdash \mathsf{subst}\ L\ N : B\{Q/x\} \mid \Delta & (\mathsf{subst}) \\
\Rightarrow \Gamma \vdash [\alpha]\mathsf{subst}\ L\ N : \bot \mid \alpha : B\{Q/x\}, \Delta & (\mathsf{subst}) \\
\Rightarrow \Gamma \vdash \mu\alpha.M\{[\alpha]\mathsf{subst}\ (\bullet)\ N/[\alpha]\bullet\} : B\{Q/x\} \mid \Delta &
\end{array}$$

Appendix A.1.4 Claim: ECC$_\mu$ Consistency

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

$$\begin{aligned}
[\![A + B]\!] &= (b{:}\mathbb{B}) \times (\mathtt{if}\ b\ \mathtt{then}\ A\ \mathtt{else}\ B) \\
[\![\mathsf{inj}_1(M)]\!] &= (\mathtt{true}, M) \\
[\![\mathsf{inj}_2(M)]\!] &= (\mathtt{false}, M) \\
[\![\mathsf{case}\ M \rhd z.A\ \mathsf{of}\ (x_1.n_1 | x_2.n_2)]\!] &= \mathsf{if}\ \pi_1([\![M]\!]) \\
&\qquad \mathsf{then}\ (\mathsf{let}\ x_1 = \pi_2([\![M]\!])\ \mathsf{in}\ N_1) \\
&\qquad \mathsf{else}\ (\mathsf{let}\ x_2 = \pi_2([\![M]\!])\ \mathsf{in}\ N_2) \\
[\![\mu\alpha.[\beta]M]\!] &= \mathsf{catch}_\alpha\ \mathsf{throw}_\beta\ M \\
[\![M]\!] &= [\![\cdot]\!] \text{ applied recursively to the subterms of } M
\end{aligned}$$

With the reverse translation given by:

$$\begin{aligned}
[\![\mathbb{B}]\!] &= 1 + 1 \\
[\![\mathtt{true}]\!] &= \mathsf{inj}_1(\langle\rangle) \\
[\![\mathtt{false}]\!] &= \mathsf{inj}_2(\langle\rangle) \\
[\![\mathtt{if}\ b\ \mathtt{then}\ M\ \mathtt{else}\ N]\!] &= \mathsf{case}\ [\![b]\!] \rhd z.[\![\mathbb{B}]\!]\ \mathsf{of}\ ([\![M]\!] \mid [\![N]\!]) \\
[\![\mathsf{catch}_\alpha\ M]\!] &= \mu\alpha.[\alpha]M \\
[\![\mathsf{throw}_\alpha\ M]\!] &= \mu\_.[\alpha]M \\
[\![M]\!] &= [\![\cdot]\!] \text{ applied recursively to the subterms of } M
\end{aligned}$$

We then use the fact that ECC$_\mathrm{K}$ is consistent [31].

Appendix A.2 Type Systems

Appendix A.2.1 ECC$_\mu$ Subtyping

Subtyping Rules for ECC$_\mu$ [33, 31] $\dfrac{\Gamma \vdash t : A \mid \Delta \qquad \Gamma \vdash A \leqslant B \mid \Delta}{\Gamma \vdash t : B \mid \Delta}$ $\dfrac{}{\Gamma \vdash \mathcal{U}_i \leqslant \mathcal{U}_{i+1} \mid \Delta}$

$$\frac{\Gamma \vdash A_1 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_2 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_1 \simeq A_2 : \mathcal{U} \mid \Delta \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2 \mid \Delta}{\Gamma \vdash (x{:}A_1) \to B_1 \leqslant (x{:}A_2) \to B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_2 : \mathcal{U} \mid \Delta \qquad \Gamma \vdash A_1 \simeq A_2 : \mathcal{U} \mid \Delta \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2 \mid \Delta}{\Gamma \vdash (x{:}A_1) \times B_1 \leqslant (x{:}A_2) \times B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 \leqslant A_2 \mid \Delta \qquad \Gamma \vdash B_1 \leqslant B_2 \mid \Delta}{\Gamma \vdash A_1 + B_1 \leqslant A_2 + B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash A_2 : \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash A_1 \simeq A_2 : \mathcal{U}_i \mid \Delta}{\Gamma \vdash A_1 \leqslant A_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 \leqslant A_2 \mid \Delta \qquad \Gamma \vdash A_2 \leqslant A_3 \mid \Delta}{\Gamma \vdash A_1 \leqslant A_3 \mid \Delta}$$

Appendix A.3 Bidirectional Algorithms for ECC$_\mu$

Appendix A.3.1 Bidirectional Type Assignments

The rules are derived by combining the bidirectional style of [33] with the type system in Figure 6

(*Valid Contexts*):

$$(\cdot): \frac{}{\emptyset \vdash \cdot \mid \emptyset} \quad (Ax): \frac{\Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta \rhd B}{\Gamma, x{:}A \vdash x \Rightarrow A \mid \Delta \rhd x} \quad (\alpha x)\ \frac{\Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash \cdot \mid \alpha{:}A, \Delta}$$

(*Function Introduction/Formation*):

$$(\to I): \frac{\Gamma, x{:}A \vdash M \Leftarrow B \mid \Delta \rhd t}{\Gamma \vdash \lambda x.M \Leftarrow C \mid \Delta \rhd \lambda x.t}\ (C \to_{whnf} (x{:}A) \to B)$$

$$(\Pi): \frac{\Gamma \vdash e_1 \Rightarrow C_1 \mid \Delta \rhd A \quad \Gamma, x{:}A \vdash e_2 \Rightarrow C_2 \mid \Delta \rhd B}{\Gamma \vdash (x{:}e_1) \to e_2 \Rightarrow \mathcal{U}_{i \sqcup j} \mid \Delta \rhd (x{:}A) \to B}\ (C_1 \to_{whnf} \mathcal{U}_i \wedge C_2 \to_{whnf} \mathcal{U}_j)$$

(*Pair Introduction/Formation*):

$$(\times I): \frac{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t \quad \Gamma \vdash N \Leftarrow B[t/x] \mid \Delta \rhd u}{\Gamma \vdash (M,N) \Leftarrow C \mid \Delta \rhd (t,u)}\ (C \to_{whnf} (x{:}A) \times B)$$

$$(\Sigma): \frac{\Gamma \vdash e_1 \Rightarrow C_1 \mid \Delta \rhd A \quad \Gamma, x{:}A \vdash e_2 \Rightarrow C_2 \mid \Delta \rhd B}{\Gamma \vdash (x{:}e_1) \times e_2 \Rightarrow \mathcal{U}_{i \sqcup j} \mid \Delta \rhd (x{:}A) \times B}\ (C_1 \to_{whnf} \mathcal{U}_i \wedge C_2 \to_{whnf} \mathcal{U}_j)$$

(*Coproduct Introduction/Formation*):

$$(+I_1): \frac{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t \quad \Gamma \vdash B \Leftarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash \mathsf{inj}_l\,(M) \Leftarrow C \mid \Delta \rhd \mathsf{inj}_l\,(t)}\ (C \to_{whnf} A + B)$$

$$(+I_2): \frac{\Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta \quad \Gamma \vdash M \Leftarrow B \mid \Delta \rhd t}{\Gamma \vdash \mathsf{inj}_r\,(M) \Leftarrow C \mid \Delta \rhd \mathsf{inj}_r\,(t)}\ (C \to_{whnf} A + B)$$

(*Non-Dependent Elimination*):

$$(\to E): \frac{\Gamma \vdash M \Rightarrow C \mid \Delta \rhd t \quad \Gamma \vdash N \Leftarrow A \mid \Delta \rhd u}{\Gamma \vdash MN \Rightarrow B \mid \Delta \rhd tu}\ (C \to_{whnf} (x{:}A) \to B \wedge x \notin fv(B))$$

$$(let): \frac{\Gamma \vdash M \Rightarrow A \mid \Delta \rhd t \quad \Gamma, x{:}A \vdash N \Rightarrow B \mid \Delta \rhd u}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N \Rightarrow B \mid \Delta \rhd \mathsf{let}\ x = t\ \mathsf{in}\ u}\ (x \notin fv(B))$$

$$(\times E_1): \frac{\Gamma \vdash M \Rightarrow C \mid \Delta \rhd t}{\Gamma \vdash \pi_1(M) \Rightarrow A \rhd \pi_1(t)}\ (C \to_{whnf} (x{:}A) \times B \wedge x \notin fv(B))$$

$$(\times E_2): \frac{\Gamma \vdash M \Rightarrow C \mid \Delta \rhd t}{\Gamma \vdash \pi_2(M) \Rightarrow B \rhd \pi_2(t)}\ (C \to_{whnf} (x{:}A) \times B \wedge x \notin fv(B))$$

$$(+E): \frac{\Gamma \vdash M \Rightarrow D \mid \Delta \rhd t \quad \Gamma \vdash C \Rightarrow \mathcal{U}_i \mid \Delta \rhd E \quad \Gamma, x{:}A \vdash N_1 \Rightarrow E \mid \Delta \rhd u_1 \quad \Gamma, y{:}B \vdash N_2 \Rightarrow E \mid \Delta \rhd u_2}{\Gamma \vdash \mathsf{case}\ M \rhd z.C\ \mathsf{of}\ (x.N_1 \mid y.N_2) \Rightarrow E \rhd \mathsf{case}\ t \rhd z.E\ \mathsf{of}\ (x.u_1 \mid y.u_2)}\ (D \to_{whnf} A + B \wedge z \notin f$$

(*Dependent Elimination*):

$$(\to E^d) : \frac{\Gamma \vdash M \Rightarrow C \mid \Delta \rhd t \quad \Gamma \vdash N \Leftarrow A \mid \Delta \rhd u}{\Gamma \vdash MN \Rightarrow B[N/x] \mid \Delta \rhd tu} \; (C \to_{whnf} (x{:}A) \to B)$$

$$(let^d) \; \frac{\Gamma \vdash M \Rightarrow A | \Delta \rhd t \quad \Gamma, x{:}A \vdash N \Rightarrow B \mid \Delta \rhd u}{\Gamma \vdash \mathsf{let}\; x = M \;\mathsf{in}\; N \Rightarrow B[M/a] \mid \Delta \rhd \mathsf{let}\; x = t \;\mathsf{in}\; u}$$

$$(\times E^d_1) : \frac{\Gamma \vdash M \Rightarrow C \mid \Delta \rhd t}{\Gamma \vdash \pi_1(M) \Rightarrow A \mid \Delta \pi_1(t)} \; (C \to_{whnf} (x{:}A) \times B)$$

$$(\times E_2) : \frac{\Gamma \vdash M \Rightarrow C \mid \Delta \rhd t}{\Gamma \vdash \pi_2(M) \Rightarrow B[\pi_1(M)/x] \mid \Delta \rhd \pi_2(t)} \; (C \to_{whnf} (x : A) x B)$$

$$(+E) : \frac{\Gamma \vdash M \Rightarrow D \mid \Delta \rhd t \quad \Gamma, z{:}A + B \vdash C \Rightarrow \mathcal{U}_i \mid \Delta \rhd E \quad \begin{array}{c} \Gamma, x_1{:}A \vdash N_1 \Rightarrow E\{\mathsf{inj}_1(N_1)/z\} \mid \Delta \rhd u_1 \\ \vdots \; \Gamma, x_2{:}B \vdash N_2 \Rightarrow E\{\mathsf{inj}_2(N_2)/z\} \mid \Delta \rhd u_2 \end{array}}{\Gamma \vdash \mathsf{case}\; M \rhd z.C \;\mathsf{of}\; (x.N_1 \mid y.N_2) \Rightarrow C[M/z] \mid \Delta \rhd \mathsf{case}\; t \rhd z.E \;\mathsf{of}\; (x_1.u_1 \mid x_2.u_2)} \; (D \to_{whnf} A + B)$$

(*NEF*):

$$(\textsc{nef}I) : \frac{\Gamma \vdash M \Rightarrow A \mid \Delta \rhd t}{\Gamma \vdash M \Rightarrow A \mid \Delta \rhd t} \; (M \in \textsc{nef}) \quad (\textsc{nef}E) : \frac{\Gamma \vdash M \Rightarrow A \mid \Delta \rhd t}{\Gamma \vdash M \Rightarrow A \mid \Delta \rhd t}$$

$$(\textsc{nef}I) : \frac{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t}{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t} \; (M \in \textsc{nef}) \quad (\textsc{nef}E) : \frac{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t}{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t}$$

(*Control*):

$$(\mu) : \frac{\Gamma \vdash M \Leftarrow 0 \mid \alpha : A, \Delta \rhd t}{\Gamma \vdash \mu\alpha.M \Leftarrow A \mid \Delta \rhd \mu\alpha.t} \quad (name) : \frac{\Gamma \vdash M \Leftarrow A \mid \Delta \rhd t}{\Gamma \vdash [\alpha]M \Rightarrow 0 \mid \alpha : A, \Delta \rhd [\alpha]t} \quad (\mathsf{tp}) : \frac{\Gamma \vdash M \Leftarrow 0 | \Delta \rhd t}{\Gamma \vdash [\mathsf{tp}]M \Rightarrow 0 \mid \Delta \rhd [\mathsf{tp}]t}$$

(*Types*):

$$(1) : \Gamma \vdash 1 \Rightarrow \mathcal{U}_i \mid \Delta \rhd 1 \quad (unit) : \frac{}{\Gamma \vdash \langle\rangle \Rightarrow 1 \rhd \langle\rangle} \quad (\mathcal{U}_i) : \frac{}{\Gamma \vdash \mathcal{U}_i \Rightarrow \mathcal{U}_{i+1} \rhd \mathcal{U}_{i+1}}$$

(*Propositions*):

$$(\mathbb{P}) : \frac{}{\Gamma \vdash \mathbb{P} \Rightarrow \mathcal{U}_0 \rhd \mathbb{P}}$$

$$(\Pi_{\mathbb{P}}) : \frac{\Gamma \vdash e_1 \Rightarrow C_1 \mid \Delta \rhd A \quad \Gamma, x{:}A \vdash e_2 \Rightarrow C_2 \mid \Delta \rhd B}{\Gamma \vdash (x{:}e_1) \to e_2 \Rightarrow \mathbb{P} \mid \Delta \rhd (x{:}A) \to B} \; (C_1 \to_{whnf} \mathcal{U}_i \wedge C_2 \to_{whnf} \mathbb{P})$$

(*Equality*):

$$(refl) : \Gamma \vdash p \equiv q \mid \Delta \rhd t \equiv u \Gamma \vdash \mathsf{refl} \Leftarrow p = q \mid \Delta \rhd \mathsf{refl}_{t=u}$$

$$(subst) : \frac{\Gamma \vdash M \Rightarrow p = q \mid \Delta \rhd t \quad \Gamma \vdash N \Rightarrow B[p/x] \mid \Delta \rhd u \quad \Gamma, x{:}A \vdash B \Rightarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash \mathsf{subst}\; M \; n \Rightarrow B[q/x] \mid \Delta \rhd \mathsf{subst}\; t \; u}$$

Appendix A.3.2 Bidirectional Subtyping

Subtyping [33] $\dfrac{A \to_{whnf} A' \qquad B \to_{whnf} B' \qquad \Gamma \vdash A' \leqslant: B' \mid \Delta}{\Gamma \vdash A \leqslant B \mid \Delta}$

Subtyping for Types in whnf $\dfrac{}{\Gamma \vdash \mathcal{U}_i \leqslant: \mathcal{U}_{i+1} \mid \Delta}$ $\dfrac{\Gamma \vdash A_1 \leqslant: A_2 \mid \Delta \qquad \Gamma \vdash B_1 \leqslant: B_2 \mid \Delta}{\Gamma \vdash A_1 + B_1 \leqslant: A_2 + B_2 \mid \Delta}$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \Leftarrow \mathcal{U}_i \mid \Delta \text{ for some } i \qquad \Gamma, x : A_1 \vdash B_1 \leqslant: B_2 \mid \Delta}{\Gamma \vdash (x{:}A_1) \times B_1 \leqslant: (x{:}A_2) \times B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \Leftarrow \mathcal{U}_i \mid \Delta \text{ for some } i \qquad \Gamma, x : A_1 \vdash B_1 \leqslant: B_2 \mid \Delta}{\Gamma \vdash (x{:}A_1) \to B_1 \leqslant: (x{:}A_2) \to B_2 \mid \Delta}$$

$$\frac{\Gamma \vdash A \equiv B \mid \Delta \qquad \Gamma \vdash A \Leftarrow \mathcal{U}_i \mid \Delta \qquad \Gamma \vdash B \Leftarrow \mathcal{U}_i \mid \Delta}{\Gamma \vdash A \leqslant: B \mid \Delta}$$

Appendix A.4 Implementation

Appendix A.4.1 Syntax

Dependently Typed Theorem Prover Syntax

$$
\begin{array}{rcl}
\langle \mathit{name} \rangle & ::= & \text{[unicode letters]} \\
\langle \mathit{var} \rangle & ::= & {}' \mid \langle \mathit{name} \rangle \\
\langle \mathit{term} \rangle, \langle \mathit{type} \rangle & ::= & \langle \mathit{name} \rangle \\
 & \mid & \backslash(\langle \mathit{var} \rangle +) \to \langle \mathit{term} \rangle \\
 & \mid & \backslash(\langle \mathit{var} \rangle : \langle \mathit{type} \rangle) \to \langle \mathit{term} \rangle \\
 & \mid & \langle \mathit{term} \rangle + \\
 & \mid & \backslash \langle \mathit{var} \rangle : \langle \mathit{var} \rangle \backslash \ \langle \mathit{term} \rangle \\
 & \mid & \texttt{inj}(1|2)\ \langle \mathit{term} \rangle \\
 & \mid & \texttt{case-or}\ \langle \mathit{term} \rangle\ \texttt{of}\ (\langle \mathit{var} \rangle . \langle \mathit{term} \rangle | \langle \mathit{var} \rangle . \langle \mathit{term} \rangle) \\
 & \mid & (\langle \mathit{term} \rangle , \langle \mathit{term} \rangle) \\
 & \mid & \texttt{proj}(1|2)\ \langle \mathit{term} \rangle \\
 & \mid & (\langle \mathit{term} \rangle) \\
 & \mid & ?[0-9]+ \\
 & \mid & \texttt{Top} \\
 & \mid & <> \\
 & \mid & \texttt{Bot} \\
 & \mid & \langle \mathit{type} \rangle \to \langle \mathit{type} \rangle \\
 & \mid & \langle \mathit{telescope} \rangle \to \langle \mathit{type} \rangle \\
 & \mid & \langle \mathit{type} \rangle\ \pm \langle \mathit{type} \rangle \\
 & \mid & \langle \mathit{type} \rangle\ \times\ \langle \mathit{type} \rangle \\
 & \mid & (\langle \mathit{name} \rangle : \langle \mathit{type} \rangle)\ \times \langle \mathit{type} \rangle \\
 & \mid & (\langle \mathit{type} \rangle) \\
 & \mid & \texttt{case}\ \langle \mathit{term} \rangle\ \texttt{of}\ \langle \mathit{pattTree} \rangle \\
 & \mid & \texttt{elim}\ \langle \mathit{term} \rangle\ \texttt{by}\ \langle \mathit{pattTree} \rangle \\
 & \mid & \texttt{build}\ \langle \mathit{pattTree} \rangle \\
 & \mid & \texttt{Type} \mid \texttt{Prop} \\
 & \mid & \texttt{refl} \\
 & \mid & \texttt{subst}\ \langle \mathit{term} \rangle\ \langle \mathit{term} \rangle \\
 & \mid & \langle \mathit{term} \rangle = \langle \mathit{term} \rangle \\
\langle \mathit{pattTree} \rangle & ::= & (\langle \mathit{var} \rangle + \to \langle \mathit{term} \rangle) * \\
\langle \mathit{decl} \rangle & ::= & \langle \mathit{name} \rangle\ :\ \langle \mathit{type} \rangle \\
 & \mid & \langle \mathit{name} \rangle\ =\ \langle \mathit{term} \rangle \\
 & \mid & \texttt{variable} \langle \mathit{name} \rangle : \langle \mathit{type} \rangle \\
 & \mid & \texttt{data}\ \langle \mathit{name} \rangle\ \langle \mathit{telescope} \rangle : \langle \mathit{telescope} \rangle\ \texttt{where} \\
 & & \quad (\langle \mathit{name} \rangle\ :\ \langle \mathit{telescope} \rangle \to \langle \mathit{type} \rangle) * \\
 & \mid & \texttt{record}\ \langle \mathit{name} \rangle\ \langle \mathit{telescope} \rangle : \langle \mathit{telescope} \rangle\ \texttt{where} \\
 & & \quad (\langle \mathit{name} \rangle\ :\ \langle \mathit{type} \rangle) * \\
\langle \mathit{telescope} \rangle & ::= & (\langle \mathit{name} \rangle + : \langle \mathit{type} \rangle) \langle \mathit{telescope} \rangle \\
 & \mid & \langle \mathit{type} \rangle\ \langle \mathit{telescope} \rangle \\
 & \mid & \epsilon
\end{array}
$$