# Literature Study:<br>Curry-Howard Correspondence for Classical Logic

Hans-Dieter A. Hiep

August 7, 2018

## Contents

## 1 Introduction

This document is one of the results of the Literature Study course 2017–2018, at Vrije Universiteit Amsterdam. The other result was a presentation given on June 15, 2018. This work was supervised by Femke van Raamsdonk. Thank you, Femke!

## 1.1 Motivation

This section is not necessary for the rest of this document. The author is interested in the foundations of computing and concurrency.

To understand and study computation, one considers formal languages. Expressions of formal languages are sets of finite words defined by a grammar. Grammars in general are effective procedures that decide set membership. To understand and study concurrency, one considers formal protocols. Expressions of formal protocols are sets of infinite words defined by a grammar.

Formal languages have semantics. Some methods for defining semantics are: operational semantics, denotational semantics, and reduction semantics. Semantics of formal languages might be defined by formal protocols. Examples of formal languages are the configurations of Turing machines and the terms of $\lambda$-calculus. Examples of formal protocols are the traces of Turing machines and the reduction sequences of $\lambda$-calculus, such that the final configuration of a Turing machine is repeated forever, and the normal form of a $\lambda$-term is repeated forever.

We say an infinite word is rational if we can factorize it into two parts: a finite prefix and a finite suffix, such that the infinite word is the prefix appended to the suffix that is repeated forever. A formal protocol is said to be rational iff there exists a formal language and an interpretation of finite words to infinite words that covers all infinite words of the formal protocol. It is, as if each infinite word was the result of unravelling or unfolding a finite word into an infinite word.

One may illustrate rational infinite words by means of decimal expansion of non-negative rational numbers: every rational number, say $\frac{2}{1}$ or $\frac{1}{2}$ or $\frac{1}{3}$, always consists of a finite prefix, say 2. or $0.5$ or $0.$, and a finite suffix that is repeated forevery, say $0$ or $0$ or $3$ to make $2.00\ldots$ or $0.500\ldots$ or $0.333\ldots$ respectively. Here our finite language is the ordered pair of a natural number and a positive integer, $(2,1)$ for $\frac{2}{1}$ or $(1,2)$ for $\frac{1}{2}$ or $(1,3)$ for $\frac{1}{3}$. The interpretation is the decimal expansion. Whenever we obtain a decimal expansion, we ask ourselves: what is the corresponding ordered pair that unravels into it? In general, not every decimal expansion (being an arbitrary infinite sequence of decimal digits) is a rational number, as witnessed by the existence of real numbers such as $\sqrt{2}$ or $\pi$.

A question that is undecidable in general is the question: is a given formal protocol terminating? The question of termination now is the question whether all infinite words in a formal protocol have an 'acceptable' suffix. For example, a Turing machine terminates if it always reaches a final configuration that is repeated forever. Turing's undecidability theorem of the halting problem demonstrates that this question is undecidable for Turing machines. Thus the question whether a formal protocol is rational is also undecidable. Another related theorem is Gödel's incompleteness theorem. The proofs of these theorems rely on Cantor's diagonalization argument, and is related to showing that real numbers are uncountable.

There exists rational formal protocols. For example, reduction sequences of the simply typed $\lambda$-calculus. Simply typed $\lambda$-calculus is strongly normalizing, that is, every term can be reduced to a normal form in a finite number of reduction steps. The normal form of a simply typed $\lambda$-term is repeated forever. The formal protocol is rational because there exists a formal language (that generates finite reduction sequences) and an interpretation (that unfolds the normal form into an infinite suffix) that covers all infinite sequences of our formal protocol.

The simply typed $\lambda$-calculus is closely related to a constructive logic, called minimal logic or the implicational fragment of intuitionistic propositional logic, as understood by the Curry-Howard correspondence. We shall not only look into term calculi that correspond to constructive logics, but also into term calculi that correspond to classical logic.

The author's research agenda is to gain a deep understanding of how classical logic and concurrency are related. This literature study is however restricted to classical logic only, and we do not explore concurrency any further here.

## 1.2 Outline

The approach in this study is to study the correspondence of classical logic to term calculi.

Logic is closely related to computation, as understood by the correspondence between intuitionistic logic and typed $\lambda$-calculi. See the book "Type Theory and Formal Proof" by Nederpelt and Geuvers for an introduction.

The main point is given here by example: formulas of minimal propositional logic and types of simply typed $\lambda$-calculus are in a 1-to-1 correspondence, the construction of proofs in minimal propositional logic and the construction of terms in simply typed $\lambda$-calculus are in a 1-to-1 correspondence, and removing detours of proofs in minimal logic and $\beta$-reduction of $\lambda$-terms are in a 1-to-1 correspondence.

We describe proof systems, natural deduction and sequent calculus. As the study progressed, the author has found the need to explain how proof systems can be viewed in general. These three sections are *not* intended as a very precise, complete or definite description of proof systems. Instead, a simple message is conveyed: sequent calculus is a formalization of derivability in natural deduction. This message seems to fit with the original reasons sequent calculus was developed. (Section 2)

We then consider $\lambda\mathcal{C}$-calculus and its relation to classical propositional logic. The terms of $\lambda\mathcal{C}$-calculus are in a 1-to-1 correspondence to the proofs of classical propositional logic in natural deduction. However, we discuss a major drawback of this calculus, that is the lack of type preservation, and an ugly trick for reparation. Although given in Section 3, this topic was actually studied last during this literature study. This topic was also presented on June 15, 2018.

We consider $\mu\tilde{\mu}$-calculus, which is similar to the calculus of the previous section, but has a proof theory based on a focused sequent calculus. It is a calculus extensible by user-defined data types, say as in ML. It allows the expression of pattern matching on data types, and copattern matching on codata types. This calculus is no longer confluent, but that can be fixed by enforcing a strategy. It does, however, have the property of type preservation. See Section 4.

We finally give an overview of the studied material, from a historical perspective, in Section 5.

# 2 Proof systems

We give a rough sketch of the intuition that sequent calculus formalizes derivability in natural deduction. A proof system consists of a set of objects and a set of inference rules. The set of objects is assumed to have a syntactical structure—say formulas or sequences—defined by a grammar. An inference rule relates finite subsets of objects to finite subsets of objects. Rules are schematically defined, that is, the rule schema contains meta-variables that can be instantiated by concrete objects to obtain a concrete inference rule.

Rule schemas are depicted as below.

$$\frac{a_1 \quad \dots \quad a_n}{c_1 \quad \dots \quad c_m}$$

where $n$ and $m$ are non-negative integers. Above the line, each $a_1$ up to $a_n$ is called an assumption. Below the line, each $c_1$ up to $c_m$ is called a conclusion. Assumptions and conclusions are objects, possibly containing meta-variables. An inference rule is called an axiom if its set of assumptions is empty.

We consider proof systems that have precisely one conclusion for each rule. A derivation of such a proof system is a labelled tree. Each vertex in the tree is labeled by a concrete inference rule. Given a vertex and one of its children, if any, then the assumption of the label of the vertex is the same object as the conclusion of the label of its child. The number of children of the vertex corresponds to the number of assumptions of the rule it is labeled by.

## 2.1 Natural deduction

Natural deduction is understood as a proof system in which the objects are formulas. We first consider the implicational fragment of formulas, called minimal logic. Later we extend it to propositional classical logic.

The grammar of formulas of minimal logic is given below:

$$\phi, \psi ::= P \mid (\phi \to \psi)$$

An atomic proposition, denoted by an uppercase roman letter such as $P$, is a formula. Formulas are closed under implication, that is, given formulas $\phi$ and $\psi$ we can construct the formula $(\phi \to \psi)$. The use of parentheses is conventional, and $\to$ associates to the right.

The rules of minimal logic are given below:

$$\frac{}{\phi}\ ax \qquad \frac{(\phi \to \psi) \quad \phi}{\psi}\ mp \qquad \frac{\psi}{(\phi \to \psi)}\ ii$$

The proof system of minimal natural deduction has three rules. The first is an axiom: a rule without assumptions and as conclusion some formula $\phi$. The second is the modus ponens rule. It has as assumptions formulas $(\phi \to \psi)$ and $\phi$, and as conclusion formula $\psi$. The third is the implication introduction rule. It has as assumption $\psi$ and as conclusion $(\phi \to \psi)$.

We consider the notion of open and closed assumptions. The set of open assumptions is defined inductively over the tree structure of derivations.

A derivation tree with one vertex labeled by an axiom has as set of open assumption, the singleton set containing the conclusion of the rule. Given a derivation tree with the modus ponens rule at the root, the set of its open assumptions is defined recursively: the set of open assumptions of the whole tree is the union of its two subderivations. Given a derivation tree with implication introduction at the root and $(\phi \to \psi)$ as conclusion, the of set of its open assumptions is that of the subderivation but without $\phi$.

We define the derivability relation $\vdash$ that relates a set of formulas to a single formula. Let $\Gamma$ denote a set of formulas. We define that $\Gamma \vdash \phi$ holds iff there exists a derivation tree such that its set of open premises is $\Gamma$ and the conclusion of the rule at the root is $\phi$. We do not write $\{\}$ or commas when considering the set $\Gamma$. Examples are: $P \vdash P$ and $\vdash P \to P$.

The formulas of classical logic are: $P, Q$ for atomic propositions, $\bot$ for falsehood, and closure under implication.

$$\phi, \psi ::= P \mid \bot \mid (\phi \to \psi)$$

Minimal logic plus the axiom schema of double negation elimination results in a logic equivalent to classical logic [**?**]. The proof system of classical natural deduction has all the rules from minimal logic, and in addition the axiom schema with as conclusion $((\phi \to \bot) \to \bot) \to \phi$.

$$\frac{}{\phi}\ ax \qquad \frac{}{(((\phi \to \bot) \to \bot) \to \phi)}\ dn \qquad \frac{(\phi \to \psi) \quad \phi}{\psi}\ mp \qquad \frac{\psi}{(\phi \to \psi)}\ ii$$

Open and closed assumptions are defined as before, with the addition that, the set of open assumptions for the double negation elimination axiom is empty. We adapt the definition of the derivability relation $\vdash$ accordingly. For example, it holds that $\vdash ((P \to Q) \to P) \to P$.

We let $\neg\phi$ be an abbreviation of $\phi \to \bot$. We abbreviate $((\phi \to \psi) \to \psi)$ as $\phi \vee \psi$ and abbreviate $\neg(\neg\phi \vee \neg\psi)$ as $\phi \wedge \psi$. We now have the conventional connectives of classical propositional logic. In addition to the given rule schemas, we also have that the conventional rule schemas of classical logic are derivable (omitted for natural deduction for brevity, but shown in the next section).

Now, the derivability relation $\vdash$ in classical logic is an interesting object of study itself. For example, we could prove that $\vdash \phi \to \psi$ holds iff $\phi \vdash \psi$ holds.

## 2.2 Sequent calculus

A main drawback of the proof system for natural deduction is the implicit treatment of open assumptions. We first consider a formalization of the derivability relation $\vdash$.

This new proof system has as objects sequents. Let $\Gamma$ be a sequence of formulas of classical logic separated by commas. To avoid confusion, we consider sequents within square brackets. Let $S$ be a sequent, then the grammar of the objects of our new proof system is:

$$\begin{aligned} \phi, \psi, \chi &::= P \mid \bot \mid (\phi \to \psi) \\ \Gamma &::= \emptyset \mid \Gamma, \phi \\ S &::= [\Gamma \vdash \phi] \end{aligned}$$

Our intention is that $\Gamma$ models a set. By $\Gamma_1, \phi, \psi, \Gamma_2$ we mean that $\phi, \psi$ are arbitrary two successive elements within the sequence. We consider the following rule schemas:

$$\frac{[\Gamma, \phi, \phi \vdash \chi]}{[\Gamma, \phi \vdash \chi]} \qquad \frac{[\Gamma_1, \phi, \psi, \Gamma_2 \vdash \chi]}{[\Gamma_1, \psi, \phi, \Gamma_2 \vdash \chi]}$$

We also call $\Gamma$ (a set of) assumptions. We abbreviate $\emptyset, \phi$ simply to $\phi$. For each of the rule schemas of classical logic, we also have a rule schema:

$$\frac{}{[\Gamma, \phi \vdash \phi]} \quad \frac{}{[\Gamma \vdash (\neg\neg\phi \to \phi)]} \quad \frac{[\Gamma \vdash (\phi \to \psi)] \quad [\Gamma \vdash \phi]}{[\Gamma \vdash \psi]} \quad \frac{[\Gamma, \phi \vdash \psi]}{[\Gamma \vdash (\phi \to \psi)]}$$

Using these rule schemas we can derive sequents. We write $\Rightarrow [\Gamma \vdash \phi]$ if a derivation with $[\Gamma \vdash \phi]$ as conclusion at its root exists. We have that a sequent $[\Gamma \vdash \phi]$ is derivable in sequent calculus if and only if $\Gamma \vdash \phi$ is derivable in natural deduction.

**Proposition 1.** $\Rightarrow [\Gamma \vdash \phi]$ *if and only if* $\Gamma \vdash \phi$.

Our new proof system formalizes open assumptions explicitly. However, remark that the derivability relation relates sets of formulas to a single formula—an asymmetry that can be removed. We shall now consider a set of formulas related to a set of formulas. This brings us to sequent calculus:

$$\begin{aligned} \phi, \psi &::= P \mid \bot \mid (\phi \to \psi) \\ \Gamma, \Delta &::= \emptyset \mid \Gamma, \phi \\ S &::= [\Gamma \vdash \Delta] \end{aligned}$$

Again we have that $\Gamma$ and $\Delta$ models sets:

$$\frac{[\Gamma, \phi, \phi \vdash \Delta]}{[\Gamma, \phi \vdash \Delta]} \quad \frac{[\Gamma_1, \phi, \psi, \Gamma_2 \vdash \Delta]}{[\Gamma_1, \psi, \phi, \Gamma_2 \vdash \Delta]} \quad \frac{[\Gamma \vdash \Delta, \phi, \phi]}{[\Gamma \vdash \Delta, \phi]} \quad \frac{[\Gamma \vdash \Delta_1, \phi, \psi, \Delta_2]}{[\Gamma \vdash \Delta_1, \psi, \phi, \Delta_2]}$$

We now also have the following rule schemas:

$$\frac{}{[\Gamma, \phi \vdash \Delta, \phi]} \quad \frac{}{[\Gamma, \bot \vdash \Delta]} \quad \frac{[\Gamma \vdash \Delta, \phi] \quad [\Gamma, \psi \vdash \Delta]}{[\Gamma, (\phi \to \psi) \vdash \Delta]} \quad \frac{[\Gamma, \phi \vdash \Delta, \psi]}{[\Gamma \vdash \Delta, (\phi \to \psi)]}$$

We claim that this sequent calculus is isomorphic to the previous natural deduction proof system for classical logic. One may obtain one direction from the fact that all the rules with a single premise on the right are admissible. We show that the so-called weakening rule schema are admissible:

$$\frac{[\Gamma \vdash \Delta]}{[\Gamma, \phi \vdash \Delta]} \qquad \frac{[\Gamma \vdash \Delta]}{[\Gamma \vdash \Delta, \phi]}$$

We show this by the following argument. Given a derivation $[\Gamma \vdash \Delta]$, we perform induction on the structure of its derivation and add $\phi$ to the left-side of $\vdash$: as base case, both axioms are still valid. Similar for the right-side of $\vdash$. With these rules in hand, we can derive the following rule schemas:

$$\frac{[\Gamma \vdash \Delta, \phi]}{[\Gamma, \neg\phi \vdash \Delta]} \qquad \frac{[\Gamma, \phi \vdash \Delta]}{[\Gamma \vdash \Delta, \neg\phi]}$$

These are derived by application of the rule for left implication, and the rule for right implication and right weakening. They in turn allow for the derivation of the following axiom schema:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{[\Gamma, \phi \vdash \phi]}}{[\Gamma \vdash \neg\phi, \phi]}}{[\Gamma, \neg\neg\phi \vdash \phi]}}{[\Gamma \vdash (\neg\neg\phi \to \phi)]}$$

We have the derivability of the following rule schema:

$$\cfrac{\cfrac{\cfrac{[\Gamma \vdash \Delta, \phi]}{[\Gamma \vdash \Delta, \phi, \psi]} \quad [\Gamma, \psi \vdash \Delta, \psi]}{[\Gamma, \phi \to \psi \vdash \Delta, \psi]}}{[\Gamma \vdash \phi \vee \psi, \Delta]} \qquad \cfrac{\cfrac{[\Gamma \vdash \Delta, \psi]}{[\Gamma, \phi \to \psi \vdash \Delta, \psi]}}{[\Gamma \vdash \phi \vee \psi, \Delta]} \qquad \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[\Gamma, \phi \vdash \Delta]}{[\Gamma, \phi \vdash \Delta, \neg\psi]}}{[\Gamma \vdash \Delta, \neg\phi, \neg\psi]}}{[\Gamma \vdash \Delta, \neg\phi \vee \neg\psi, \neg\phi \vee \neg\psi]}}{[\Gamma \vdash \Delta, \neg\phi \vee \neg\psi]}}{[\Gamma, \phi \wedge \psi \vdash \Delta]}$$

$$\cfrac{\cfrac{\cfrac{[\Gamma, \phi \vdash \Delta]}{[\Gamma, \phi \vdash \Delta, \psi]}}{[\Gamma \vdash \Delta, \phi \to \psi]} \quad [\Gamma, \psi \vdash \Delta]}{[\Gamma, \phi \vee \psi \vdash \Delta]} \qquad \cfrac{\cfrac{\cfrac{[\Gamma \vdash \Delta, \phi]}{[\Gamma, \neg\phi \vdash \Delta]} \quad \cfrac{[\Gamma \vdash \Delta, \psi]}{[\Gamma, \neg\psi \vdash \Delta]}}{[\Gamma, \neg\phi \vee \neg\psi \vdash \Delta]}}{[\Gamma \vdash \Delta, \phi \wedge \psi]}$$

The only that remains to be shown is admissibility of the modus ponens rule (cut). By induction on the structure of subderivations, we obtain subderivations free of the modus ponens rule. The right subderivation can thus only be obtained by the right rule for implication, with as subderivation $[\Gamma, \phi \vdash \psi]$. We perform induction on this derivation, substituting all occurrences of the axiom $\Gamma, \Gamma', \phi \vdash \Delta, \phi$ to the derivation of $\Gamma \vdash \phi$ weakened to $\Gamma, \Gamma' \vdash \phi, \Delta$ such that $\phi \notin \Gamma, \Gamma'$. This completes one direction.

The other direction is by showing that for each derivable sequent $[\Gamma \vdash \Delta]$ we have a derivation of natural deduction of $[\Gamma \vdash \phi]$ where $\phi$ is a big disjunction of the formulas in $\Delta$. The first two rules are easy: $[\Gamma, \phi \vdash (\bigvee \Delta) \vee \phi]$ is shown by or-introduction and the axiom $[\Gamma, \phi \vdash \phi]$, and $[\Gamma, \perp \vdash (\bigvee \Delta)]$ is shown by the (derivable) ex falso rule.

The third rule is by induction on the structure of subderivations: we may inspect the last or-introduction rules and from that conclude that either $[\Gamma \vdash \delta]$ for one $\delta \in \Delta$ or $[\Gamma \vdash \phi]$ was derived. In the first case, we are done by sufficient introducing as open assumption $(\phi \to \psi)$ by detour. In the second case, $[\Gamma \vdash \phi]$ is derivable and we may replace all occurrences of the axiom for $[\Gamma, \psi \vdash \psi]$ by modus ponens of the open assumption $(\phi \to \psi)$ and with a derivation of $[\Gamma \vdash \phi]$ pasted in.

The last rule is by induction on the structure of the subderivation: we inspect the last or-introduction rules and conclude that either $[\Gamma, \phi \vdash \delta]$ for $\delta \in \Delta$ or $[\Gamma, \phi \vdash \psi]$ was derived. In the last case we close the open assumption $\phi$ and derive $[\Gamma \vdash (\phi \to \psi)]$. In the first case, we use (derivable) law of excluded middle axiom to derive $[\Gamma \vdash \phi \vee \neg\phi]$ and use or-elimination: the branch where $\phi$ is an open assumption we derive $[\Gamma \vdash (\bigvee \Delta)]$ from $[\Gamma, \phi \vdash \delta]$. The branch where $\neg\phi$ is an open assumption, we introduce $[\Gamma, \neg\phi, \phi \vdash \psi]$ that is derived by negation elimination and (derivable) ex falso rule.

In summary, the sequent calculus we have outlined can be thought of as a proof system that formalizes the derivability relation of natural deduction. We have outlined a correctness proof.

## 3 $\lambda\mathcal{C}$-calculus and classical propositional logic

The $\lambda\mathcal{C}$-calculus was given in a future work section of Felleisen's Ph.D. thesis [1], and later revisited in an article by Felleisen and Hieb [2]. We shall here consider the terms, simple types, typing judgment and reduction semantics.

The terms $M, N$ of $\lambda\mathcal{C}$-calculus are defined by the grammar:

$$M, N ::= x \mid (\lambda x.M) \mid (MN) \mid (\mathcal{C}N)$$

where $x$ is a variable, to be bound by abstraction $(\lambda x.M)$. We consider terms equivalent modulo renaming, treat parenthesis conventionally, and associate applications $(MN)$ and $(\mathcal{C}N)$ to the left.

The simple types $\sigma, \tau$ of $\lambda\mathcal{C}$-calculus are defined by the grammar:

$$\sigma, \tau ::= P \mid \perp \mid (\sigma \to \tau)$$

where $P$ is any base type, and $\to$ associates to the right. Again let $\neg\sigma$ be an abbreviation for $(\sigma \to \perp)$. Let $\Gamma$ be a typing context, consisting of pairs of terms and their type $M : \sigma$. What follows is still essentially natural deduction; typing judgments relate a typing context to only a single type. We define the type judgment $\Gamma \vdash M : \sigma$ as follows:

$$\frac{}{\Gamma, M : \sigma \vdash M : \sigma} \qquad \frac{\Gamma \vdash N : \neg\neg\phi}{\Gamma \vdash (\mathcal{C}N) : \sigma}$$

$$\frac{\Gamma \vdash M : (\sigma \to \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma \to \tau)}$$

It was Griffin that made the discovery that $\mathcal{C}$ can be given the type of the axiom schema of double negation elimination [3]. We also define the following abbreviation:

$$(\mathcal{A}N) = (\mathcal{C}(\lambda y.N))$$

for some fresh variable $y$ not occurring in $N$.

Formulas and proofs of classical propositional logic are in a 1-to-1 correspondence with types and terms in $\lambda\mathcal{C}$-calculus, respectively. We shall now consider a reduction relation that involves rewriting only under certain evaluation contexts. Let $V$ define a value, where values are a proper subclass of terms, defined so:

$$V ::= x \mid (\lambda x.M)$$

and let $E$ be evaluation contexts defined by the following grammar:

$$E ::= \Box \mid (VE) \mid (EM)$$

and let $E[N]$ denote the substitution of term $N$ in the hole of $E$:

$$\begin{array}{rcl} \Box[N] & = & N \\ (VE)[N] & = & (V(E[N])) \\ (EM)[N] & = & ((E[N])M) \end{array}$$

We shall define an unusual rewrite relation, so-called contextual rewriting [4]. It can be seen as a cross between operational semantics and reduction semantics. In operational semantics, one defines a transition relation between terms of a certain structure. In reduction semantics, one defines a rewrite relation that is extended to a transition relation by closing it under arbitrary context: rewrite relations are congruence relations.

Here, we define a contextual reduction semantics for $\lambda\mathcal{C}$-calculus. Given a term, we look for a suitable evaluation context such that its hole is assigned a subterm: this is comparable to unification. We define a transition relation between terms; the rewrite step happens deep within the term under an applicative context, but is not necessarily a congruence as in rewriting semantics. In particular, evaluation contexts disallows any rewriting under $\lambda$-abstractions.

We define the contextual rewrite relation $\mapsto$:

$$E[((\lambda x.M)V)] \mapsto E[M[x := V]]$$
$$E[(\mathcal{C}N)] \mapsto (N(\lambda x.(\mathcal{A}E[x])))$$

where $M[x := V]$ is capture-avoiding substitution of $x$ by term $V$. For every non-value, an evaluation context exists. A non-value is either an application $(\mathcal{C}N)$ or an application $(MN)$. The former has $\Box$ as context, the latter depends on $M$. Suppose $M$ is a value, then the result depends on $N$. If $N$ is also a value, then we have $\Box$ as context. If $N$ is a non-value, we recursively consider the evaluation context for $N$, say $E$, and enclose it as $(ME)$. Suppose $M$ is a non-value, then we recursively consider the evaluation context for $M$, say $E$, and enclose it as $(EN)$.

Moreover, evaluation contexts for non-values are unique. This follows from the following comparison at each application: either both subterms are values, or exactly one of them is a value, or neither subterms are a value. In the first case, we must be done by $\square$. In the second case, either left is a value or right is a value, and we must continue in the other subterm. In the last case, we must continue in the left subterm. Since there is never any choice, there could not be more than one evaluation context. Since for every non-value an evaluation context exists, it thus is unique.

A term is in normal form if and only if it is a value. Suppose a term is a value, then none of the two rules of our contextual rewrite relation applies (since a variable or an abstraction are only allowed at the top-level our applicative context must be $\square$, but no rule applies since both of them require applications), hence it is normal. Suppose a term is normal. Then there does not exist any applicative context for which the rules match. If the term is a value we are done, and otherwise it is a non-value and a unique evaluation context must exists which is absurd.

The problem with our contextual rewriting relation is that it does not preserve types. We demonstrate that by the following example:

$$\lambda x.(\lambda y.(\mathcal{C}(\lambda z.(xz)y))) : (\neg A \to \neg B) \to (B \to A)$$

which corresponds to the type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x : (\neg A \to \neg B) \quad \Gamma \vdash z : \neg A}{\Gamma \vdash xz : \neg B} \qquad \Gamma \vdash y : B}{x : (\neg A \to \neg B), y : B, z : \neg A \vdash (xz)y : \bot}}{x : (\neg A \to \neg B), y : B \vdash (\lambda z.(xz)y) : \neg\neg A}}{x : (\neg A \to \neg B), y : B \vdash (\mathcal{C}(\lambda z.(xz)y)) : A}}{x : (\neg A \to \neg B) \vdash \lambda y.(\mathcal{C}(\lambda z.(xz)y)) : B \to A}}{\vdash \lambda x.(\lambda y.(\mathcal{C}(\lambda z.(xz)y))) : (\neg A \to \neg B) \to (B \to A)}$$

where $\Gamma$ abbreviates $x : (\neg A \to \neg B), y : B, z : \neg A$. Now take as variables $f : (\neg A \to \neg B)$ and $b : B$. We have the following reduction of the term:

$$(\lambda x.(\lambda y.(\mathcal{C}(\lambda z.(xz)y)))f)b$$

has applicative context $E = \square b$ and we obtain (from the first rule)

$$(\lambda y.(\mathcal{C}(\lambda z.(fz)y)))b$$

which has applicative context $E = \square$ and we obtain (from the first rule)

$$\mathcal{C}(\lambda z.(fz)b)$$

which also has applicative context $E = \square$ and we obtain (from the second rule)

$$(\lambda z.(fz)b)(\lambda x.(\mathcal{A}x))$$

from which we obtain ($E = \square$, from the first rule)

$$(f(\lambda x.(\mathcal{A}x)))b$$

Under context $f : (\neg A \to \neg B)$ and $b : B$, the type of the first term is $A$. This is easily checked by applying modus ponens twice after our previous derivation, to obtain

$$(\lambda x.(\lambda y.(\mathcal{C}(\lambda z.(xz)y)))f)b : A$$

We next check the type of the term $(f(\lambda x.(\mathcal{A}x)))b$ after our reduction, under the same context. Since $f$ expects $\neg A$ as argument, we have $(\lambda x.(\mathcal{A}x)) : \neg A$ and so $x : A$ and $(\mathcal{A}x) : \bot$. But $\mathcal{A}x$ is an abbreviation for $\mathcal{C}(\lambda y.x)$ and hence $\mathcal{C}(\lambda y.x) : \bot$. From this, we need $(\lambda y.x) : \neg\neg\bot$ and thus $y : \neg\bot$ and $x : \bot$. This contradicts our earlier assumption that $x : A$. For this term there cannot be a type derivation. Hence, there is no type preservation!

This problem is discussed in detail in the literature, and results in the notion of an explicit top-level *prompt*. Consider what happens when one encounters a term $(\mathcal{A}M)$. In effect, we discard our current applicative context and we replace the current proof by $M$. This is demonstrated as follows. Consider just the term $\mathcal{A}M$. Since it is an abbreviation for $\mathcal{C}(\lambda y.M)$ where $y$ is not free in $M$, we can ask what type it has. $\mathcal{C}(\lambda y.M)$ has type $\sigma$ whenever $(\lambda y.M)$ has type $\neg\neg\sigma$, hence $y$ must have type $\neg\sigma$ and $M$ must be $\bot$. Operationally, $\mathcal{A}M$ does something remarkable. We start with $\mathcal{C}(\lambda y.M)$ and it reduces (by $E = \Box$ and the second rule) to $(\lambda y.M)(\lambda y.(\mathcal{A}y))$. Since $y$ does not occur free in $M$, this reduces (by $E = \Box$ and the first rule) to just $M$. This means that for type preservation to hold, $M$ must also be derivable. But since $M$ must be typed by $\bot$, this means that $\bot$ must be derivable: and this is clearly undesirable.

We thus consider an alternative, although ugly, solution to preserve types under contextual rewriting. We adapt the contextual rewrite relation as follows. When given a term $M : \tau$ we shall wrap it in a fresh context: $\mathcal{C}(\lambda o.oM)$. Here $o$, called a prompt, has type $\neg\tau$. Let $F$ be any term that is not of the shape $\mathcal{C}(\lambda o.\ldots)$ and not a value: the first rule wraps all these terms into a context. The second and third rules now all apply under this context. The last rule extracts a value by unwrapping it out its context.

$$\begin{gathered}
F \mapsto \mathcal{C}(\lambda o.oF)\\
\mathcal{C}(\lambda o.E[((\lambda x.M)V)]) \mapsto \mathcal{C}(\lambda o.E[M[x := V]])\\
\mathcal{C}(\lambda o.E[(\mathcal{C}N)]) \mapsto \mathcal{C}(\lambda o.N(\lambda x.(\mathcal{A}E[x])))\\
\mathcal{C}(\lambda o.oV) \mapsto V
\end{gathered}$$

## 4 $\mu\tilde{\mu}$-calculus with generalized connectives

We will formalize a slightly different sequent calculus here that introduces the notion of focus. What we will see is a simplification of the type system in [5]. The sequent calculus presented here is without defining the grammar of formulas: what is considered is called the core $\mu\tilde{\mu}$-calculus. We later introduce formulas and add typing rules that are specific to connectives.

We have three syntactic categories: values, environments, and commands. Environments are also called contexts.

$$\Gamma \vdash \phi \mid \Delta \quad \text{(value)}$$
$$\Gamma \mid \phi \vdash \Delta \quad \text{(context)}$$
$$\Gamma \vdash \Delta \quad \text{(command)}$$

The symbols $\vdash$ and $\mid$ are special separators, part of the sequent. The reason for using the special symbol $\mid$ is because it is different than commas that separate formulas in $\Gamma$ and $\Delta$. $\Gamma$ and $\Delta$ stand for sequences of formulas seperated by commas, and we assume similar structural rules as before allowing us to handle such sequences as sets. The formula $\phi$ above is called the focussed formula: that focussed formula is on the right of $\vdash$ in $\Gamma \vdash \phi \mid \Delta$ and it is on the left of $\vdash$ in $\Gamma \mid \phi \vdash \Delta$. A sequent that has a focussed formula is called a focussed sequent. Hence, the last sequent is unfocussed.

All formulas on the left of $\vdash$ are still called assumptions, including the focussed formula. Similarly, all formulas on the right of $\vdash$ are still called conclusions. The logical interpretation of these sequences are the same: the left forms a big conjunction, the right forms a big disjunction, or similar, that there exists a derivation in classical natural deduction with as open premises the set of assumptions and a conclusion on the right.

We define the following rule schemas on focussed sequents:

$$\frac{\Gamma \vdash \phi \mid \Delta \quad \Gamma \mid \phi \vdash \Delta}{\Gamma \vdash \Delta}\ \text{cut} \qquad \frac{}{\Gamma \mid \phi \vdash \phi, \Delta}\ \text{covar} \qquad \frac{}{\Gamma, \phi \vdash \phi \mid \Delta}\ \text{var}$$

(Found in section 4 of [5].)

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma \mid \phi \vdash \Delta}\ \text{coact} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \mid \Delta}\ \text{act}$$

The cut rule is the only rule that allows one to eliminate a focussed formula, while the other rules introduce a focussed formula: the two rules on the top right close the focussed formula by assumption, the two bottom right rules introduce focus by eliminating a command.

We now turn to a facility to specify the connectives of formulas. What we do here is essentially the same as in Hagino's typed lambda calculus with categorical type constructors [6], but in sequent calculus.

Consider that in programming languages such as ML, providing a facility for user-defined types is important. Types allow one to organize information and processes. We also consider defining data types [7, 8]. A definition of a data type consists of: the name of a type which is defined, free type variables and a set of sequents. The set of sequents may only consist of focussed sequents, and focussed formulas must have the defined type connective at the root of the focussed formula. Such sequents are called the defining clauses of the type definition. One should be careful to distinguish the use of sequents in two different places: sequents as the defining clauses of a type definition, sequents as the objects of our proof system.

The computational intuition behind such clauses can be found in [7]: "assumptions act as inputs and conclusions act as outputs." This intuition will be further developed in the next few sections.

There are two kinds of type definitions: $\mathbf{data}$ and $\mathbf{codata}$. For the first kind, all defining clauses have focus on the right, that is, all defining clauses are value sequents. For the second kind, all defining clauses have focus on the left, i.e. are context sequents.

**Example 2.** Well-known constructions for sets are shown below. Consider the first data type definition. Given two types $A$ and $B$, then the disjoint union $A \oplus B$ is formed by either an element of $A$ or an element of $B$. Following our previous computation intuition: given as input some $A$ then there exists (by definition) some output $A \oplus B$, and similar for the second clause. The other data type definitions have a similar intuition: $A \otimes B$ pairs two elements and 1 is a unit type.

$$\begin{array}{llll}
\mathbf{data}\ A \oplus B & \mathbf{data}\ A \otimes B & \mathbf{data}\ 1 & \mathbf{data}\ 0 \\
\quad A \vdash A \oplus B \mid & \quad A, B \vdash A \otimes B \mid & \quad \vdash 1 \mid & \\
\quad B \vdash A \oplus B \mid & & &
\end{array}$$

Remark that the definition of 0 has no defining clauses. This is valid since a type definition consists of a set of clauses that could possibly be empty. Remember that for data types, all clauses have focus on the right.

**Example 3.** Lesser-known constructions for logical connectives:

$$\begin{array}{llll}
\mathbf{codata}\ A \& B & \mathbf{codata}\ A \parr B & \mathbf{codata}\ \bot & \mathbf{codata}\ \top \\
\quad \mid A \& B \vdash A & \quad \mid A \parr B \vdash A, B & \quad \mid \bot \vdash & \\
\quad \mid A \& B \vdash B & & &
\end{array}$$

Remember that for codata types, all clauses have focus on the left.

It might seem at first counter-intuitive, that both 0 and $\top$ have no defining clauses. The intuition is that a value of 0 is never constructable, whereas $\top$ is an abstract object with no possible observations [7]. The defining clause $\mid \bot \vdash$ signifies that there exists a derivation in natural deduction with its conclusion in the empty set—that is obviously absurd, and we might believe that $\bot$ is false.

**Example 4.** For implication we give the following definition. In addition we define the dual of implication, called difference.

$$\begin{array}{ll}
\mathbf{codata}\ A \to B & \mathbf{data}\ A - B \\
\quad A \mid A \to B \vdash B & \quad A \vdash A - B \mid B
\end{array}$$

User-defined types extend the inference rules of the proof system we consider. We will later employ such extended proof system as a type system for checking whether terms have a valid type. Given a type definition, we have that certain witnesses correspond to the defining clauses. In our proof system each type definition has two associated kinds of inference rules: a left and a right rule. Note that these typing rules were found in [8], but were absent from [7].

Instead of presenting the generalized rules, which in my opinion are confusing, we will show the typing rules by a concrete example. In general we have the following pattern:

For data types, there is precisely one left rule which has, for each clause an assumption, and for each clause there is one right rule. For codata types, there is precisely one right rule which has, for each clause an assumption, and for each clause there is one left rule.

The assumptions of the left rule for data and the right rule for codata are unfocussed, and each assumption sequent per clause corresponds to the additional input and output types of that clause.

Each right rule for data and left rule for codata, per clause has as many assumptions as there are other input and output types.

For the right rules for data and the left rules for codata, the assumptions corresponding to input are focussed on the right, and assumptions corresponding to output are focussed on the left.

So, in our concrete example, formulas are now considered to be constructed by the connectives:

$$\sigma, \tau ::= \top \mid 0 \mid \bot \mid 1 \mid (\sigma \& \tau) \mid (\sigma \oplus \tau) \mid (\sigma ⅋ \tau) \mid (\sigma \otimes \tau) \mid (\sigma \to \tau) \mid (\sigma - \tau)$$

Notice how each connective corresponds to a data type definition or a codata type definition. We consider the calculus consisting of these types to consists of the typing rules as shown below, in addition to the type rules for the core system as given before, which are cut, (co)var, and (co)act.

**Example 5.** The typing rules for all the data definitions of the previous examples are given here. We first see the data type rules of $A \oplus B$, $A \otimes B$, 1, 0, and $A - B$. The codata type rules of $A \& B$, $A ⅋ B$, $\bot$, $\top$, and $A \to B$ are quite similar to those of the data types.

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma \mid A \oplus B \vdash \Delta}\, L_\oplus \qquad \frac{\Gamma \vdash A \mid \Delta}{\Gamma \vdash A \oplus B \mid \Delta}\, R_{\oplus,1} \qquad \frac{\Gamma \vdash B \mid \Delta}{\Gamma \vdash A \oplus B \mid \Delta}\, R_{\oplus,2}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma \mid A \otimes B \vdash \Delta}\, L_\otimes \qquad \frac{\Gamma \vdash A \mid \Delta \quad \Gamma \vdash B \mid \Delta}{\Gamma \vdash A \otimes B \mid \Delta}\, R_\otimes$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \mid 1 \vdash \Delta}\, L_1 \qquad \frac{}{\Gamma \mid 0 \vdash \Delta}\, L_0 \qquad \frac{}{\Gamma \vdash 1 \mid \Delta}\, R_1$$

$$\frac{\Gamma, B \vdash A, \Delta}{\Gamma \mid A - B \vdash \Delta}\, L_- \qquad \frac{\Gamma \mid A \vdash \Delta \quad \Gamma \vdash B \mid \Delta}{\Gamma \vdash A - B \mid \Delta}\, R_-$$

---

$$\frac{\Gamma \mid A \vdash \Delta}{\Gamma \mid A \& B \vdash \Delta}\, L_{\&,1} \qquad \frac{\Gamma \mid B \vdash \Delta}{\Gamma \mid A \& B \vdash \Delta}\, L_{\&,2} \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \& B \mid \Delta}\, R_\&$$

$$\frac{\Gamma \mid A \vdash \Delta \quad \Gamma \mid B \vdash \Delta}{\Gamma \mid A ⅋ B \vdash \Delta}\, L_⅋ \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A ⅋ B \mid \Delta}\, R_⅋$$

$$\frac{}{\Gamma \mid \bot \vdash \Delta}\, L_\bot \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot \mid \Delta}\, R_\bot \qquad \frac{}{\Gamma \vdash \top \mid \Delta}\, R_\top$$

$$\frac{\Gamma \vdash A \mid \Delta \quad \Gamma \mid B \vdash \Delta}{\Gamma \mid A \to B \vdash \Delta}\, L_\to \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B \mid \Delta}\, R_\to$$

In my opinion, we can appreciate two kinds of dualities here. Taking a dual is a syntactic operation that is an involution, i.e. twice the dual results in the same object.

The first duality is swapping conjunction and disjunction. In the typing rules, we see that $L_\oplus$ and $L_\otimes$ are syntactically similar: one has meta-level conjunction with multiple assumptions in the rule, the other has object-level conjunction with multiple assumptions in the sequent. We also see a syntactical similarity between $R_\oplus$ and $R_\otimes$.

The second duality is swapping data and codata. In the typing rules, we see that $L$ and $R$ rules are swapped. For focussed sequents, swap the symbols $|$ and $\vdash$ in assumptions and conclusion of the inference rules. Furthermore, we swap the additional variables on the left and right of $\vdash$ for unfocussed sequents. We swap [$\oplus$ and $\&$], [$\otimes$ and $⅋$] and all other type connectives that were shown side by side in examples 2 and 3.

## 4.1 Term language

In this section we will treat the $\mu\tilde{\mu}$-calculus, and relate it to user-defined types of the previous section. The type definitions we have seen previously will be used to construct grammars of the inhabitant terms of types. Existence of an inhabitant of a user-defined type is justified by one of its defining clauses. The core calculus also consists of primitive terms, which we will see now.

## 4.2 The core $\mu\tilde{\mu}$-calculus

The core $\mu\tilde{\mu}$-calculus has three syntactic categories of terms: values, contexts and commands. These are defined by the following grammar:

$$v ::= x \mid \mu\alpha.c \qquad e ::= \alpha \mid \tilde{\mu}x.c \qquad c ::= \langle v \mid e \rangle$$

note that this just forms the core grammar, and we extend it later on with terms that are the inhabitants of user-defined types:

$$v ::= x \mid \mu\alpha.c \mid \ldots \qquad e ::= \alpha \mid \tilde{\mu}x.c \mid \ldots \qquad c ::= \langle v \mid e \rangle$$

By $x, y, z, \ldots$ we denote value variables of which there are countably many. By $\alpha, \beta, \gamma, \ldots$ we denote context variables, disjoint from value variables, of which there are also countably many. There are two binding constructs: $\mu\alpha.c$ is a value which itself binds an environment variable $\alpha$ in the nested command $c$, and $\tilde{\mu}x.c$ is an environment which binds a value variable. We consider our language up to renaming of bound variables.

Example terms are: $x$, $\alpha$, $\mu\alpha.\langle x \mid \alpha\rangle$, $\tilde{\mu}x.\langle x \mid \alpha\rangle$ and $\langle \mu\alpha.\langle x \mid \alpha\rangle \mid \tilde{\mu}x.\langle x \mid \alpha\rangle\rangle$. A term is closed if all its variable occurrences are bound by a surrounding $\mu$ or $\tilde{\mu}$ term. Think of all possible closed terms of this calculus.

## 4.3 The $\mu\tilde{\mu}$-calculus with user-defined types

Once we have the definition of user-defined types, we also need to extend the syntax of terms. We shall do this again by showing a concrete example. In [8] it is presented in full generality.

**Example 6.** Consider the type definition of functions. We now shall show the syntax of defining clauses that demonstrates the Curry-Howard correspondence: we have formulas/types $F$ and proofs/terms $t$ combined in typing judgements of the form $t : F$. Every unfocussed judgement on the left-hand side of $\vdash$ types a value variable. Every unfocussed judgement on the right types a context variable. For data, each clause introduces a value term, and for codata, each clause introduces a context term. The term typed by the focussed formula contains only variables that occur in the rest of the clause.

$\mathbf{codata}\ A \to B\ \mathbf{where}$
  $x : A \mid A[x, \alpha] : A \to B \vdash \alpha : B$

We abbreviate $A[x, a]$ as $x \cdot \alpha$. We can reconsider the syntactic categories of values and contexts by adding:

$$v ::= \ldots \mid \mu(x \cdot \alpha.c) \qquad e ::= \ldots \mid v \cdot e$$

Introducing a data type involves the following changes:

$\mathbf{data}\ A \otimes B\ \mathbf{where}$
  $x : A, y : B \vdash P(x, y) : A \otimes B \mid$ we abbreviate $P(x, y)$ as just $(x, y)$

$$v ::= \ldots \mid (v, v) \qquad e ::= \ldots \mid \tilde{\mu}[(x, y).c]$$

Having multiple clauses reveals more interesting patterns:

$\mathbf{data}\ A \oplus B\ \mathbf{where}$
  $x : A \vdash L(x) : A \oplus B \mid$
  $x : B \vdash R(x) : A \oplus B \mid$

$\mathbf{codata}\ A \& B$
  $\mid I[\alpha] : A \& B \vdash \alpha : A$
  $\mid J[\alpha] : A \& B \vdash \alpha : B$

$$v ::= \ldots \mid L(v) \mid R(v) \qquad e ::= \ldots \mid \tilde{\mu}[L(x).c \mid R(x).c]$$
$$v ::= \ldots \mid \mu(I[\alpha].c \mid J[\alpha].c) \qquad e ::= \ldots \mid I[e] \mid J[e]$$

The binding construct becomes a (co)pattern matching construct: it now contains more than one nested commands. We will see that, for example, $\langle \mu(I[\alpha].c_1 \mid J[\alpha].c_2) \mid I[e] \rangle$ reduces to $c_1$ with $\alpha$ substituted by $e$: the copattern indicates that we have a choice between $c_1$ and $c_2$ that depends on the observation of the environment. Dually, $\langle R(v) \mid \tilde{\mu}(L(x).c_1 \mid R(x).c_2) \rangle$ reduces to $c_2$ with $x$ substituted by $v$: again we have a choice that depends on the construction of the value.

## 4.4 Typing rules

We observe that the proof rules corresponding to a user-defined types also are related to the typing rules of its introduced terms. The typing rules for the core $\mu\tilde{\mu}$-calculus are first shown. Just as the proof rules are generally applicable, we can always type the core terms using these rules. The typing rules for the parametric $\mu\tilde{\mu}$-calculus is not given in full generality, but we will only consider an example.

We have found these rules in section 4 of [5] and in figure 4 of [8].

$$\frac{\Gamma \vdash v : \phi \mid \Delta \quad \Gamma \mid e : \phi \vdash \Delta}{\langle v \mid e \rangle : (\Gamma \vdash \Delta)}\ Cut$$

$$\frac{}{\Gamma \mid \alpha : \phi \vdash \alpha : \phi, \Delta}\ CoVar \qquad \frac{}{\Gamma, x : \phi \vdash x : \phi \mid \Delta}\ Var$$

$$\frac{c : (\Gamma, x : \phi \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : \phi \vdash \Delta}\ CoAct \qquad \frac{c : (\Gamma \vdash \alpha : \phi, \Delta)}{\Gamma \vdash \mu\alpha.c : \phi \mid \Delta}\ Act$$

**Example 7.** The typing rules for terms of $A \oplus B$ are below. This is very closely related to the proof rules we have seen before for the user-defined type $A \oplus B$.

$$\frac{c_1 : (\Gamma, x : A \vdash \Delta) \quad c_2 : (\Gamma, x : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}(L(x).c_1 \mid R(x).c_2) : A \oplus B \vdash \Delta}\ L_\oplus$$

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash L(v) : A \oplus B \mid \Delta}\ R_{\oplus,1} \quad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash R(v) : A \oplus B \mid \Delta}\ R_{\oplus,2}$$

The application operator $\cdot$ deserves more attention. We repeat the definition of $A \to B$ below, with $x \cdot \alpha$ for $A[x, \alpha]$.

**codata** $A \to B$ **where**
 $x : A \mid x \cdot \alpha : A \to B \vdash \alpha : B$

The function type is a co-data type, since the focussed judgement is on the left. We now have that $\lambda$-abstraction is a derived concept, as corroborated by the following definition:

$$\lambda x.v = \mu(x \cdot \alpha.\langle v \mid \alpha \rangle)$$

From the definition of $A \to B$ above it is apparent that only application between a proof and an context is defined. But application of two proof terms is also a derived concept:

$$v \cdot w = \mu\beta.\langle v \mid w \cdot \beta \rangle$$

Given that $\lambda x.v = \mu(x \cdot \alpha.\langle v \mid \alpha \rangle)$, the typical typing rules for $\lambda$-abstraction and $\lambda$-application are admissible:

$$\frac{\dfrac{\dfrac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma, x : A \vdash v : B \mid \alpha : B, \Delta}\ W \quad \dfrac{}{\Gamma, x : A \mid \alpha : A \vdash \alpha : A, \Delta}\ CoVar}{\langle v \mid \alpha \rangle : (\Gamma, x : A \vdash \alpha : B, \Delta)}\ Cut}{\Gamma \vdash \mu(x \cdot \alpha.\langle v \mid \alpha \rangle) : A \to B \mid \Delta}\ R_\to$$

$$\frac{\dfrac{\dfrac{\Gamma \vdash v : A \to B \mid \Delta}{\Gamma \vdash v : A \to B \mid \beta : B, \Delta}\ W \quad \dfrac{\dfrac{\Gamma \vdash w : A \mid \Delta}{\Gamma \vdash w : A \mid \beta : B, \Delta}\ W \quad \dfrac{}{\Gamma \mid \beta : B \vdash \beta : B, \Delta}\ CoVar}{\Gamma \mid w \cdot \beta : A \to B \vdash \beta : B, \Delta}\ L_\to}{\langle v \mid w \cdot \beta \rangle : (\Gamma \vdash \beta : B, \Delta)}\ Cut}{\Gamma \vdash \mu\beta.\langle v \mid w \cdot \beta \rangle : B \mid \Delta}\ Act$$

Where $W$ is the admissible weakening rule, since the $CoVar$ and $Var$ rules have arbitrary contexts $\Gamma$ and $\Delta$ we can modify the derivation on top by adding aribtrary judgements.

We will animate these expressions in the next section.

## 4.5 Reduction relation

There are two core rewrite rules:

$$\langle \mu\alpha.c \mid e\rangle \to_{\mu} c[\alpha := e] \qquad \langle v \mid \tilde{\mu}x.c\rangle \to_{\tilde{\mu}} c[x := v]$$

where substitute $c[\alpha := e]$ denotes substitution of variable $\alpha$ for $e$, and $c[x := v]$ denotes substitution of variable $x$ for $v$, both avoiding capture by suitable renaming. We write $c[x := v, y := w]$ for the simultaneous substitution of $x$ for $v$ and $y$ for $w$. The free occurrences of variables in the nested command are bound by the variable in the outer construct. A closed term has no free variables (of either kind) occurring.

**Example 8.** Given the term $\tilde{\mu}x.\langle \mu\alpha.\langle x \mid \alpha\rangle \mid \tilde{\mu}y.\langle y \mid \alpha\rangle\rangle$. This environment term is not closed, since $\alpha$ occurs free. However it also contains a bound $\alpha$, which we can freely rename to $\beta$: $\tilde{\mu}x.\langle \mu\beta.\langle x \mid \beta\rangle \mid \tilde{\mu}y.\langle y \mid \alpha\rangle\rangle$. There are two reduction paths to the normal form $\tilde{\mu}x.\langle x \mid \alpha\rangle$, namely one by reducing the inner command $\langle \mu\beta.\langle x \mid \beta\rangle \mid \ldots\rangle$ and the other reducing $\langle \ldots \mid \tilde{\mu}y.\langle y \mid \alpha\rangle\rangle$.

The rewrite system consisting only of the core rules is already not confluent, since there is a critical pair that is not necessarily joinable. An example is

$$\langle x \mid \tilde{\mu}y.\langle y \mid \alpha\rangle\rangle \leftarrow \langle \mu\alpha.\langle x \mid \alpha\rangle \mid \tilde{\mu}y.\langle y \mid \alpha\rangle\rangle \to \langle y \mid \mu\alpha.\langle x \mid \alpha\rangle\rangle$$

Confluence can be restored by prioritizing one rule over the other. A particular priority scheme is called a strategy. The calculus is parametric over its evaluation strategy. Two strategies are well-known: the call-by-value strategy always prefers a $\mu$-step, and the call-by-name strategy always prefers a $\tilde{\mu}$-step. Other strategies exists.

The rewrite rules are also extended for user-defined types. Again, instead of giving the generalized version, we will instead show it by example.

For terms of type $A \otimes B$ we have the reduction:

$$\langle (v, w) \mid \tilde{\mu}[(x, y).c]\rangle \to_{\otimes} c[x := v, y := w]$$

and for terms of type $A \to B$ we have the reduction:

$$\langle \mu(x \cdot \alpha.c) \mid v \cdot e\rangle \to_{\to} c[x := v, \alpha := e]$$

**Example 9.** Consider $A \otimes B \to A$. We show a term that inhabits this type, and its typing derivation: $\mu(x \cdot \alpha.\langle x \mid \tilde{\mu}[(l, r).\langle l \mid \alpha\rangle]\rangle)$.

Let $\Gamma = x : A \otimes B, l : A, r : B$ in

$$\dfrac{\dfrac{}{x : A \otimes B \vdash x : A \otimes B \mid \alpha : A}\,Var \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash l : A \mid \alpha : A}\,Var \quad \dfrac{}{\Gamma \mid \alpha : A \vdash \alpha : A}\,CoVar}{\langle l \mid \alpha \rangle : (x : A \otimes B, l : A, r : B \vdash \alpha : A)}\,Cut}{x : A \otimes B \mid \tilde{\mu}[(l, r).\langle l \mid \alpha \rangle] : A \otimes B \vdash \alpha : A}\,L_{\otimes}}{\dfrac{\langle x \mid \tilde{\mu}[(l, r).\langle l \mid \alpha \rangle]\rangle : (x : A \otimes B \vdash \alpha : A)}{\vdash \mu(x \cdot \alpha.\langle x \mid \tilde{\mu}[(l, r).\langle l \mid \alpha \rangle]\rangle) : A \otimes B \to A \mid}\,R_{\to}}\,Cut$$

Let us call above expression $\pi_1 : A \otimes B \to A$. We can now apply this function to an argument as follows: $\pi_1 \cdot (y, z)$ for variables $y : A, z : B$.

$$\begin{aligned}
\pi_1 \cdot (y, z) &= \mu\beta.\langle \pi_1 \mid (y, z) \cdot \beta \rangle && \text{(def)} \\
&= \mu\beta.\langle \mu(x \cdot \alpha.\langle x \mid \tilde{\mu}[(l, r).\langle l \mid \alpha \rangle]\rangle) \mid (y, z) \cdot \beta \rangle && \text{(def)} \\
&\to \mu\beta.\langle (y, z) \mid \tilde{\mu}[(l, r).\langle l \mid \beta \rangle]\rangle && \to_{\to} \\
&\to \mu\beta.\langle y \mid \beta \rangle && \to_{\otimes}
\end{aligned}$$

One can prove that the term is well-typed by deriving $y : A, z : B \vdash \pi_1 \cdot (y, z) : A \mid$.

In general, for every typable term our reduction relation is terminating. We additionally have type preservation, that is, if a term $t$ with type $\tau$ reduces into a term $t'$ then the typing judgment $t' : \tau$ is derivable.

## 5 Historical context

As mentioned in the introduction of [9], the type of double negation elimination is given to the control operator $\mathcal{C}$ due to Griffin [3]. The $\lambda\mathcal{C}$-calculus is due to Felleisen [1] and Hieb [2]. This extends the syntax of simply typed $\lambda$-calculus in two ways, namely by adding the type $\bot$, and by adding the constant $\mathcal{C}$ such that there are two kinds of applications: $(MN)$ and $(\mathcal{C}N)$ for terms $M$ and $N$.

Parigot found that a natural deduction system with multiple conclusions was more convenient to work with. In my mind, this already brings the resulting natural deduction system closer to sequent calculus. The result is called the $\lambda\mu$-calculus, which was developed in the quest of finding a suitable computational interpretation that corresponds to classical logic [10].

De Groote showed in 1994 that $\lambda\mathcal{C}$-calculus and $\lambda\mu$-calculus are isomorphic [11, 12]. A similar result was given in [13], where three classes of natural deduction and their relations are defined: minimal logic, minimal classical logic, and classical logic.

In $\lambda\mu$-calculus an additional binding term is introduced to $\lambda$-calculus, denoted $\mu\alpha.[\beta]M$ for $M$ any term. Named terms are $[\beta]M$ for any term $M$, and are themselves also considered terms. The calculus can be untyped or typed. Intuitively, we can understand the operator $\mu$ as being a $\lambda$ which potentially accepts an infinite number of arguments [9]: "The effect of the reduction of $(\mu\beta.u)v_1 \ldots v_n$ is to give the arguments $v_1, \ldots, v_n$ to the subterms of $u$ named $\beta$, and this independently of the number $n$ of arguments $\mu\beta.u$

is applied to:" the reduction rule mentioned here is defined as $(\mu\beta.u)v \to \mu\beta.u[v/^*\beta]$ where $u[v/^*\beta]$ replaces in $u$ each subterm $[\beta]w$ by $[\beta](w)v$.

In proof theory we can embed classical logic in intuitionistic logic by applying the double negation translation. In translating $\lambda\mu$-terms to $\lambda$-terms, this is called continuation-passing-style translation. According to [9], the difference between $\lambda$-calculus and $\lambda\mu$-calculus, is that only the latter allows infinite arguments whereas the same term translated back to $\lambda$-calculus does not.

We then move to sequent calculus. A goal behind earlier work was to find a "sequent calculus version" of Parigot's $\lambda\mu$-calculus, which resulted in the $\bar{\lambda}\mu$-calculus by Herbelin. Sequent calculus was originally developed by Gentzen to study cut elimination for first-order classical logic, and is considered by [5] to be more well-behaved than natural deduction. The $\bar{\lambda}\mu$-calculus is presented as a sequent calculus variant of $\lambda\mu$-calculus. $\bar{\lambda}\mu$-calculus is essentially the same as $\lambda\mu$-calculus, in the sense that it preserves normal forms, and that a homomorphism with respect to call-by-name evaluation exists. But it differs in the sense that cut elimination and reduction steps are no longer directly corresponding.

The $\lambda$-calculus and $\lambda\mu$-calculus have the property of confluence: the outcome of reduction does not depend on the chosen evaluation strategy. However, it might be useful to have direct control over evaluation strategy. This became apparent in $\bar{\lambda}\mu$-calculus, which only preserves call-by-name reduction of translated $\lambda\mu$-terms. The $\bar{\lambda}\mu$-calculus is extended with the dual of the $\mu$ binder, denoted as $\tilde{\mu}$. The result is called $\bar{\lambda}\mu\tilde{\mu}$-calculus, which no longer is confluent [5].

It turns out that precise control over evaluation strategy now is necessary to regain confluence. This calculus has two confluent subsyntaxes, one preserves call-by-name reduction of translated $\lambda\mu$-terms and the other preserves call-by-value reduction. The two sub syntaxes are dual, in the sense that a syntactic transformation exists between the two: this explains why some consider the evaluation strategies call-by-name and call-by-value to be duals.

The calculus that is considered here is the parametric $\mu\tilde{\mu}$ core with user-defined (co-)data types as given in [7]. We have derived $\lambda$-terms as instances of a user-defined type, and the usual typing rules can be mimicked by the typing rules that are instances of the given generalized system.

It seems that there is no clear reference to the origin of this generalized type system; it does not appear in [7], but in [8] it cites [7] as the origin. Also, in our version of [8], the type system contained some typos, but which turned out not to be of big importance for our interpretation. The reason for absence of reference turned out that [14] was not yet finished at the time of writing [7, 8]. It seems that [14] is the first complete description of the generalized type system.

## References

[1] Matthias Felleisen. The calculi of lambda-v-cs conversion: A syntactic theory of control and state in imperative higher-order programming languages. 1987.

[2] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

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

[4] Traian Florin Serbanuta and Grigore Rosu. A rewriting logic approach to operational semantics. Technical report, 2006.

[5] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *5th ACM SIGPLAN International Conference on Functional Programming*, volume 35, pages 233–243. ACM, 2000.

[6] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category theory and computer science*, pages 140–157. Springer, 1987.

[7] Paul Downen and Zena Ariola. The duality of construction. In *17th European Symposium on Programming Languages and Systems*, pages 249–269. Springer, 2014.

[8] Paul Downen, Philip Johnson-Freyd, and Zena M Ariola. Structures for structural recursion. In *20th ACM SIGPLAN International Conference on Functional Programming*, volume 50, pages 127–139. ACM, 2015.

[9] Michel Parigot. Classical proofs as programs. In *Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 263–276. Springer, 1993.

[10] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *3rd International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 190–201. Springer, 1992.

[11] Philippe De Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 31–43. Springer, 1994.

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

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

[14] Paul Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, University of Oregon, 2017.